TÉCNICO LISBOA

Crypto Cloud

Filipe Duarte Bento Custódio

Thesis to obtain the Master of Science Degree in

Information Systems and Computer Engineering

Supervisors: Prof. Ricardo Jorge Fernandes Chaves
Eng. Luís Carlos Alves Henriques

Examination Committee

Chairperson:	Prof. Ana Teresa Correia de Freitas
Supervisor:	Prof. Ricardo Jorge Fernandes Chaves
Member of the Committee:	Prof. Miguel Ângelo Marques de Matos

November 2017

Acknowledgements

I would like to express my sincere gratitude to my supervisors Ricardo Chaves and Luís Henriques for their continuous support. Their patience, motivation and immense knowledge provided me the proper guidance during the research and writing of this thesis.

I extend my gratitude to my colleagues at Multicert, especially to Pedro do Vale and to Carlos Cardoso, for their support, encouragement and motivation during this thesis.

Last but not least, I would like to thank my parents and my brother for their love and support throughout this thesis and in my life in general. This work would not be possible without them.

Lisboa, November 2017

Filipe Duarte Bento Custódio

For my parents,

Resumo

A utilização dos Serviços de Armazenamento na Nuvem teve um acentuado crescimento nos últimos anos. A migração para estes serviços despoletou o surgimento de vários riscos de segurança, tal como a falta de privacidade sobre os dados armazenados e a necessidade de confiar nas intenções dos provedores de serviços de nuvem. Para superar estes problemas, esta tese propõe o Crypto Cloud, um sistema seguro de armazenamento na nuvem baseado na abordagem do Storekeeper [20, 21]. Esta solução permite a utilização de vários provedores de serviços de nuvem sem renunciar a privacidade, garantido a confidencialidade e integridade dos ficheiros, o que origina novos desafios relacionados com a gestão de chaves e o controlo de acessos. Para abordar a gestão de chaves, o Crypto Cloud implementa o protocolo KMIP [18], que permite a gestão remota das chaves criptográficas dos utilizadores, libertando os utilizadores e os serviços de nuvem da responsabilidade de gerir essas chaves. O componente que implementa o protocolo KMIP tira vantagem de Módulos de Segurança de Hardware para manipular as chaves dos utilizadores e realizar operações criptográficas. Para garantir a certificação das chaves dos utilizadores e assegurar as suas identidades quando acedem a ficheiros, o sistema conta com uma Infraestrutura de Chaves Públicas que age como terceira parte confiável. A solução segue normas amplamente utilizadas em aplicações web, como os protocolos TLS/SSL [23, 27], para o estabelecimento de canais seguros de comunicação entre os intervenientes, e o protocolo OAuth [14], para controlo de acessos e autorização dos pedidos dos utilizadores.

Abstract

The use of Cloud Storage Services has had an accentuated growth over the past few years. With the migration of these services, several security risks have risen, such as the lack of privacy in the stored data and the need to trust the cloud providers' intentions. To overcome these problems, this thesis proposes Crypto Cloud, a secure cloud storage system based on the Storekeeper's approach [20, 21]. This solution allows for the use of multiple cloud providers without renouncing privacy, guaranteeing the confidentiality and integrity of managed files, which brings new challenges related to key management and access control. To address the key management, Crypto Cloud implements the KMIP [18] protocol, which provides remote management of the users' cryptographic keys, decoupling the users and the clouds from the responsibility of managing these keys. The component implementing the KMIP protocol takes advantage of Hardware Security Modules to handle the users' keys and perform remote cryptographic operations. To guarantee proper certification of the users' keys and to assure their identity when accessing the managed files, the system relies on a Public Key Infrastructure, which acts as a trusted third-party entity. The solution also follows standards widely used in web applications, such as the TLS/SSL [23,27] protocols, for the establishment of secure communication channels between the interacting parties, and the OAuth [14] protocol, for access control and authorization of users' requests.

Palavras Chave
Keywords

Palavras Chave

Serviços de Armazenamento na Nuvem

Segurança e Privacidade

Gestão Remota de Chaves

Interoperabilidade na Gestão de Chaves

Módulos de Segurança de Hardware

Keywords

Cloud Storage Services

Security and Privacy

Remote Key Management

Key Management Interoperability

Hardware Security Modules

Contents

1 Introduction **1**

1.1 Motivation 2

1.2 Goals 3

1.3 Requirements 3

1.4 Structure of the Document 4

2 Background **5**

2.1 Cryptographic Mechanisms and Protocols 5

2.1.1 Cryptography 5

2.1.2 Digital Certificates 7

2.1.3 TLS/SSL 8

2.1.4 OAuth Protocol 9

2.2 Public Key Insfrastructures 10

2.3 Key Management 11

2.3.1 Key Storage 11

2.3.2 Client-Server Model 13

2.3.3 Standards 14

3 Related Work **17**

3.1 BlueSky 17

3.2 SPORC 18

3.3 DepSky 20
3.4 SCFS 22
3.5 Storekeeper 23
3.6 Comparative Analysis 26

4 Crypto Cloud **29**

4.1 Overview 30
4.2 Architecture 31
4.2.1 Crypto Cloud Client Application 31
4.2.2 Crypto Cloud Directory Server 33
4.2.3 Key Management Server 34
4.2.4 Cloud Stores 34
4.2.5 Public Key Infrastructure 35
4.3 Models 35
4.3.1 Trust Model 35
4.3.2 Threat Model 36
4.3.3 Access Control Model 38
4.4 User Authentication 39
4.4.1 Crypto Cloud Authentication 39
4.4.2 Cloud Store Authentication 40
4.4.3 Key Management Authentication 41
4.5 Functional Algorithm 41
4.5.1 Base Algorithm 41
4.5.2 Version Control 43
4.5.3 File Sharing 44

4.5.4 File Homing 45
4.6 Summary 47

5 Implementation **49**

5.1 Overview 49
5.2 Data Structures 51
5.3 Protocol 53
5.4 KMIP 56
5.4.1 Architecture 57
5.4.2 Transport 60
5.4.3 Profiles 62
5.4.4 Decisions 63
5.5 Summary 64

6 Evaluation **65**

6.1 Performance Evaluation 65
6.1.1 Crypto Cloud Performance 66
6.1.2 KMIP Performance 69
6.2 Security Analysis 71
6.3 Compliance with Requirements 72
6.4 Summary 73

7 Conclusions **75**

7.1 Conclusions 75
7.2 Future Work 76

Bibliography **79**

Appendices **81**

A Protocol Procedures **83**

List of Figures

3.1 BlueSky's architecture (by Vrable et al. [33]). 17

3.2 SPORC's architecture (adapted from Feldman et al. [12]). 19

3.3 DepSky's architecture (by Bessani et al. [4]). 21

3.4 SCFS's architecture (by Bessani et al. [5]). 22

3.5 Storekeeper's architecture (by Pereira et al. [21]). 24

4.1 Crypto Cloud's system architecture. 31

4.2 Crypto Cloud Client Application's architecture. 32

4.3 Crypto Cloud Directory Server's architecture. 33

4.4 Key Management Server overview. 34

4.5 Example scenario to illustrate the base algorithm. 42

5.1 CCDS's database schema. 52

5.2 Main architecture of the Key Management Interoperability Protocol (KMIP) protocol's implementation. 58

5.3 KMIP User Manager interface provided by the admin module. 61

6.1 Crypto Cloud's mean latency and corresponding standard deviation per operation. 67

6.2 Evolution of revoke operation latency depending on the file's ACL size. 68

6.3 Evolution of KMIP encrypt operation's latency in result of the number of requests. 70

6.4 Evolution of KMIP decrypt operation's latency in result of the number of requests. 70

A.1 Read protocol flowchart. 85

A.2 Write protocol flowchart. 86

A.3 Sync protocol flowchart. 89

List of Tables

3.1 Comparison of existing solutions for secure cloud systems. 26

4.1 Crypto Cloud's permissions matrix. 39

Acronyms

ACL Access Control List

API Application Programming Interface

AT Access Token

CA Certification Authority

CCDS Crypto Cloud Directory Server

CIFS Common Internet File System

CoC Cloud-of-Clouds

CRL Certificate Revocation List

DBMS Database Management System

DHE Ephemeral Diffie-Hellman

ECC Elliptic Curve Cryptography

EKMI OASIS Enterprise Key Management Infrastructure

FK File Key

GUI Graphical User Interface

HMAC Hash-based Message Authentication Code

HSM Hardware Security Module

HTTPS Hyper Text Transfer Protocol Secure

IETF Internet Engineering Task Force

IK Integrity Key

IST Instituto Superior Técnico

IV Initialization Vector

JCA Java Cryptography Architecture

JSON JavaScript Object Notation

JVM Java Virtual Machine

KMIP Key Management Interoperability Protocol

KMS Key Management Service

MAC Message Authentication Code

NFS Network File System

NIST National Institute of Standards and Technology

OASIS Organization for the Advancement of Structured Information Standards

OT Operational Transformation

PKCS Public Key Cryptography Standards

PKI Public Key Infrastructure

PK Private Key

PU Public Key

RA Registration Authority

REST Representational State Transfer

RK Read Key

SDS Storekeeper Directory Server

SISWG Security in Storage Working Group

SQL Structured Query Language

SR Share Revision

SSL Secure Sockets Layer

TCG Trusted Computing Group

TLS Transport Layer Security

TPM Trusted Platform Module

TTLV Tag Type Length Value

UK User Key

URL Uniform Resource Locator

1 Introduction

Nowadays, the Internet is responsible for exchanging high volumes of information across the world. This information has to be stored in proper infrastructures, representing considerable management efforts for organizations. To decrease these costs, the concept of Cloud Computing was created. Cloud Computing results from the virtualization of physical infrastructures and its main goal is to provide high availability, flexibility and scalability of computing resources (e.g. processing, storage, applications) across the globe. These cloud resources offer elastic characteristics and can be provisioned on-demand by the users. The Cloud Storage Services provide new features to manage files, such as: file sharing, file versioning, concurrent access or disaster recovery. Given this, users and organizations started abandoning their private servers and migrating to cloud services. However, the migration to these services present some security risks, since users are forgoing their privacy and trusting their entire data to cloud providers, relying on their intentions and behaviour. Thus, most of the cloud providers have full access to users' files and do not apply any type of protection over them. Hereupon, the cloud services have been the target of massive attacks, with the objective of reading or modifying the stored data. For instance, in August 2014, hundreds of private photos of celebrities were leaked as result of a vulnerability in Apple iCloud's authentication mechanism [2]. Two years later, it was announced that 68 millions of user passwords from Dropbox were leaked on the Internet [3].

With this in mind, Instituto Superior Técnico (IST) in partnership with Multicert proposed the creation of Crypto Cloud, a secure cloud storage system. Crypto Cloud follows the Storekeeper's approach [21] and applies a novel key management scheme that introduces two new components to the existing architecture: a Key Management Server and a Public Key Infrastructure (PKI). These components are responsible for implementing the Key Management Interoperability Protocol (KMIP) standard [18], which allows the remote access and management of user's cryptographic keys, and certification of the users' identity, guaranteeing proper authentication when protecting and sharing sensitive files. The solution also follows widely used standards for web applications, such as the TLS/SSL [23, 27] or the OAuth [14] protocols.

1.1 Motivation

In the last years, the demand of users for higher capacity and availability of their resources lead to an accentuated growth of cloud providers. These systems started by providing simple storage services and now present a huge portfolio of services and functionalities. The users' dependency on these services created new challenges for the cloud, such as the security of the data. The providers are now searching for new ways of ensuring the security and confidentiality of stored and exchanged data.

Recently, some systems started to emerge in order to overcome the cloud security issues. Systems like BlueSky [33] guarantee data confidentially, however they rely on a single cloud storage and do not provide enough functionality to captivate the users. SPORC [12] provides data security mechanisms and various functionalities, such as data sharing and file versioning, yet they lack on mechanisms to support multiple cloud storages. To support multiple cloud storage services, systems like DepSky [4], SCFS [5], and Storekeeper [20, 21] started to appear, creating a new concept of Cloud-of-Clouds (CoC). A CoC combines multiple cloud storages and presents them to the user as a single one [8]. These systems also employ mechanisms to provide data confidentiality and support some cloud features (e.g. versioning, sharing), however they do not offer a rigorous key management scheme for the cryptographic keys. The management of the keys represent a critical component of a cryptographic system. This component is responsible for securing the complete key's lifecycle, including storage, use, and exchange. This leads to new challenges, such as: how to securely store the keys without compromising the accessibility, or how to implement a scheme considering a large number of users.

Considering the previous challenges, Crypto Cloud aims to develop a secure cloud storage system that supports multiple cloud providers and implements a robust key management scheme. The work of this thesis was developed in partnership with Multicert, a company that provides digital certification and security solutions, therefore some particular tools, models and standards need to be followed (e.g. security models and key management standards), according to the infrastructure of Multicert.

1.2 Goals

The main goal of Crypto Cloud, herein proposed, is to improve the performance and security of an existing secure cloud storage system, where users can securely use multiple cloud storage providers to store files, without trusting the cloud storage systems.

The second objective of this work is to improve the usability of the system in particular the user key management and usage. Considering this, a remote secure Key Management Service (KMS) supported by Hardware Security Modules (HSMs), is also used. To achieve this, open standards are followed, assuring high-compatibility with existing systems.

1.3 Requirements

To achieve the Crypto Cloud's proposed goals, the following requirements must be fulfilled:

Functional requirements:

1. Crypto Cloud must support the usage of multiple cloud storage providers;
2. Users should be able to store, access and share files without compromising their storage credentials;
3. The system must provide a secure way of storing the users' private keys;
4. The system must provide a mechanism to validate the public keys and associated identities of the users;

Non-functional requirements:

5. The system must be able to handle a large number of users;
6. Users must authenticate themselves in order to use the system;
7. The communication channels between the Crypto Cloud's components must be secure;
8. The implementation should follow open standards, serving a wide spectrum of enterprise equipment and applications.

1.4 Structure of the Document

Throughout this thesis, we describe the proposed secure cloud solution and related systems. This description is organized as follows. Chapter 2 presents the required knowledge to proper understand the problem and the proposed solution. Chapter 3 presents an overview of the related work. Chapter 4 describes the proposed solution, detailing its architecture, security models and functional algorithms. Chapter 5 describes the implementation details of our prototype. Chapter 6 presents the evaluation of the proposed solution and the analysis of the obtained results from performance and security perspectives. Finally, Chapter 7 concludes this document, summarizing the developed work and presenting improvements that should be taken in account for future work.

2 Background

This chapter presents the required background to proper understand the problem and the proposed solution. We first start by introducing some cryptographic primitives and protocols. After that, we explain the concept and the operation of a Public Key Infrastructure (PKI). This section is concluded by presenting the management models for cryptographic keys and introducing the existing standards.

2.1 Cryptographic Mechanisms and Protocols

This section introduces the cryptographic primitives and protocols designed to secure the web. Considering the scope of this thesis, the most used techniques and protocol in the Internet nowadays were considered, namely data protection using AES, RSA and HMAC cryptographic algorithms, establishment of secure communication channels through TLS/SSL protocols [23,27] and user authentication and authorization provided by the OAuth protocol [14].

2.1.1 Cryptography

Cryptography can be divided in tree main algorithms types: hashing, symmetric cryptography, and asymmetric cryptography. If properly applied, these cryptographic mechanisms can secure the data by providing confidentiality, authentication, authorization and non-repudiation.

Hashing (message digest)

Hash functions use mathematical operations to create a message digest (i.e. hash value) that represents a fingerprint of the input data [23, 27]. These functions respect the following properties: the same data will always produce the same hash value; similar data produces very different hash values; the hash value is not invertible, meaning that it is computationally

infeasible to obtain the original data. A hash function is also collision resistant, meaning that it is computationally infeasible to find two inputs that generate the same hash value. Hash functions can be used to check the integrity of the data (e.g. files, messages or passwords) by comparing a stored hash value with the hash value of the actual data.

Symmetric Cryptography

Symmetric cryptography uses a secret key to encrypt and decrypt data, transforming the input value (i.e. plaintext) into the encrypted data (i.e. ciphertext), and vice-versa [23,27]. This key can be shared by two or more entities, guaranteeing confidentially among all parties that possess the key. If the key is compromised, the data confidentiality will also be compromised.

The security level of symmetric cryptography depends on the algorithm being used, the size of the secret key and on the management of this secret key. The algorithm must guarantee that opponents cannot take advantage of the knowledge of the ciphertext and the algorithm used to discover the secret key or the plaintext. Currently, the AES algorithm is considered one of the most common encryption standards for symmetric cryptography.

Symmetric cryptography is usually used to encrypt data and in the creation of secure communication channels, where the key is shared between the sender and the receiver.

Asymmetric Cryptography

Asymmetric cryptography uses two different keys (i.e. public/private key pairs) to encrypt and decrypt the data respectively [23, 27]. The public key is used to encrypt data or validate signatures and can be distributed among users. Digital certificates can be used to validate public keys, increasing the trust among users (see Section 2.1.2). The private key is used to decrypt or sign the data and must be kept private. This key must be secured and properly stored, since all protected data can be revealed if the private key is compromised.

The security of this mechanism depends on the algorithm and the key used. The algorithm must assure that it is infeasible to determine the private key by knowing the public key and it is infeasible to recover the plaintext by knowing the public key and the ciphertext. Both keys are generated using one-way mathematical functions, meaning that despite of being related, it is

impossible to do the reverse process. Currently, the RSA standard is widely used for asymmetric cryptography on web applications.

Asymmetric cryptography is usually used to protect data, sign documents, establish secret keys for secure communications (see Section 2.1.3) and to authenticate users. This type of cryptography is slow, demanding more computationally power when compared to symmetric cryptography. To workaround this problem, a hybrid technique can be used, which consists of using a symmetric key to encrypt the data and an asymmetric key to protect (i.e. wrap) the the symmetric key.

Message Authentication Code

The Message Authentication Code (MAC) consists of an authentication technique, which uses a secret cryptographic key to calculate an authentication code of a received message [27]. This mechanism requires that both the sender and the receivers share the same secret key. When interacting, the sender can send the MAC result alongside with the original message in order to ensure the integrity and authenticity of the exchanged message. The receiver performs the same MAC calculation over the received message and compares the result with the received MAC result. By doing this verification, changes in the message can be detected.

The Hash-based Message Authentication Code (HMAC) is a particular type of MAC that uses hash functions and cryptographic keys, guaranteeing the same properties from the original MAC technique [27]. The cryptographic strength of HMAC algorithms depends upon the quality of the used key and the strength of the used hash function. The size of the HMAC function's result is the same as the result of the underlying hash function. The use of HMAC techniques is motivated by the fact that hash functions are widely available in several cryptographic libraries.

2.1.2 Digital Certificates

Digital certificates consist of a document including the public key and an ID that represents the owner of the respective key pair [22,23,27]. This information is signed by a third-party entity (i.e. Certification Authority (CA)) that must be trusted by the users. The users can register their keys on the CA to obtain the certificates. The certificate can be validated by checking the signature of the CA or verifying the certificate chain.

Certificates contain a period of validity and can be revoked under special occasions, such as: if the user's private key is compromised; if the CA's certificate is compromised; if the key is no longer certified by the CA. Certificates can be distributed by various manners, the most common is using PKIs, described in more detail in Section 2.2.

The X.509 standard has become the universal standard to format digital public-key certificates [23, 27]. This standard assumes that the certificate was created by a trusted CA. The X.509 presents a structured format that includes various elements to describe properties and functions of the certified public key, such as: the issuer name, the subject name, the period of validity of the certificate, the subject's public key, the CA's signature that covers all the certificate, and other fields.

2.1.3 TLS/SSL

Transport Layer Security (TLS) and Secure Sockets Layer (SSL) are cryptographic protocols designed to provide authentication and data encryption between two parties using insecure infrastructures [23, 27]. These protocols operate in the 7^{th} OSI Layer level, between the application and session layers. SSL was originally developed by Netscape. The early versions of this protocol were poorly designed and presented serious weaknesses. The 3^{rd} version of this protocol (SSL v3) was designed with the help of security experts and public review. This new version of the protocol was then submitted for Internet standardization. Subsequently, the TLS group was formed to develop the common standard, which is currently in version 1.2 (TLS v1.2). Considering the small differences between these protocols, the SSL and TLS terms are usually used in conjunction.

The TLS mechanism is based on a client-server model and is composed of two protocols: the handshake protocol and the record protocol. The first is responsible for establishing the connection parameters and authentication among the communication parties. The latter one is responsible for the compression, encryption and to control the integrity of the exchanged messages.

TLS supports different key exchange algorithms, including RSA, Ephemeral Diffie-Hellman (DHE) and Elliptic Curve Cryptography (ECC). RSA key exchange algorithm uses the server's asymmetric key pair to exchange the session's key. This approach does not provide perfect for-

ward secrecy, meaning that if the server long-term private-key is compromised, all past sessions' keys will also be compromised. DHE algorithm uses mathematical operations to achieve agreement on a common key over an insecure channel. This key can be ephemeral, meaning that it is regenerated on every execution of the exchange protocol and never saved on disk, guaranteeing perfect forward secrecy. A DHE variation using ECC can also be used. This variant has a low computational cost but is only supported by modern clients.

2.1.4 OAuth Protocol

OAuth is an authorization protocol, developed by Internet Engineering Task Force (IETF), which allows an application to obtain access to a web service on behalf of a user [14]. This protocol allows a user to delegate access to his resources without compromising his credentials. The OAuth protocol is widely used by services such as Google, Dropbox, Facebook, Twitter, among others to allow their users to share resources with third-party applications without the need of re-authentication.

The standard defines four roles for users and applications: the resource owner, the resource server, the client application and the authorization server. The resource owner represents the person or application that owns the resource that will be shared. The resource server is the server (e.g. API server) that stores the resources. The client application represents the entity requesting access to the resources owned by the user. The authorization server is the interface that approves or denies the client application's request to access the user's resources.

The protocol defines four modes of authorization, called *grant types*. In the *authorization code grant type*, the user (1) authenticates himself towards the authentication server, which (2) provides an authorization code. The user (3) introduces the authorization code in the client application and the client application (4) requests the authentication server for an access token. After this, the client application (5) uses the provided access token to access the user's resources stored at the resources server. The *implicit grant type* follows a similar flow. However, after the (1) user authentication step, the authorization server (2) provides an access token. The user (3) inserts the access token directly in the client application and the client application (4) uses that access token to access the user's resources. When using the *resource owner password credentials grant type*, the user (1) provides to the client application direct access to his credentials, (2) the client application uses the user's credentials to request the authorization server for a valid access

token and (3) uses the provided access token to access the user's resources. When using *client credentials grant type*, the client application (1) authenticates itself with its own credentials towards the authorization server, in order to (2) receive a valid access token. Then, the client application (3) uses the provided access token to access the resources (e.g. make API calls).

2.2 Public Key Insfrastructures

Public Key Infrastructure (PKI) is an infrastructure composed of a set of roles, policies and procedures created with the objective of enabling efficient and secure use of public keys [23]. The infrastructure is responsible for storing, managing and distributing digital certificates based on asymmetric key pairs.

A PKI is composed of three main entities: a repository, a Certification Authority (CA), and an user entity. The repository is the entity responsible for storing certificates and Certificate Revocation Lists (CRLs). The main purpose of the CA is to create certificates and CRLs. The user entity represents the users of the system (e.g. terminals, servers) and are responsible for using the services provided by the PKI. Users normally have two roles: subscriber role and verifier role. Subscriber users use their private keys and requests CAs for certification of his public keys. Verifier users are responsible for verifying the validity of certificates using the repository and the CRLs. PKIs can also have two optional identities: a Registration Authority (RA), responsible for assisting the CA in various administrative functions, such as registration of user entities, and a CRL issuer, responsible for publishing CRLs.

PKIs can establish trust relations between their CAs by issuing certificates of other CAs and by requesting certification of its public keys to other CAs. These relations can be done using a flat, hierarchical, mesh, or bridge model. The validation of certificates can be done by following the certification chain until a trusted CA is found.

Properly certified public keys provide higher trust level in the provided services, such as: encryption and/or authentication of documents or messages, authentication of users, initialization of secure communication protocols, web services, and mobile signatures.

2.3 Key Management

Key management is a scheme responsible for storing, protecting and accessing cryptographic keys, representing the critical component of a cryptosystem [32]. This scheme involves technology, tasks and people to deal with various processes, including generation, activation, transport, storage, and destruction of cryptographic keys. The adoption of an effective key management scheme is crucial for encryption services, as it helps to ensure confidentiality and integrity of data. This section will address secure key storage solutions, key management models based on the key's location and existing standards, considering cloud storage scenarios.

2.3.1 Key Storage

In this context, key storage implies the secure storage of cryptographic keys, including adequate access control in order to protect the keys from malicious users. To achieve this, storage solutions based on hardware must be used, providing extra security for the stored qualified keys. This section describes reliable hardware solutions for the cryptographic key storage.

Smartcards

Smartcards are composed of embedded integrated circuits that can transmit, process and store data [22,27]. These cards can easily be carried by the user. The cards contain a microprocessor that, for security purposes, can only communicate with specific readers and require the user to enter a secret code (e.g. PIN code or password) to establish the connection before using the card. The system uses hardware and software mechanisms to protect stored data against manipulation and unauthorized access. Smartcards run specific operating systems (e.g. Java Card) and can be programmable to provide the needed functionalities (e.g. deactivate a card when illegal use is detected). However, this system presents some disadvantages, such as: false sense of security, as the card can be easily stolen and the identity theft, and limited resources and bandwidth of I/O operations.

Smartcards provide the use and storage of symmetric keys for multiple cryptographic purposes. These cards can also store asymmetric key pairs and cryptographic certificates. Typically, the stored keys are used on the hardware device itself and never leave the token. This happens

especially with the private keys. Most smartcards also implement physical security measures to prevent attackers from physically retrieving the card's internal data.

Currently, these cards are used in several applications, including telecommunication, banking applications and electronic identification (e.g. eID cards).

Trusted Platform Modules

A Trusted Platform Module (TPM) is a standard developed by Trusted Computing Group (TCG) to design secure cryptoprocessors [28]. The TPM's architecture consists of a micro-CPU, a memory and an I/O communication interface. The micro-CPU can perform cryptographic operations, including encryption, decryption, signing, random number generation, and RSA key generation. The memory is used to store configuration registers, temporary keys and long-term keys. The TPM comes with one unique RSA asymmetric key pair that is stored in memory and cannot be accessed from outside of the TPM.

TPMs are usually devices built-into motherboards and represent the root of trust of the platform. These modules are usually used in computers to store cryptographic keys and ensure that the computer runs authenticated code on boot. These modules can also certify the device's configuration by signing its description with the TPM's private key. Newer TPM chips can be used together with third-party applications to provide disk encryption capabilities in order to protect data (e.g. Windows BitLocker Drive Encryption [16]).

Hardware Security Modules

Hardware Security Modules (HSMs) are tamper-resistant computing devices dedicated to cryptoprocessing [9,15,29]. These modules are responsible for storing, managing and controlling keys' lifecycles. HSMs can also perform cryptographic operations, including encryption, decryption, and signing without granting direct access to the stored keys. However, these modules present a limited storage capacity for the keys. To address this limitation, HSMs are able to wrap stored keys. The wrapped keys can later be exported and stored on external databases.

HSMs are usually connected to a network via TCP/IP or to a host via an expansion card or other ports. These modules interact with applications, protecting transactions and identities. To assume lawful usage, the user has to authenticate himself in order to interact with the system

(e.g via a secret). The connection between HSMs and applications must be secure. For remote connections, a secure channel must be used (e.g. SSL channel).

These modules are currently used in various sectors, including banking, payments or PKIs, to provide proper cryptographic key usage.

2.3.2 Client-Server Model

The following describes several key management schemes based on a client-server model [6]. These schemes are applied to a scenario where files are stored in an encrypted form using cloud storage services. The keys can be managed at the client side, server side, or on both sides using a key splitting technique.

Key Management at Client Side

In this approach, the cryptographic keys are managed and preserved at the client side. The cloud maintains data on an encrypted form and the cryptographic process is done entirely at the client side. The keys can be provided to the client by different means, such as, local computer, smartcards, tokens, mobile phones. The user has total control of the keys and symmetric keys can be used.

Key Management at Server Side

This approach considers the use of asymmetric key pairs. The user's private key is maintained at the client side and the public key is stored in a central server. The data stored on the cloud is ciphered using a symmetric key. This key is wrapped using the client's public key so the client can decipher it with his matched private key. The central server is responsible for managing the users' public keys and distribute them among other users, allowing the sharing mechanisms to work. The client must arrange a secure way of storing and accessing his private key without compromising it.

Key Management at Both Sides (Server and Client Side)

In this scheme, the key splitting technique is used to split the key in two partial keys. One partial key remains at the client and the other is maintained at the server. The data is stored encrypted and to perform the cryptographic operations, both keys need to be combined. Both sides of this scheme do not store the entire key, meaning that if one side is compromised, the key and the data still remain secure.

Key Splitting Technique

The Key Splitting Technique was first introduced by A. Shamir [24] to help in the management of cryptographic keys. Shamir presented a secret share scheme, called (k, n) *threshold scheme*, where a secret S can be divided into n pieces, so that the knowledge of $k-1$ pieces cannot reveal any information about the secret. However, the scheme allows the secret to be easily revealed when k or more pieces are known [24]. For instance, if we consider a (k, n) *threshold* where $n = 2k - 1$, we get a robust key management scheme since it is only possible to recover the secret if $\frac{n}{2} + 1$ pieces are known. If half of the pieces $(\frac{n}{2})$ are compromised, the secret remains secure.

2.3.3 Standards

Key management is a complex, difficult and risky process. As previously stated, key management is a crucial component for cryptographic services. With the actual growth of cloud applications and security services, the need for an effective key management scheme increased. Multiple vendors started developing their own corporate implementations of KMS and interoperability issues between systems started to appear. To address these problems, various committees have been created with the objective of proposing open standards.

Initially, some recommendations and standards have been proposed, such as OASIS Enterprise Key Management Infrastructure (EKMI), Security in Storage Working Group (SISWG), or National Institute of Standards and Technology (NIST) recommendations, but they failed to reach the vendors' attention. Currently, some standards are becoming widely accepted by vendors, who are using them in their systems. The following describes these standards.

Public Key Cryptography Standards #11

Public Key Cryptography Standards (PKCS) is a group of agreements published on the late 90s by RSA Laboratories for cryptography implementations [10, 19]. This set is currently composed of 12 standards, each one focused on a specific cryptographic mechanism.

PKCS#11 is a standard that defines an interface between applications and cryptographic devices, also called Cryptoki Application Programming Interface (API). The API follows an object-based approach, and can be implement in numerous programming languages and applied for multiple cryptographic devices, such as, smartcards, USB tokens, HSMs and others. Since 2013, PKCS#11 is the property of the Organization for the Advancement of Structured Information Standards (OASIS) and was renamed to OASIS PKCS#11 Technical Committee.

PKCS#11 describes several elements, such as: slot, token, object and session. A slot represents a socket or a device reader (e.g. card reader). The token is a cryptographic element (e.g. smartcard) that can be inserted into the slot and contains data as objects form, such as keys or a certificate. These objects can be public or private and associated to attributes. To access the token, a session has to be established between the corresponding slot and the application. An unauthenticated session only grants access to public objects. After authenticating the session with a PIN code, access to public and private objects is granted. Once the session is established, the API allows to reference the objects.

PKCS#11 is a well-established and adopted standard, being the main chosen standard for operating with HSMs, however some vulnerabilities have been discovered (e.g. a local host with malicious code can easily intercept the PIN code provided at the authentication process). The standard states that keys marked with the sensitive attribute cannot be transferred from the token and cannot be compromised ("a key that is sensitive will always remain sensitive" [19]). However, as a result of some conflicting properties in certain attributes, several attack methods that violate this property have been discovered. A list of the known vulnerabilities was published by J. Clulow [7], as long as possible solutions for these weaknesses.

The use of PKCS standards in industry stimulates interoperability between various vendors and applications. However vendors usually change the core of the API to remove known vulnerabilities, increasing his security level.

Key Management Interoperability Protocol

KMIP is a standard published by OASIS in 2010 that intends to achieve interoperability across different key management implementations [18, 25]. The open standard was designed to fully support the cryptographic key lifecycle and establish a communication interface between the KMS and client services. In order to maintain the continuity of the project, OASIS created a technical committee, currently composed of more than thirty companies, such as: Cisco, EMC, Oracle, Cryptsoft, HP, IBM, SafeNet and others.

The KMIP protocol is based on a client/server interaction and supports multiple key generators. The protocol is designed for easy maintenance and auditing. OASIS openly provides the protocol's specification, profiles and use cases, as well as the organization also runs interoperability tests among different implementations.

To date, each system had proprietary protocols, forcing organizations to use various key management tools for different purposes. This required extra efforts in staff training, software implementation and operation. KMIP solves this issue by proposing a single common protocol that allows for a key infrastructure to communicate with any KMS and any encryption service. By following this, KMIP results in reduce costs, lower complexity and more efficient encryption systems. However, open-source implementations of the protocol are still under development.

The KMIP's managed objects can be personalized with attributes. The protocol stipulates methods to get, add, modify or delete attributes. Attributes can represent various characteristics of an object, such as, unique identifier, name, type, group, cryptographic details, state, usage limits, and others.

The communication channel between servers and clients must use TLS in order to maintain confidentiality and integrity properties. KMIP requests (excepting queries) may also require authentication. The protocol supports the complete key lifecycle with the operations: create, register, locate, get, regenerate, recovery, archive, revoke and destroy operations along with MAC, hash, cipher, decipher, sign and generate random numbers functions. Functions to query participants about their capabilities are also available in the framework.

In addition, this standard also defines the message encoding format, specifies the expected results for certain use cases and describes proper error handling that may occur during the interaction.

3 Related Work

In this chapter, we detail relevant systems that were developed with the main objective of providing secure cloud solutions. We start this presentation by exposing their approaches in detail and describing their features. To conclude, an overview of the state-of-the-art and a comparative analysis are presented.

3.1 BlueSky

BlueSky [33] focuses on replacing traditional network file services with commodity cloud services at an enterprise level without compromising user's data. Thus, one of the system's goal is to provide data confidentiality and integrity. BlueSky aims at transparent proxy-based architecture, where a proxy server will provide the illusion of a single traditional file server. The cloud replication and elastic capabilities, associated with lower costs, provide a proper solution to persistently store data. The authors assume that users keep a safe backup of keys, consequently no cryptographic keys are shared with the cloud.

The proxy is the main component of the BlueSky architecture (see Figure 3.1). It supports Network File System (NFS) and Common Internet File System (CIFS) network file systems and

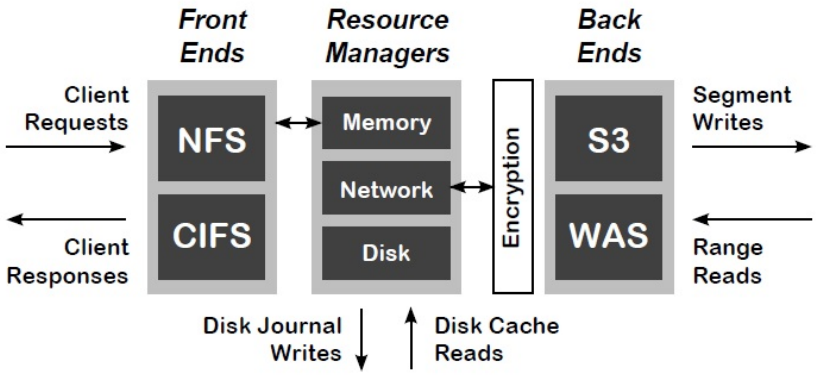

Figure 3.1: BlueSky's architecture (by Vrable et al. [33]).

can translate the requests into appropriate Amazon EC2/S3 and Microsoft's Azure calls. Clients can access the files by simply mounting BlueSky proxy file system through the previous protocols. The proxy is responsible for protecting the data before uploading to the cloud, therefore it uses AES for data encryption and HMAC-SHA-256 for data integrity protection. In order to reclaim unused space, the proxy maintains a log structure and implements a garbage collector system. The proxy maintains a local cache on disk and implements a write-back caching mechanism which sends data asynchronously to the cloud, increasing the performance of the system.

BlueSky system uses different objects to represent data and metadata. The data is encrypted and is represented using datablocks and inodes. The system does not encrypt metadata, which is represented using inode maps and checkpoints. The datablocks are used to store files data, which is split into 32 KB blocks, instead of the usual 4KB blocks of normal file systems. This partitioning decreases the system's overhead [33]. Inodes are responsible for storing basic metadata, like ownership, access control, timestamps, etc. The Inodes objects can also include a list of pointers to datablocks and directory entries, in order to reduce overheads originated from path transversal. Inode maps are used to locate inodes, pointing to the most recent version of each inode in the log file. The checkpoints objects are used to maintain file system integrity in case of failures. This object stores pointers to the location of the current inode maps and can also be used to provide versioned backups. On initialization, the system will read the log backwards and search for the most recent checkpoint.

3.2 SPORC

SPORC's [12] approach takes advantage of the high availability, fault tolerance, elasticity and scalability of cloud platforms without trusting on their cloud providers. It is assumed that the cloud resources can be infected or act malicious, thus the application's server only stores data on an encrypted form and cannot have abnormal behaviour without being detected.

SPORC is a generic framework developed with the objective of allowing creation of collaborative applications using untrusted servers (e.g. collaborative edition of files by a group of users). Additionally, it was designed with the following goals: keep data confidential from the server and unauthorized users, propagate shared state's changes quickly, tolerate slow networks or offline clients, detect a misbehaving server and recover from it.

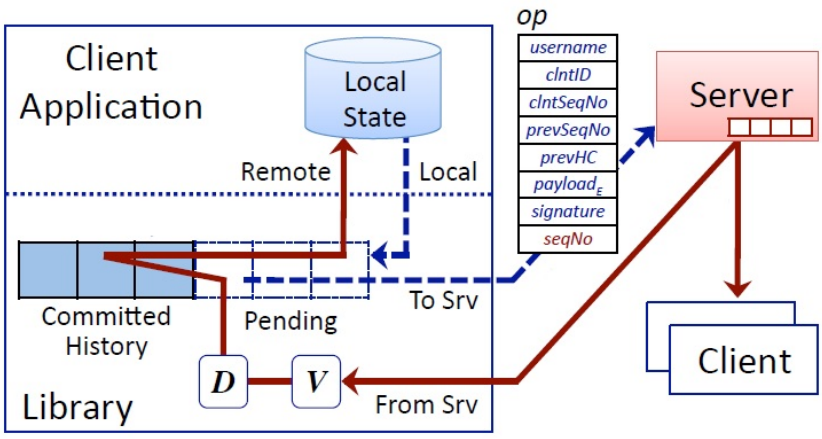

Figure 3.2: SPORC's architecture (adapted from Feldman et al. [12]).

SPORC's architecture, depicted in Figure 3.2, is composed of an untrusted server and clients. The untrusted server will order operations and distribute them through the clients. Clients maintain a local state of documents (i.e. files) and exchange two types of operations: document operations, that represent modifications in the document's content, and meta-operations, which correspond to document metadata's modifications, such as changes to the document's Access Control List (ACL). Modifications of the same document by multiple clients can lead to concurrent operations, resulting in conflicts. These conflicts can be automatically resolved using Operational Transformation (OT) without the need of using locks. New operations generated by clients are applied to its local state before uploaded to other clients. OT's conflict resolution mechanism preserves causal consistency and transforms received operations before applying them to the client's local state, leading to a consistent state between clients of the same group. The server acknowledges operations by setting unique sequence numbers to them and after that, these operations are considered committed. To prevent the central server of acting malicious and forging or modifying operations, every operation is digitally signed with their creator's private key. To ensure that the server does not provide different views of the operations' history to different clients, clients generate a hash chain over the history of operations. When a client creates a new operation, it includes the actual hash chain and a client sequence number. Once a client receives an operation, the actual hash chain is compared with the received chain and the consistency of operations is guaranteed.

SPORC assumes that users have a secure way of storing asymmetric key pairs, therefore the framework does not provide authentication mechanisms. However, the framework provides

a Document Membership Management system with three privilege levels: reader, editor, and administrator. Readers can only read documents, while editors and administrators can both read and submit new operations to the document. Administrator users can also add or remove participants from a document by sending meta-operations. ACLs are maintained at server and client side to prevent misbehaved conducts by both. When a document is created, an AES symmetric key will be randomly generated and used to encrypt document operations under AES algorithm before uploading. Asymmetric key pairs are then used to distribute AES keys between the same document's members. Every time a user is removed from a group, the correspondent document's key has to be changed and redistributed, ensuring that removed participants cannot decrypt subsequent operations.

In order to build collaborative applications using this framework, it is necessary to import the libraries that implement the SPORC protocol, define a data type for SPORC operations and how they should be transformed and combined. The central server of the application does not need to be modified.

3.3 DepSky

DepSky [4], depicted in Figure 3.3, is a CoC storage system that overcomes individual cloud storage systems' limitations, such as: loss of availability, loss and corruption of data, loss of privacy, and vendor lock-in. DepSky deals with the first two limitations by employing a Byzantine fault-tolerant replication scheme and using multiple cloud providers to store data. DepSky assures confidentiality by encrypting data before storing on cloud. The vendor lock-in limitation is solved by storing a fraction of the total amount of data in each provider, which originates balanced accesses and a decreased cost of migration data to other providers.

The system consists of a client application that uses cloud servers as passive storage servers, so no code is required to run at the server side. DepSky relies on cloud providers' access controls, requiring that each user has to use its own credentials. Thus, sharing is only possible if multiple users use the same account. The system's protocol supports files versioning and uses the providers' APIs to communicate with the servers.

The replication scheme of DepSky requires a number of servers (clouds) n, such as that $n \geqslant 3f + 1$ in order to tolerate f faulty servers. The key distribution mechanism is performed

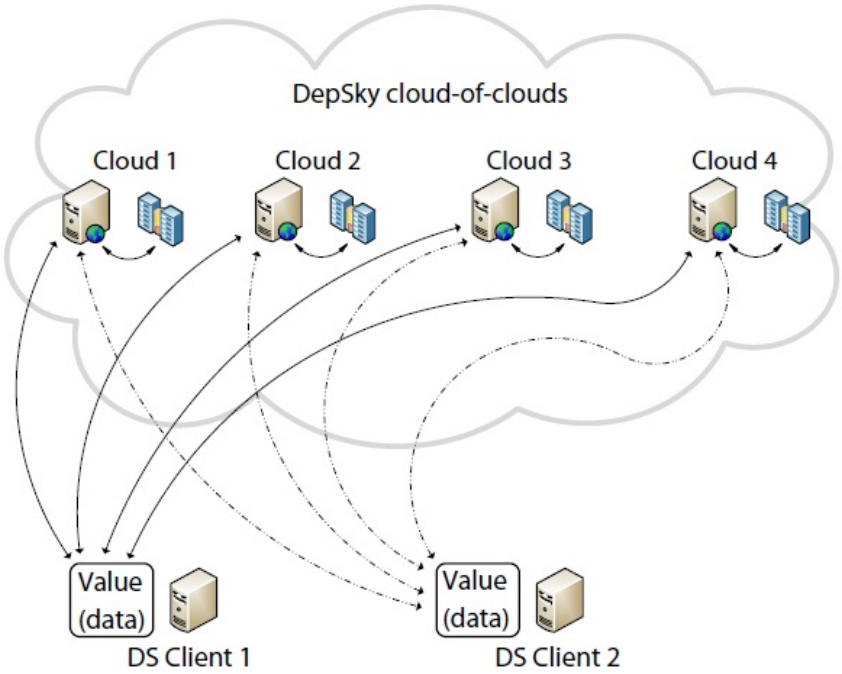

Figure 3.3: DepSky's architecture (by Bessani et al. [4]).

by a secret sharing scheme that is combined with the replication mechanism, where a dealer distributes a share of a secret to k players (clouds), such as $k \geqslant f + 1$. The secret can only be revealed by combining $f + 1$ shares and combinations of f shares cannot reveal any information about the secret, preventing data disclosure. Each cloud maintains metadata files, that are signed by its creators, responsible for storing data file's meta content, such as: version number, data pointer, and cryptographic details (e.g. signature, secret's share, etc.). In order to read data, a user asks all clouds for the file's metadata and waits for $n - f$ metadata files. Having done that, the user verifies the signature reads the version number included in the metadata file and asks all clouds for the correspondent data of that version. After receiving the data, the user compares the cryptographic details of the metadata and the actual data file and reads it if matched. When a user wants to write a file, he asks all clouds for metadata and waits for $n - f$ correct metadata files. The user writes the new file's version to all clouds and waits for $n - f$ acknowledgements. Done that, he updates the metadata file, incrementing the version number and sends it to all clouds. When the user gets $n - f$ acknowledges of the metadata update operation, the write process is considered completed.

3.4 SCFS

SCFS [5] is a cloud-backed file system that addresses typical cloud storage and cloud-backed systems limitations, such as: single point of failure, sharing and provider's trust. SCFS deals with these limitations by allowing the use of redundant cloud services, employing a controlled sharing service, and using a secret shared scheme (similar to DepSky [4]).

SCFS is mostly based on a client side and its architecture is composed of a backend cloud storage, a coordination service, and a SCFS Agent (see Figure 3.2). The backend cloud storage can be constituted by single or multiple clouds and it is only used to store data. The Coordination Service is responsible for metadata storing and management, access control and lock control. This component provides synchronization, sharing and multi-versioning services, and must run on some sort of computational servers (e.g. virtual machines, computing clouds, and others). The SCFS Agent implements most of the SCFS functionality and is responsible for mounting the file system at the client machine.

SCFS guarantees strong consistency by the concept of consistency anchor. Consistency anchor's algorithm will generate an id and a hash code for a certain version of a data object and store them together. Additionally, the id and hash code are duplicated and stored on metadata files. SCFS ensures data confidentiality, integrity and availability by implementing the DepSky's secret shared scheme to distribute keys and protect data files. SCFS can also provide high-scalability by using byzantine fault tolerant replication scheme using multiple cloud providers.

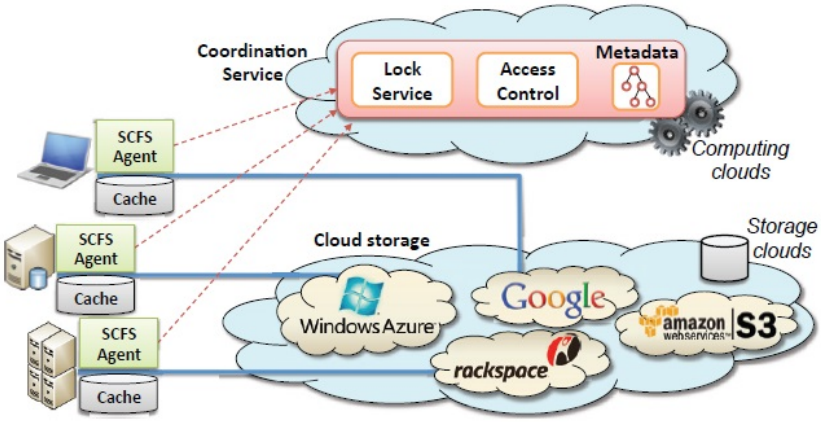

Figure 3.4: SCFS's architecture (by Bessani et al. [5]).

The system implements multiple mechanisms to improve performance, such as: separation of data, caching and garbage collection. Separation of data and metadata among distinct services allow parallel access to files. The caching mechanism maintains the entire data objects in memory to locally perform read and write operations, which will lead to decreased cloud accesses and consequently decreased cloud costs. When a cached object suffers write operations the agent has to upload the object to the cloud. Also, the agent has to check the cached objects' freshness on every read operation. The check can be done by requesting the coordination service for the correspondent metadata and posterior comparing hashes. The garbage collection mechanism runs at an isolated SCFS Agent's thread and reclaims used space based on two parameters: number of written bytes to keep and number of versions to keep. These parameters can be individually set by the user.

SCFS system can be configured to operate in three different modes: blocking, non-blocking and non-sharing. The blocking mode send the file immediately after a close instruction. The non-blocking mode uploads the file asynchronously after the close instruction. The non-sharing mode disables the SCFS sharing mechanisms.

3.5 Storekeeper

Storekeeper [20,21] is a cloud storage aggregator that guarantees data confidentiality without compromising cloud user's credentials - i.e. users do not have to share their account credentials with servers or other users. However, the system does not preserve data integrity and availability. Storekeeper application aggregates multiple cloud storage services (e.g. Dropbox or Google Drive), presenting them as a single storage workspace, and enables secure file sharing between users without requiring individual cloud accounts. In terms of consistency semantics, the system adopts an eventual consistency model, allowing file versioning. Write conflicts are manually resolved by the user.

The system follows three design principles: users must be restricted to write on their accounts only, users must have their files physically located on their accounts, and users do not need to maintain a persistent state at the client side. By ensuring the first principle, phenomenon known as "free ride", where users can use other users' cloud space, is prevented. The second principle ensures that users keep full control of their files, even though the files are shared

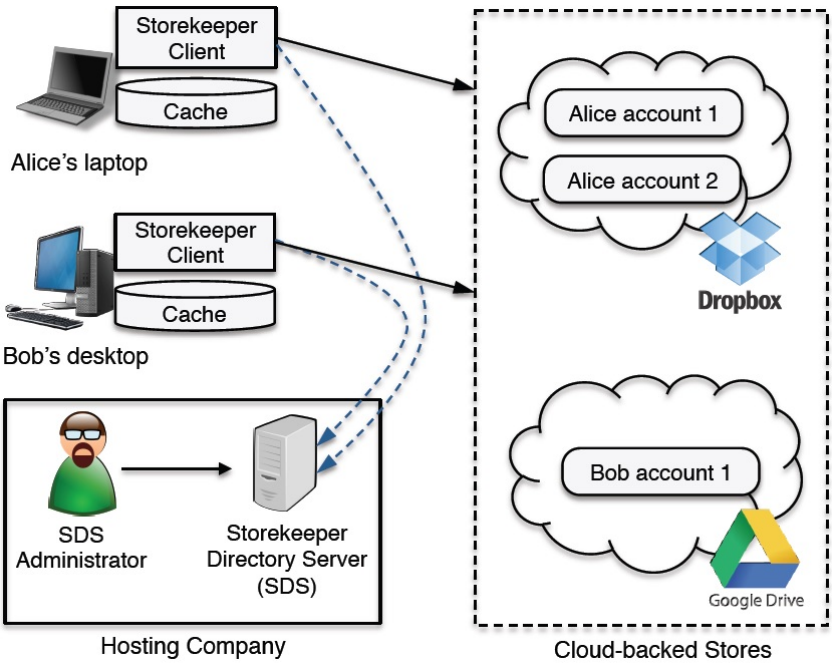

Figure 3.5: Storekeeper's architecture (by Pereira et al. [21]).

with other users. The last principle frees the users from the responsibility of keeping all persistent state of the application (e.g. private keys) preventing data loss and resulting in a more convenient service.

Storekeeper is composed of two main components: a client application and the Storekeeper Directory Server (SDS), as depicted in Figure 3.5. The first, is an application that has to be installed on the user's device and is responsible for accessing the cloud and maintaining a local cache of the user files. The last, runs on a server and represents the core of Storekeeper, which is responsible for managing metadata, users, files and cloud services. The SDS is assumed to be honest but curious (i.e. can listen to messages and learn information, but follows the protocol and does not launch active attacks) and does not access cloud storage spaces, nor store user files. The communication between the client's application and the SDS is done using SSL channels, authenticating the SDS server and ensuring integrity and confidentiality of exchanged messages.

Application users are registered in the system by an administrator. After logged in the system, the users have to register their cloud accounts in the client application by entering access tokens, without having to share their credentials with server or other users. These access tokens are generated by cloud providers and are used by third-party applications (e.g. Storekeeper client) through their respective APIs. The system also generates and associates a pair

of RSA asymmetric keys to the user, for later use in data protection mechanisms. The user's correspondent private-key pair and the user's access tokens are protected together with a cryptographically secure password (or key) that are only known to the user. These credentials are then stored at the SDS for convenience, relieving the user from maintaining persistent state of the application.

Storekeeper internally implements an access control mechanism based on ACLs where users can have three privileges over a file: read, write, and share. Read privileges only allows the user to read the file. Write privileges allows both read and write operations (i.e. create, update and delete the file) over the file. Share privileges allow to read, write and shares a file with other users (i.e. change permissions of the file). Exceptionally, the file's owner has full access privileges. The SDS will do ACL checks on every request to enforce access control policies.

The system guarantees end-to-end confidentiality by encrypting files with AES symmetric keys at the client side. This file encryption key (KF) is later secured by being encrypted using the owner's public-key pair of RSA asymmetric keys, assuring that only the owner can access it. The file sharing service is implemented by following a simple protocol: when a file's owner wants to share a file to other user, a symmetric key is generated and used to encrypt KF. This key, called read key, is then encrypted using the other user's public key, allowing the user to decrypt KR and KF, and consequently decrypt the file's content. Every time a revocation occurs, a new read key (KR) have to be generated and distributed among the new set of users using their public keys. Whenever a writer submits a file update, a new file encryption key (KF) is generated and used to encrypt the new content of the file. This key is then encrypted with the current KR and distributed among the group's users using their private key. As a result of this method, revocation is achieved, since a revoked user will not be able to read future updates. The system also has to update the ACLs at the SDS on every file's group members change.

The access to files is done via an Uniform Resource Locator (URL), allowing reads to be performed. Write operations via URL are not allowed by the cloud providers. The SDS is responsible for saving files' access URLs in metadata objects. The system allows cross-reads (i.e. reads of files located on other user's space) but prevents cross-writes (i.e. writes in files located on other user's spaces). Whenever a user with W privileges wants to update a file owned by other user, he writes the file in his own cloud space. The SDS will then update the metadata of the file, changing the URL to point to this new version. Old metadata versions

are still maintained at the SDS. As a result, the free ride phenomenon (i.e. when users can abuse from other user's storage space) is prevented. In the worst case, this technique comes with a cost of potentially losing the last update if the user deletes his account. To overcome this problem, storekeeper can periodically relocate these versions to the owner's cloud space and update the correspondent URL at the SDS. Storekeeper also implements a simple garbage collection mechanism to delete stale files (i.e. files that are no longer managed by the system but still occupy users' cloud space) that result from the previous technique.

The Storekeeper Directory Server (SDS) is a potential bottleneck to the scalability of the system and represents a single point of failure. To improve availability and assure durability, a master-slave backup configuration can be deployed.

3.6 Comparative Analysis

Table 3.1 summarizes the features of the previous presented approaches. These systems are compared based on their data and metadata protection properties, sharing and file versioning mechanisms and key distribution schemes.

System	Data Sharing	Data Confidentiality	Data Integrity	Data Versioning	Metadata Confidentiality	Metadata Integrity	Key Distribution	Multi-Cloud
BlueSky [33]		✓	✓	✓				
SPORC [12]	✓	✓	✓	✓	✓	✓	✓	
DepSky [4]		✓	✓	✓		✓	✓	✓
SCFS [5]	✓	✓	✓	✓			✓	✓
Storekeeper [21]	✓	✓		✓			✓	✓

Table 3.1: Comparison of existing solutions for secure cloud systems.

BlueSky [33] provides data confidentiality using their proxy to perform the encryption of the data. Data integrity and file versioning features are guaranteed by performing regular checkpoints and logging operations. However, this system relies on a single cloud to backup local storage and does not provide sharing services.

SPORC's [12] approach achieved all the security properties using a simple scheme: the confidentiality is guaranteed using symmetric keys and the integrity is achieved by signing documents' operations with asymmetric keys. This system also implements sharing, versioning and key distribution mechanisms. However, this system relies on a single cloud.

DepSky [4] guarantees integrity and availability properties by implementing a byzantine

fault-tolerant quorum system. The approach also provides data confidentiality, by using symmetric keys, and data sharing, using a secret share scheme to distribute file keys. The file versioning is assured by the metadata objects.

SCFS [5] presents a similar approach to DepSky, using byzantine fault tolerant replication and an analogous secret sharing scheme. This system allows the use of single or multiple clouds. However, authors do not state if users have to trust the system with their cloud credentials and how the access is done.

Storekeeper [20,21] supports multiple cloud providers and is supported by a directory server to store metadata. It is assumed that this server does not launch active attacks, thus metadata protection was not part of the project's scope. The users can use their personal cloud accounts and an API token is used to protect users from trusting their credentials to the server. The system guarantees data confidentiality using symmetric keys, that are distributed to users using public-key pairs. The approach also provides sharing and access control mechanisms. However, this approach does not consider data integrity and employs a non-efficient key management mechanism that performs extensive number of requests to the SDS, resulting in a decrease of the application's performance.

4 Crypto Cloud

The previous chapter introduced several cloud storage solutions that provide security, redundancy and isolation over different cloud providers. All these systems present distinct characteristics, however none of them fulfil the set of requirements in Section 1.3. Even though the solution presented by Storekeeper [21] does not fulfil all the requirements, it is the most complete one given the goals, and its implementation is open source. Thus, the proposed solution is based on Storekeeper. Storekeeper supports the aggregation of public clouds and allows collaboration between users, providing files' sharing mechanisms without disregarding files' security. The solution is capable of handling a large number of users without compromising their cloud credentials. This is done by following a novel approach that uses authorization tokens to access users' cloud resources. The system is capable of preventing information disclosure from stored files, provides a key distribution scheme and implements authentication and access control mechanisms. However, Storekeeper assumes that the Storekeeper Directory Server (SDS) follows the protocol and does not act maliciously, which presents some fragilities. If the SDS is attacked and start acting maliciously, the integrity of the stored files and the users' public keys can be tampered, since this metadata is managed only by the SDS and the system does not provide any mechanism to detect these violations. Also, the SDS does not implements a robust access control mechanism capable of preventing identity spoofing from users who start acting maliciously after successful authentication towards the system.

In this chapter, we introduce Crypto Cloud, an improved version of Storekeeper. We start by presenting an overview of our proposed solution (Section 4.1). Then, we present the proposed architecture, detailing each of its components (Section 4.2). Following this, we describe the security models that were considered during the design phase (Section 4.3). Next, we describe different user authentication mechanisms herein proposed (Section 4.4). After that, we introduce the algorithms responsible for supporting the system's functionality (Section 4.5). To conclude, we summarize all aspects of the proposed solution (Section 4.6).

4.1 Overview

This section presents an overview of the proposed solution for a secure cloud system. Crypto Cloud is based on the Storekeeper's concept, focusing on overcoming the previous identified weaknesses. The Crypto Cloud Directory Server (CCDS) is a revised version of Storekeeper's SDS, in which some mechanisms, such as the authentication and access control, were reshaped. Crypto Cloud follows the OAuth protocol [14], which widely used in web and mobile applications to authorize users' requests after successful authentication. The OAuth protocol is implemented by the CCDS, guaranteeing that an authenticated user is only authorized to manage his files. Each of the Crypto Cloud's users hold a unique asymmetric key pair, which includes a Private Key (PK) and its corresponding Public Key (PU) that are used to protect (i.e. wrap) the symmetric keys used to secure the Crypto Cloud's files.

In order to enhance the key management scheme, Crypto Cloud introduces two new components to the existing Storekeeper's architecture, namely the Key Management Server and a PKI. The Key Management Server is a remote server that implements the KMIP protocol, as recommended by Multicert, and is responsible for the access, storage and management of the users' cryptographic keys. This component allows the client application to obtain the user's key and perform the cryptographic operations locally or send a KMIP request to perform the cryptographic operation remotely. The PKI component acts as a third-party entity responsible for the certification of the users' cryptographic keys, establishing a linkage between the user's identity and his PU. With the addition of these new components, it is possible to achieve a robust key management scheme for the users' cryptographic keys without relying in the CCDS.

In terms of functionality, Crypto Cloud implements an augmented version of Storekeeper's functional algorithms. The basic algorithm was refined in order to provide integrity protection over stored files without relying in the CCDS to store the hash value. The file sharing algorithm follows the same approach, however, the validity of users' certificates is checked before triggering the sharing mechanism. The file homing algorithm was enhanced, applying a garbage collector mechanism capable of releasing cloud storage space from stale versions of managed files.

4.2 Architecture

This section presents an overview over Crypto Cloud's architecture, illustrated in Figure 4.1. The system consists of five components: the client application, responsible for the system's main functionality; the Crypto Cloud Directory Server (CCDS), which manages the system's metadata; the Key Management Server (KMIP Server), for the access and management of cryptographic keys; and the Cloud Stores, which are responsible for providing users' storage space. Crypto Cloud also takes advantage of a PKI to certify and validate the users' public keys. The interaction between the components is secured using the TLS communication protocol, which is widely used on the web, ensuring confidentiality, integrity and authentication of the exchanged messages.

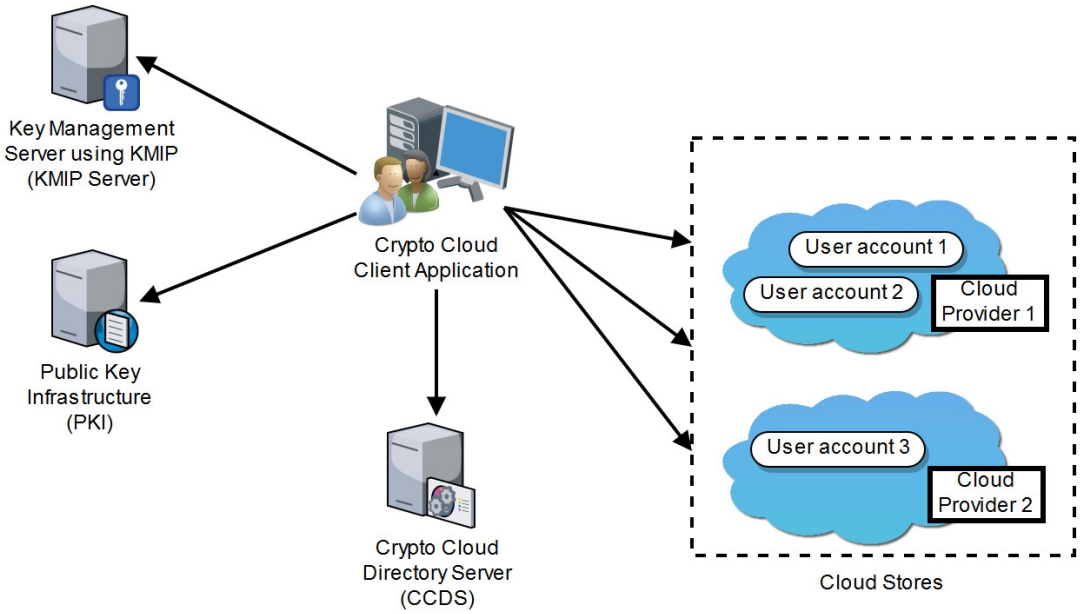

Figure 4.1: Crypto Cloud's system architecture.

4.2.1 Crypto Cloud Client Application

The client application is the central component of the system. It is responsible for managing the user's files and interacts with all other Crypto Cloud's components. The application allows authenticated users to upload and download files, as well as to manage their access permissions. It is also possible to perform special operations, such as the generation of a replacement pair for the user's cryptographic key pair, which represents a complex process, or the request for registration of new users, requiring approval from the administrators.

The application, depicted in Figure 4.2, provides an interface layer that allows the interaction with the user, through a command line console. This layer receives the user's input, validates it and forwards the request to a service layer, in order to perform the required operations. The service layer is responsible for performing the users' operations, guaranteeing the system's functionality and security properties. To ensure this, the service deals with files, metadata, clouds, cryptography and performs complex operations (e.g. cryptographic operations on files before uploading them to the cloud stores or validation of users' certificates before using their public keys). To process the requests, the service relies on a group of individual modules: the session manager, the communication manager, the key manager, and the file manager. The execution of the requests through the different modules is completely transparent to the user.

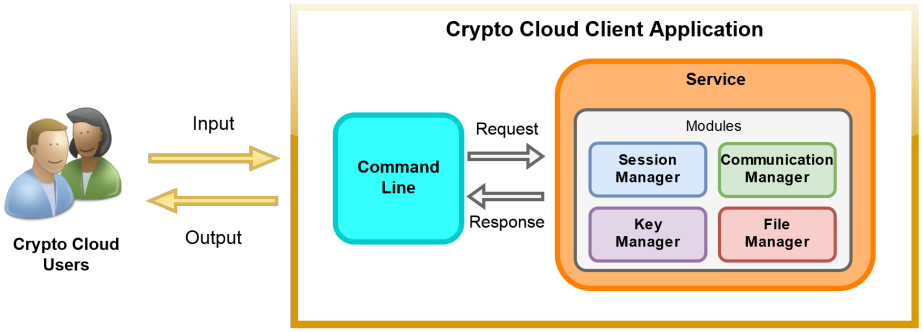

Figure 4.2: Crypto Cloud Client Application's architecture.

The **session manager** has the responsibility of preserving the application's local state after the user successfully performed the login on the system. This includes caching information about the user (e.g. username, cloud accounts, and others) and about the user's managed files (e.g. file name, version, share revision, and others).

The **communication manager** is the module responsible for interacting with the CCDS. During the login process, it requests the CCDS for the user's authentication, receiving an authorization token. Future interactions with the CCDS will use the granted authorization token to access the desired resources.

The **key manager** controls the user's cryptographic key pair. It collaborates with the KMIP Client API in order to access the cryptographic keys and perform the needed cryptographic operations. This API was implement for the Client Application, since there was no open source version of the protocol available. Depending on the key usage, this module can offer two different work modes: the local mode and the remote mode. The first uses the KMIP protocol to retrieve

the user's private key parameters (i.e. modulus and private exponent). The private pair is reconstructed and the cryptographic operations are performed locally on the device. The second uses the KMIP protocol to perform remote cryptographic operations using the user's private key, without the need of locally maintaining the key's sensitive information.

The **file manager** maintains a workspace that contains local copies of user's files. This module is capable of creating, reading and writing files. It also manages a temporary directory used for temporary files (e.g. when downloading a file), and a conflict directory, where the conflict files are moved (e.g. when name conflicts occur or invalid versions are detected).

4.2.2 Crypto Cloud Directory Server

The Crypto Cloud Directory Server (CCDS) serves the client application. This component acts as a metadata repository, responsible for the system's metadata associated to users, files, shares and clouds. Its main function is to maintain the metadata used to manage the files, allowing to keep track of these files in the cloud stores. The CCDS also stores ACL entries along with the files' cryptographic keys, which are protected (i.e. wrapped) using the users' keys. The managed information is persistently stored using a structured database.

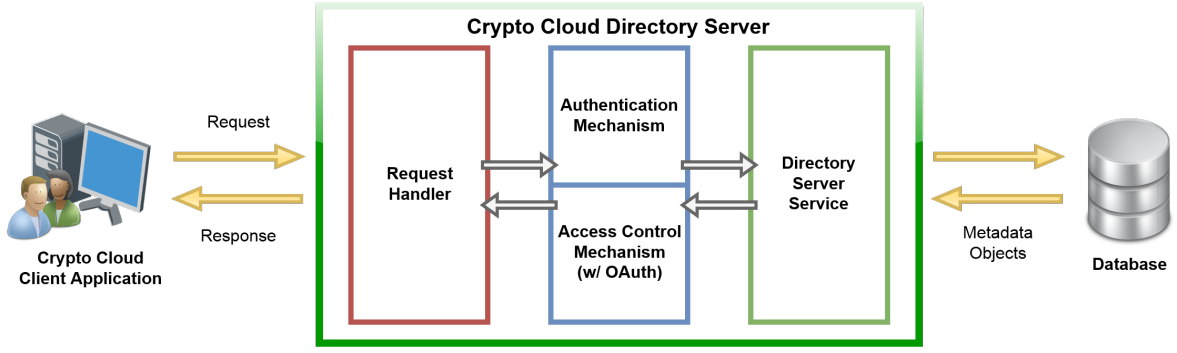

Figure 4.3: Crypto Cloud Directory Server's architecture.

The CCDS require users to authenticate themselves before starting serving their requests. After successful authentication, the CCDS grants an authorization token that allows access to its resources. Also, the access control mechanisms are checked on every request to prevent improper access to the managed information.

4.2.3 Key Management Server

The system includes a Key Management Server, illustrated on Figure 4.4, responsible for accessing and protecting users' private keys. This component frees the users and the CCDS from the responsibility of managing users' cryptographic keys. The server follows the KMIP protocol [18], allowing users to generate, obtain, use and destroy their cryptographic keys. The protocol specifies well-defined policies to access the available resources and perform operations. It requires the authentication of every request handled by the server. By following the KMIP standard, this component can also serve clients that are out of the Crypto Cloud's scope.

Even though the KMIP standard is open source, there was no stable open source implementation of the protocol available. As result of that, we implemented the KMIP protocol for this server component. In order to serve its clients, our implementation for the KMIP protocol uses dedicated key storage hardware, namely a remote HSM provided by Multicert, to generate, access and protect (i.e. wrapping) managed cryptographic keys. The architecture, implementation and integration details of the KMIP protocol on the proposed solution are further described, in Section 5.4.

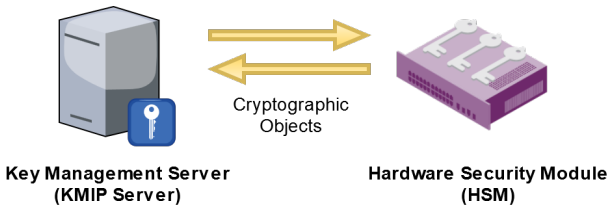

Figure 4.4: Key Management Server overview.

4.2.4 Cloud Stores

The cloud stores represent cloud accounts registered in the system by their owners. These stores act as passive storage space, maintaining the users' files persistently stored in the cloud without needing to run any type of application's code.

To manage hosted files, the cloud providers grant a Access Token (AT) after proper authentication of the account's owner. These tokens are used to access the stores using the respective provider's API. The providers' APIs offer methods that allow users to create, access, modify and delete stored files. Crypto Cloud call these methods in order to manage users' files. These

files are protected and entirely hosted on users' cloud spaces, in a specific application's directory.

4.2.5 Public Key Infrastructure

The Public Key Infrastructure (PKI) component is an infrastructure responsible for digital certification and validation of users' public keys. It acts as a trusted third party entity, increasing the system's trust when using public keys. This component is capable of emitting X.509 certificates, binding the contained public key to its subject's identity. It also maintains a repository of revoked certificates, called Certificate Revocation List (CRL). This mechanism allows clients to check if the certificate was revoked before using it (e.g. check certificate before sharing files).

The digital certificates are signed by the infrastructure during the user's PU certification process. Therefore, clients should have their means to validate a certain certification chain in order to consider the whole infrastructure and the emitted certificates trustable.

4.3 Models

This section describes the models considered during the designing of our solution. We start by defining the solution's trust model (Section 4.3.1), where some assumptions are considered. Then, we identify possible threats and vulnerabilities of our solution by following a STRIDE [17] model (Section 4.3.2). To conclude, we describe our access control model and its policies (Section 4.3.3).

4.3.1 Trust Model

Each component described in the system's architecture runs in distinct environments: the client application executes in the user's device, the CCDS runs in the host's organization, the KMIP Server and the PKI are required to be deployed in special environments that present higher security levels and the clouds run over insecure environments. In order to better understand these different environments and how these components should work, it is necessary to establish a satisfactory trust model. The trust model helps us recognize special characteristics of the system and how the various entities are expected to behave.

The following details the assumptions taken into account to define the trust model of the proposed solution:

- **The client's device is trustworthy:** the software and hardware where the client application runs is reliable and not compromised;
- **The CCDS is considered honest but curious, not acting malicious:** the CCDS can listen to the exchanged messages but follows the system's protocol and does not launch active attacks;
- **The Cloud Providers can act maliciously:** the cloud environment presents some security flaws (i.e. it can have malicious insiders), so it is considered that the providers can act maliciously and try to read or change the content of hosted files;
- **The Key Management Server is trustworthy and deployed over a secure environment:** the KMIP server is deployed in a secure infrastructure, the access (i.e. physical and logical) to its resources is controlled and monitored and it only answers to KMIP requests from authenticated users;
- **The private keys are securely stored and cannot be manipulated by agents external to the client application:** these keys can only be accessed by their owners and they cannot be used outside of the Crypto Cloud's client application;
- **The PKI infrastructure is considered to be trusted:** the PKI act as root of the system's trust and its certificates are authentic, meaning that if the certificate is trustworthy, then the subject is the expected subject;
- **The communication channels are insecure:** the exchanged messages can be intercepted and eavesdropped by third-party entities;
- **The cryptographic algorithms used are sound:** the used cryptographic algorithms are considered secure and not broken.

4.3.2 Threat Model

A threat model identifies potential threats and vulnerabilities of the system. Its analysis helps design better defences against attackers and possible countermeasures, increasing the secu-

rity of the system. The STRIDE threat model [17], introduced by Microsoft, organizes possible threats according to the attacker's motivation. This model introduces the following threats' categories: spoofing identity, tampering with data, repudiation, information disclosure, denial of service and elevation of privileges.

By applying the STRIDE threat model to the proposed solution, the following potential threats were identified:

- **Spoofing Identity:**
  1. An attacker can brute-force token values in order to obtain access to the system;
  2. An attacker may try to replace a user's certificate with his own certificate;
  3. A malicious user or application may provide fake information in order to persuade users to consider him as a trustworthy certification entity.
- **Tampering with Data:**
  1. A malicious cloud provider may modify users' files.
- **Repudiation:**
  1. A user can claim that did not modified a file from the system.
- **Information Disclosure:**
  1. A malicious cloud provider may read users' files;
  2. The client's device can be targeted by a malicious application (e.g. malware) in order to access application's sensitive information (e.g. user's keys);
  3. An external agent may eavesdrop exchanged messages between components.
- **Denial of Service:**
  1. A malicious user can re-upload old versions of files in order to destroy recent updates;
  2. A malicious user can spam repeated requests in order to degrade the service.
- **Elevation of Privilege:**
  1. A malicious user can perform unauthorized operations (e.g. delete) on resources (e.g. files, keys or others) that are owned by other user.

4.3.3 Access Control Model

Current cloud providers define specific access control models for their services. This includes the models for internal and external users. Internal users represent users that have their accounts registered in the service. External users do not have their accounts registered in the service. Normally, this type of users access the clouds' resources (i.e. files) through a specific URL. Although providers grant access to these different users, they specify different access restrictions. For instance, Google Drive provides fine-grained access control over individual files or folders. It allows owners to share individual files with other users, granting them read-write permissions. Drive provides mechanisms for owners to individually manage the internal members of a sharing group, yet, external members cannot be individually controlled considering that the resources' URLs are publicly available on the web. Dropbox, on the other hand, allows access control over individual files and folders but follows different semantics. Owners can grant read permissions to other users (e.g. internal and external users) over individual files and folders, however, Dropbox only allows granting write permissions to internal users over individual folders. Write operations from external users and over individual files is not allowed. Dropbox also has the problem of Google Drive, when managing individual access over the resources' URLs. Given these different solutions, it is crucial to define a uniform access control model.

In Crypto Cloud we follow the Access Control List (ACL) approach. This model defines the access control between subjects (i.e. users) and objects (i.e. files), where a list of permissions is attached to an object, representing the permissions granted to each subject. It is considered that a subject has permission to a file if and only if an ACL entry exists. Each object has one owner (i.e. file creator) that represents a special subject who retains full-access permissions which cannot be revoked. This subject can define new ACL entries, granting permission to other subjects. Our model defines read (R), write (W) and share (S) as the possible permissions that a subject can have. It is also considered that the permissions are cumulative, meaning that subjects with read permission can only read files, however, subjects with write permissions can write (i.e. modify or delete) and also read files. S permission allows a subject to read, write and share a file. Table 4.1 represents the Crypto Cloud's access control matrix and relates the file operations authorized by Crypto Cloud (on each row) with each one of the different possible permissions (columns 2 to 4). The read operation is allowed by all three permissions (R, W and S). The update and delete operations are allowed by both W and S permissions. The chperm

operation (i.e. share and revoke functions) is only authorized to users who old S permission. When a user performs a create operation (i.e. uploads a new file for the first time), the system assigns him as the file's owner, granting full-access control.

Operation	R	W	S
Read	✓	✓	✓
Update		✓	✓
Delete		✓	✓
Chperm			✓

Table 4.1: Crypto Cloud's permissions matrix.

In our solution, the ACLs structures are maintained by the CCDS, which is the component responsible for enforcing the access control model. This process is done by checking the user's identity against the requested object's ACL on each operation.

4.4 User Authentication

User authentication represents a crucial part of the access control mechanism, being its first line of defence against intruders [28]. The authentication process is done through the verification of some type of authentication factors (i.e. secret credentials), which includes: knowledge factors, possession factors and inheritance factors. Knowledge factors include something that only the user must know in order to proof his identity (e.g. username/password pair, PIN code, answers to secret questions). Possession factors consist in anything that a user must possess in order to authenticate himself (e.g. token, smartphone, smartcard, physical key). Inheritance factors include any inheritance traits that the user have or produce (e.g. fingerprint, retina, voice pattern, handwriting characteristics). After this verification, we can confirm the user's authenticity.

The next sections describe the authentication mechanisms that each one of the system's components follow to authenticate their users.

4.4.1 Crypto Cloud Authentication

In Crypto Cloud, clients authenticate themselves towards the CCDS using knowledge factors. The CCDS uses a password-based authentication mechanism, where a client has to provide a

valid username and secret password pair in order to prove his identity. After this confirmation, the CCDS delivers a unique time-limited Access Token (AT) to the client. After receiving the AT, the client is allowed to use it by following the OAuth Authorization Protocol, described in Section 2.1.4, in order to access the resources from the CCDS. When the AT's lifetime ends, the client has to repeat the authentication process in order to get a new AT. By following this approach, users can only access their own files, as the access to the Crypto Cloud's service is only granted after presenting a valid AT. This protocol is widely used in web applications and allows the use external authentication services (e.g. Facebook Login or Google Authentication Services), relieving the CCDS from the responsibility of maintaining the users' accounts information.

4.4.2 Cloud Store Authentication

Our solution follows a similar approach as Storekeeper [20, 21] for the user authentication in the Cloud Stores. The access to the cloud stores is done through their respective APIs without compromising the users' credentials. Therefore, Crypto Cloud Client Application has to authenticate itself towards the cloud provider. This process follows a similar approach to the one previously described on Section 4.4.1. When a user wants to register a new cloud store in our application, Crypto Cloud contacts the respective API and opens a new webpage from the cloud provider. The user authenticates himself with his credentials and an authentication code is generated. Our client application receives this code and uses it for requesting access through the API. The cloud provider retrieves the authentication code from the request, verifies it and generates a unique Access Token (AT). This AT is maintained at the CCDS for future use when interacting with the user's cloud storage space.

Since the usage of an AT grants full-access to part of the user's cloud service, it needs to be securely maintained in order to protect the cloud store from unrestricted access. Before storing the token on the CCDS, the client application uses a User Key (UK), to wrap and read-protect the AT from the CCDS and remaining users, as shown in Equation 4.1:

$$AT_{CloudStore} \Rightarrow \{AT_{CloudStore}\}_{UK} \tag{4.1}$$

4.4.3 Key Management Authentication

The Key Manager Server is a crucial component of our solution, therefore it requires strong authentication mechanisms. This component implements the KMIP protocol, which requires user authentication on every request. The KMIP protocol [18] allows the use of three different types of credentials: username and password credentials, device credentials, or attestation credentials. Our solution considers the use of username and password credentials when performing KMIP requests through the KMIP Client API, allowing different users to use the same device to interact with Crypto Cloud.

In order to prevent disclosure of the user's credentials or private keys, the KMIP credentials are only handled by the KMIP protocol itself, meaning that these credentials are neither persistently stored at the CCDS nor at the Crypto Cloud Client Application.

4.5 Functional Algorithm

One of the most important aspects of Crypto Cloud is the protection of managed files, which guarantees the confidentiality and integrity of users' sensitive files on public clouds. To better understand this process, we start this section by describing the algorithms responsible for supporting that functionality. Then, we describe some emerging challenges and detail our approach to handle them.

Before introduce our algorithms, it is important to remember that registered Crypto Cloud's users hold three different keys: an asymmetric key pair, which includes a Private Key (PK) and its respective Public Key (PU), and a symmetric cryptographic key, called the User Key (UK). The first is stored at the Key Management Server and can be only handled by its owner (i.e. user). The second is maintained at the CCDS and can be used by public users in order to protect (i.e. wrap) other keys. The last is manipulated by its owner to protect sensitive data (e.g. cloud stores' access tokens).

4.5.1 Base Algorithm

The functional process of our proposed solution involves management and protection of users' files. Consider only read and write operations, and not the sharing operations. To

better understand the following use case, assume that Alice is an authorized user of the system and wants to create a new file through Crypto Cloud. Our client application has to provide mechanisms to ensure the data confidentiality before uploading it to the cloud. The first step that our application takes is to generate a new symmetric cryptographic key, called File Key (FK). Then, it performs the encryption of Alice's file using the previous generated FK. After this process, the encrypted file is uploaded to Alice's cloud store and its metadata registered on the CCDS. In order to prevent illicit access, the FK is wrapped using Alice's PU and the wrapping result stored at the CCDS. After this process, if Alice wants to read the file, all she needs to do is to: (1) download the encrypted file, (2) fetch its wrapped FK from the CCDS, (3) obtain FK by unwrapping it using her PK and (4) decrypt the content of the encrypted file using the obtained FK.

However, this approach does not guarantees the file's integrity. At any time an attacker or a malicious cloud provider can change the uploaded file's content without Alice noticing it. To overcome this problem, Crypto Cloud relies on a Message Authentication Code (MAC) function that is executed mutually with the previously explained mechanism, as illustrated in Figure 4.5. When Alice performs a create operation, the client application starts by generating another symmetric cryptographic key, called Integrity Key (IK). After that, the algorithm uses the generated key (IK) to perform a Hash-based Message Authentication Code (HMAC) calculation over the content of the plaintext file, producing a fixed-size hash result. The produced result is stored (i.e. concatenated) in plaintext alongside with the ciphered content, taking the firsts bytes of the encrypted file. Analogously to the previous mechanism, IK needs to be wrapped using Alice's PU in order to prevent illegitimate use of this key. The encrypted file is then uploaded to the cloud and the wrapped IK stored at the CCDS alongside with the remaining file's metadata.

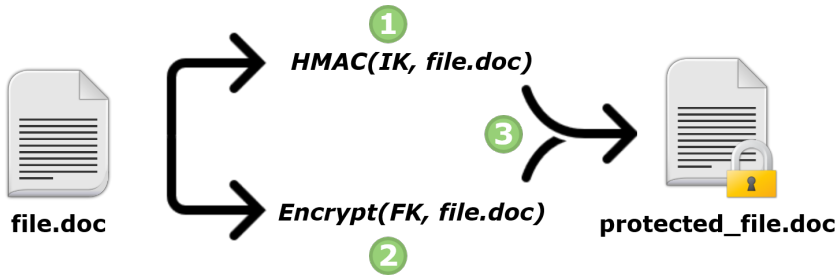

Figure 4.5: Example scenario to illustrate the base algorithm. First, (1) the HMAC of file.doc is calculated using the IK. Then, (2) the content of file.doc is encrypted using the FK. Lastly, (3) the result from the HMAC operation is merged with the result from the encrypt operation.

When Alice tries to read a file, the application performs the reverse operation. It starts by (1) downloading the ciphered file and (2) retrieving the HMAC result from it. Then, (3) the client obtains both FK and IK and unwraps them using Alice's PK. Following that, (4) the algorithm triggers the confidentiality mechanisms, decrypting the content of the downloaded file. After decryption, (5) the integrity mechanism performs a HMAC calculation over the decrypted content using the obtained IK. Then, the result of this operation is (6) compared with the retrieved result from the step (2). If both results match, it is considered that the file was untouched, otherwise, it means that the content of the file was tampered and compromised.

Considering this combined approach, our solution can guarantee the confidentiality and integrity of users' files, preventing the CCDS, the cloud providers, and other users or external agents from reading or compromising Alice's private files. If Alice wants to update the file, the application repeats the previous algorithm over the new version, updating the file's metadata stored at the CCDS. The application also allows Alice to delete her files by deleting the encrypted file from her respective cloud store and updating the CCDS with that request.

4.5.2 Version Control

The previous algorithm does not provide any type of version control over the files' content. Thus, update operations are interpreted as simple overwrite operations over the same file. Crypto Cloud addresses this issue by implementing a version control mechanism at the CCDS. This mechanism relies on two metadata elements in order to operate: the file content hash and the file version. The file content hash represents the hash result of the content of the file before encryption (i.e. in plaintext). This result is calculated by the client application and sent to the CCDS. The file version is a number managed by the CCDS, which is incremented every time a user performs an update operation over the respective file. The association of both these elements ensure that a certain file's content represents a specific file version.

Considering the relation established by this mechanism, it is possible to identify and manage the version of Crypto Cloud's files. However, our solution does not implement any version fallback mechanism as the file versioning (i.e. maintaining copies of old versions) was not considered for this work.

4.5.3 File Sharing

Although the introduced base algorithm ensures the confidentiality and integrity of stored files, it lacks in providing sharing functionalities, since the sharing operations were not contemplated in the use case. Assume for now a use case where Bob is also an authorized user of the system and Alice is the owner of a file, named *file.doc*, that is also shared with other users besides Bob. Suppose that Alice wants to share the file with Bob.

A possible approach for this sharing mechanism can be Alice securely sharing the file's FK and IK with Bob. This could be done by wrapping both FK and IK keys using Bob's PU and registering the respective ACL entry on the CCDS. By doing this, Bob could unwrap these keys using his PK and access the file's content. Now, suppose that Alice wants to revoke Bob access to *file.doc*. All Alice has to do is to re-encrypt the file using new FK and IK keys and re-upload it to the cloud. To allow other users besides Bob to access the file, Alice has also to wrap the new FK and IK keys using their public keys and update their ACL entries on the CCDS. After that, Bob will not be able to see future updates of this file. However, this process has a huge performance overhead associated as result of the re-encryption and re-upload processes.

To overcome this issue and implement efficient access revocation, Crypto Cloud follows the Storekeeper's approach [21], avoiding re-encryption of the entire file and redistribution of both new keys. In order to conceive a better sharing algorithm, this approach employs three new techniques:

1. **Readers have access to a Read Key (RK):** instead of granting direct access to the FK, an intermediary symmetric cryptographic key is used, named the Read Key (RK), that is shared between all group members of that file. This RK is used to wrap the FK, which encrypts the file's content. After that, the RK is wrapped using each one of other users' public keys;

2. **Revocations produce a new Read Key (RK):** every time a revocation occurs, a new RK has to be generated. This process includes the wrapping of the existing FK and IK keys using the new RK and the distribution of the newly generated RK amongst the new set of members of the sharing group. The file's ACL at the CCDS has also to be updated with this new metadata;

3. **Updates generate a new File Key (FK):** every time a writer issues an update to a file, the respective file's FK is replaced and the new FK is used to encrypt the new file's version. This new FK is wrapped using the current RK in order to allow the members of the sharing group to access the new FK and continue reading future updates. Additionally, Crypto Cloud's algorithm also renews the file's IK in order to decrease its exposure.

By following this algorithm, it is possible to disrupt the access to the file's content from old group members. Also, this approach presents lower overhead than the previous approach, since it introduces a new intermediary key, the Read Key (RK), preventing the re-encryption of the file's content when performing the revocation, and also refreshes the FK when performing an update, inhibiting revoked users to read new updates. However, this approach does not provide mechanisms to notify the sharing group members that Bob's access was revoked. A writer that did not noticed that Bob's access was revoked, could issue a file update and share it with Bob, using the old RK. This may result in disclosure of sensitive information.

Crypto Cloud deals with this problem by creating a pseudo-random number, named Share Revision (SR) that acts as a control variable. This number is maintained by the CCDS and is regenerated every time a revoke operation is issued. The CCDS also ensures that a newly generated SR is different from its previous one. When a user performs an update or revoke operation it has to send the respective file's SR. If the sent SR and the SR maintained by the CCDS match, the operation is carried on, otherwise the user must update his local metadata before proceeding. This method guarantees that the user who performs any of these operations has knowledge of the current set of users in the sharing group and possesses the most recent file's RK.

4.5.4 File Homing

The algorithms introduced on the previous sections describe the Crypto Cloud's approach to manage and share files without disregarding their security aspects. However, these algorithms do not describe methods to handle files on cloud stores. To address this, Crypto Cloud follows Storekeeper's Staging Space approach, which maintains staged files and respects a remote-read local-write policy [21]. This policy states that authorized users can read files from other users' cloud stores but their writes can only be performed on their own cloud stores, maintaining the

isolation between users' cloud accounts. Although this mechanism achieves its goal, maintaining staged files comes with some secondary effects, such as: lost updates when cloud stores are removed or free riding on user's clouds after a revocation occurred. For instance, suppose a scenario where Bob, with W permissions, issued an update operation over Alice's shared file, named *file.doc*. The following describes the inherent problems in this situation:

- **Lost Updates:** if Bob decides to remove his cloud store or manually delete the temporary file (i.e. staged file) from his cloud space, the update performed by Bob on *file.doc* becomes unavailable. The severity of this problem rises as the period since Alice's last update grows, considering that a higher number of cumulative updates will be lost;
- **Free Riding:** the free riding phenomenon occurs when a user is, intentionally or not, occupying some else's cloud space. Imagine that Alice revokes Bob's access to the *file.doc*. After this, Alice will be free riding Bob's cloud space, since the last version of the *file.doc* file is still stored on Bob's cloud store. The file is occupying Bob's precious cloud space and Bob does not takes any benefit from it, since his permissions were revoked.

Crypto Cloud counters these problems by employing a File Homing technique. This technique, similarly to Storekeeper [21], consists in periodically reallocate staged files back to its owner's cloud stores. The file homing process can only be performed by files' owners. Consider the previous scenario where Bob updated a file owned by Alice. When Alice triggers the file homing process, her client application starts iterating through his files, checking the location of each file's latest update. The application identifies that the last version of *file.doc* is stored outside of Alice's cloud space and the file reallocation process begins. This reallocation process consists of three tasks: (1) download the encrypted file (i.e. *encrypted file.doc*) from Bob's cloud store, (2) upload the download file to one of Alice's cloud store, (3) update the file's metadata (i.e. the *file.doc*'s new location) on the CCDS. However, after this process, Bob does not know that his version has been reallocated and can be deleted from his cloud store.

To address this issue, the CCDS stores the old location of the file in a special data structure, named Ghost File Structure, upon receiving a file reallocation request. The Ghost File Structure represents a stale file that is no more managed by Crypto Cloud but still occupies someone else's cloud space. The Crypto Cloud's users are late notified about Ghost Files hosted on their cloud stores. In the previous example, when performing the request from the step (3), the CCDS saves

a Ghost File structure with the previous file's location. When Bob interacts with the CCDS, he is notified about his ghost files. After that, the garbage collector from Bob's client application deletes these stale files from his cloud stores, claiming back his precious storage space.

4.6 Summary

In this chapter we introduced the Crypto Cloud solution. This solution consists of an augmented version of Storekeeper [21], a system capable of providing cloud's features without trusting on its providers.

Crypto Cloud is composed of five components: the client application, which runs on the user's device and provides the main functionality; the Crypto Cloud Directory Server (CCDS), responsible for managing the files' metadata; the Key Management Server (KMIP Server), which manages and accesses users' cryptographic keys; and Cloud Stores, that provide users' storage space and store their files. Additionally, this solution takes advantage of a PKI, a third party entity, which is responsible for the certification and validation of users' public keys. The exchanged messages between all components is protected by establishing secure communication channels through the TLS protocol.

To properly project our system, some security models were considered, including a Trust Model, a Threat Model and an Access Control Model. The first model stipulates the assumptions taken in consideration by our solution. The second follows the STRIDE model, which identifies potential threats and vulnerabilities of our system and helps design possible countermeasures to them. The third describes specific restrictions and policies that users have to respect when accessing resources provided by our solution.

The presented solution considers different authentication mechanisms for each one of its components. Users authenticate themselves towards the CCDS by providing a valid username/password pair. After authentication, users can access CCDS's resources using a valid authorization token. The Cloud Stores follow a similar mechanism. The providers grant access to their users' cloud spaces by providing an Access Token (AT). By following this approach, users' cloud credentials are not compromised. The Key Management Server (KMIP Server) implements the KMIP protocol, employing strong authentication mechanisms without maintaining any session state. This means that users have to authenticate themselves on each request by

using their username and password credentials.

In order to achieve the desired functionality, multiple algorithms were introduced. The base algorithm uses different cryptographic keys to guarantee the confidentiality and integrity of the uploaded files. The version control is performed by associating the result from the hash of the file's content with a version number. The sharing algorithm introduces an intermediary cryptographic key that is distributed through the sharing group by using their public keys. The staging algorithm respects a remote-read local-write policy, where readers can access shared files through a URL and writers upload their file's updates to their own cloud stores. In order to prevent lost updates and the free riding phenomenon, users periodically trigger file homing mechanisms on their files. This mechanism reallocates the last update of a shared file in the owner's cloud store and can only be triggered by the file's owner.

5 Implementation

This chapter describes the implementation of the proposed Crypto Cloud system. In Section 5.1, we present an overview of the solution's implementation, describing the technologies chosen for each component. Section 5.2 describes the data structures responsible for maintaining the application's state during its execution. Section 5.3 details the implementation of the protocols responsible for the most relevant Crypto Cloud's operations. Section 5.4 presents our implementation for the new KMIP standard, which is an important component of our solution. Section 5.5 concludes this chapter, summarizing all aspects of the implementation.

5.1 Overview

The prototype of the Crypto Cloud system was fully implemented in Java 8, which is a popular programming language in enterprise computing and continues to evolve. The Crypto Cloud prototype takes advantage of Spring Framework v4 [26], which is an open source application framework for the Java platform. The Spring Framework is widely used in the development of web applications, including in most of the Multicert's products. This framework allows the use of configurable Java Beans to wire various components together, structuring the application into multiple layers. These layers are defined using interfaces, increasing the flexibility of the implementation. Overall, the Crypto Cloud's prototype comprises 5000 lines of code, excluding the implementation of the KMIP protocol.

The CCDS runs on a Spring Boot application, an module of Spring Framework, which provides an embedded Tomcat servlet. This built-in servlet container is widely used in Multicert's products and allows the creation of a Representational State Transfer (REST) web service with little effort. The REST endpoint represents the entry point of users' requests. The CCDS allows the aggregation of multiple operations into a single request. This queueing technique decreases the number of interactions performed with the CCDS during the execution of Crypto Cloud's

protocol. The authorization to the CCDS's web resources is done through the OAuth protocol, a common authorization standard for web applications. Our implementation takes advantage of the Spring Security module to implement the OAuth protocol and configure the authorization rules (e.g. allowed endpoints, users' database and others). The developed prototype couples the Authorization Server and the Resource Server inside the CCDS's machine. The users' passwords are encoded using the BCrypt [30], which is a robust encoding algorithm commonly used and recommended by NIST standards. After successful authentication, our OAuth's implementation provides 128-bit random access tokens with one hour limited-lifetime in order to reduce the token's exposure and limit the windows of occurrence of possible collisions. Regarding the database, the CCDS relies on PostgreSQL [31] Database Management System (DBMS), provided by Multicert, to securely store and manage the data. The access to the CCDS's database is done through the Spring Data JPA framework, which provides a data access interface, reducing the effort needed to access the database's Structured Query Language (SQL) tables.

The Client Application interacts with the CCDS through its REST endpoint. The requests are encoded in JavaScript Object Notation (JSON) messages, a lightweight text-based open standard designed for exchange of data structures between applications, and sent through secure TLS channels. Our implementation for the Client Application currently supports the integration with most popular public cloud providers, such as Dropbox (API v2) and Google Drive (API v3). Our implementation also supports the use of proxy servers inside the network, resulting in benefits for their clients (e.g. protecting clients from malicious sites). Regarding cryptography, our prototype relies on Java Cryptography Architecture (JCA) APIs to perform cryptographic operations. When dealing with symmetric cryptography, we use 256-bit AES keys, with a pseudo-random Initialization Vector (IV) and the cipher is performed in CBC mode with PKCS5 padding. For asymmetric cryptography, we use 2048-bit RSA keys with PKCS1 padding. To perform the hashing of the content of the managed files, our prototype uses the SHA512 hashing algorithm. When dealing with MAC operations, our implementation uses the HMAC SHA 256 algorithm with 256-bit AES keys. Currently, given the context of the proposed solution, the chosen cryptographic algorithms, as well as the key sizes, are considered secure and not broken. Lastly, in order to decrease the exposure of cryptographic keys, we perform the renewal of the cryptographic keys and IVs in use on each execution of Crypto Cloud's functional algorithms (e.g. during the update of an existing file).

Crypto Cloud also implements logging mechanisms, both in the Client Application and in the CCDS, registering important information for future analysis. These mechanisms take advantage of the Log4J v2 framework [1], which is one of the most commonly used frameworks to implement logging on Java applications. The Client Application logs the user's flow of every interaction to a local file on the device, without disclosing user's sensitive information (e.g. key's information). By doing this, administrators can help diagnose and solve potential failures, once the users provide access to their logs. The logging mechanism from the CCDS records the workflow of each users' requests. This mechanism helps troubleshoot and investigate possible attacks or malicious operations that the CCDS may suffer.

5.2 Data Structures

In order to provide an adequate service and keep track of the managed files, Crypto Cloud system has to maintain an internal state. Since our solution aggregates different Cloud Stores, they are not capable of maintaining state of the files of the entire system. Thus, our solution only considers Cloud Stores as persistent storage space. The Client Application represents the core of the solution, however it only maintains an ephemeral internal state, since its state only exists during the session of the user. This component depends on the user's authentication process to obtain an accurate state from the CCDS. The CCDS is the component responsible for maintaining the state of the whole system. This state includes important information about users, clouds, files and sharing groups. In order to efficiently access and manage this information, the CCDS relies on a relational database provided by a DBMS. Figure 5.1 illustrates the schema of this database. Next, the tables, the constraints and the existent relations are described:

The **CC_Users** table stores all the users' information of the system. An entry in this table represents a Crypto Cloud's user, who is characterized by a unique username. Instead of storing the user's password in plaintext, the system only maintains a hash result of the user's password, protecting it from possible leaks. Alongside with this information, the system also maintains a boolean variable that represents the status of the account (e.g. if the account is enable or disabled). The user's cryptographic information is also stored in this table. This information includes the identifier of his PK, granted by the KMIP Server, the user's public certificate, his wrapped UK, and his User IV.

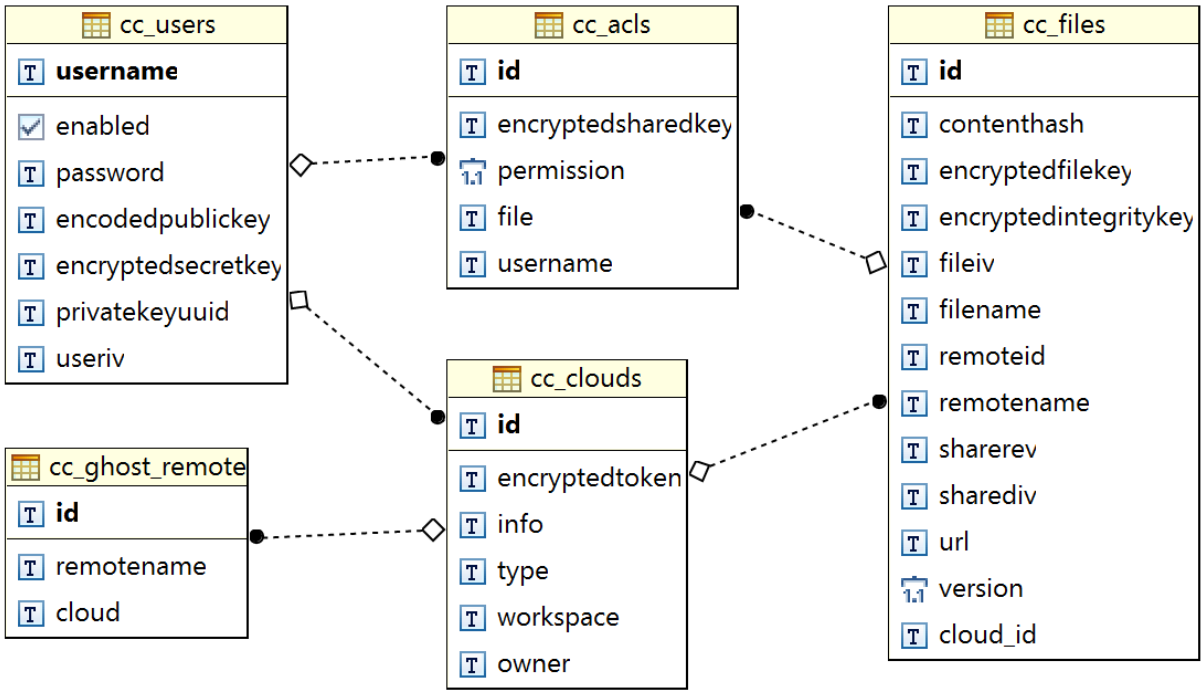

Figure 5.1: CCDS's database schema.

The **CC_Clouds** table maintains information about the users' Cloud Stores. A cloud entry contains information about its owner (i.e. user), cloud supplier (e.g. Dropbox or Google Drive), workspace path for this cloud account, encrypted token that is used to access the cloud resources through provider's API and info field, which contains the email associated to that cloud account. In order to prevent the registration of the same cloud account by different Crypto Cloud's users, this table defines a unique constraint between the type and the info fields.

The **CC_Files** table is responsible for maintaining the metadata of Crypto Cloud's managed files. The stored metadata includes the original filename, the hash result of the file's content, its current version and the share revision numbers. In order to track these files on cloud space, the table maintains the file's URL, the file's remote id and the file's remote name. These three parameters, which are acquired through the providers' APIs, allow the Client Application to read, update and delete files from Cloud Stores. The table also stores cryptographic information from files, which includes the wrapped FK, the wrapped IK and the IVs used.

The **CC_ACLs** table maintains the ACLs of the system's files. Each entry of this table represents an ACL entry for a user in a file. Aiming to implement a strong access control mechanism, this table defines a unique constraint between these two fields, meaning that a user can only have one ACL entry per file. For each ACL entry, the system also stores the permission

granted and the wrapped RK, which was wrapped using the user's PU during sharing algorithm execution.

The **CC_Ghost_Remote** table stores information about files that were deleted from the system but still occupy storage space in certain Cloud Stores. This table maintains information about the file's remotename, which identifies a file inside the cloud space and the cloud store where the file is stored. Later, during the garbage collection procedure, this information will be deleted.

During the design of the database schema, some relations were established in order to have an efficient management of the stored information. The association between *CC_Users* and *CC_Clouds*, *CC_Clouds* and *CC_Files*, and between *CC_Clouds* and *CC_Ghost_Remote* can be described through a simple one-to-many relationship: one Crypto Cloud user can have none to multiple Cloud Stores registered; one Cloud Store can host zero to several Files; and one Cloud Store can have zero to several Ghost Files to remove. However, *CC_Users* and *CC_Files* describe a more complex relationship that is established through an intermediary table, called the *CC_ACLs* table. This join table allows a many-to-many relationship between *CC_Users* and *CC_Files* to work, meaning that one user can have one to several ACL entries to different files and one file can have various ACL entries from different users. The established relations and their constraints are enforced by the system's DBMS and their violation forces the requested operation to fail.

5.3 Protocol

This section describes the implementation details of the protocols responsible for performing the Crypto Cloud's operations. The procedures taken during the execution of these protocols are described in more detail in Appendix A. During the execution of these protocols, the Client Application and the CCDS exchange JSON messages through an established secure communication channel. The Client Application sends the user's authorization token alongside with the operation request. Upon receiving a request, the CCDS verifies if the provided authorization token is valid using an in-memory table that stores the granted authorization tokens. After successfully authorization, the access control mechanism verifies if the user has access to perform the operation on the desired file, checking the file's ACL, and the execution proceeds. When

the operation ends, the CCDS answers to the Client Application, sending a JSON message with the operation result.

The **read operation** consists of updating a local file with the most recent version available from the cloud stores. During this operation, the Client Application obtains the file's metadata from the CCDS using the file's identifier. Upon receiving the file's metadata, the Client Application checks if the file exists in the local workspace or is obsolete. In this case, the application accesses the file's URL to download its content using Apache Commons IO Library v1.3, which is an API that provides utilities to manipulate files over Java applications. After receiving the file, the client application retrieves its HMAC value, unwraps the file's keys using the user's PK, deciphers its content using the JCA library and checks its integrity, comparing the obtained HMAC values. After that, the file's metadata is maintained inside a Java List structure, during the user's session, for future use.

The **write protocol** consist of uploading a local copy of a file to a registered cloud store. To create a new file in the Crypto Cloud system, the Client Application relies on the JCA library to generate the file's keys and IVs and perform the cryptographic operations, guaranteeing the confidentiality and integrity of the file. After that, the protected file is uploaded to the cloud store using the Cloud Provider's API. After uploading, the Client Application sends a request to the CCDS with the new file's metadata. The CCDS verifies if the file is unique and does not conflict with other files in the user's workspace. After registering the new file, the CCDS answers to the client with the new file's id, version number and SR. The Client Application stores the received metadata in the user's session. To perform a file update, the Client Application requests the CCDS for the file's metadata using the file's id and checks if the user maintains the most recent updated version of the file. Then, the Client Application renews the file's keys and IVs and follows similar approach to the create operation.

The **delete operation** removes a file and its corresponding metadata from the system. When performing this operation, the Client Application deletes the file from the user's local workspace and uses the cloud provider's API to delete the remote version of the file. After that, the Client Application contacts the CCDS to remove the remaining file's metadata.

The **share operation** allows a user to share a certain file with other user, called *grantee user*. When performing this operation, the user obtains the *grantee user*'s public certificate from the CCDS. After that, the Client Application verifies if the certificate is valid and was

issued by a trustworthy PKI, checking its signature and the expiration and usage fields. This verification is performed using the JCA library and consulting the cryptographic keys and the certificates stored at the application's truststore. Additionally, the Client Application obtains a CRL from the PKI and verifies if the *grantee*'s certificate was not revoked. After that, the application wraps the file's RK using the *grantee*'s PU and requests the CCDS to update the file's ACL.

The **revoke operation** allows a user to revoke other user from accessing a file. To perform this operation, the Client Application check if the file is updated and generates a new file's RK using the JCA library. After this, the application obtains the digital certificates of the members of the file's sharing group (excluding the revoked user), and follows a similar approach to the share operation. To conclude, the file's ACL stored at the CCDS is updated with the new group's information and a new SR number is generated by the CCDS.

The **sync operation** is a complex operation, that maintains the consistency of the Client Application's internal state and files. This operations starts obtaining the user's information, including cloud stores and files, from the CCDS. After that, the application loads the user's cloud stores and removes existing ghost files using the provider's API. Next, the Client Application scans the user's local workspace for outdated files and follows the read protocol. After updated the local workspace, the application proceeds with the file homing procedure, reallocating owned files to a cloud store owned by the user. After finishing this process, the user's session in the application's internal state is updated and the changes propagated to the CCDS.

The **regen operation** is a special operation that allows users to generate a new pair of asymmetric keys for the Crypto Cloud's system. Users can perform this operation to prevent the use of expired certificates or compromised keys. This operation starts by contacting the CCDS, obtaining the metadata of all user's files. After that, the application unwraps each file's RK using the current user's PK. Then, the Client Application uses the KMIP Client API to interact with the remote Key Management Server through KMIP messages and requests a generation of a new asymmetric key pair for the user. The Key Management Server receives the KMIP request, checks the access control mechanisms and proceeds with operation, requesting the HSM to perform a generation of a new key pair. When the Client Application receives the new key pair information, it requests the PKI to revoke the current user's certificate and generate a digital certificate for the new user's PU. Upon receiving the new certificate, the

application uses the new user's PU to wrap the user's UK and the previously obtained RKs. After that, the application sends a request to the CCDS, updating the user's PK identifier, the user's certificate and the wrapped keys. To conclude the operation, the application clears the application's internal state and requires the user to re-authenticate himself towards the Client Application.

5.4 KMIP

The KMIP protocol is a well-defined standard for the remote use and management of cryptographic keys and widely used by companies, such as: Cisco, Oracle, SafeNet and others. However, the implementations of these companies are considered proprietary software and there is no stable open source implementation of the protocol available for use, since this standard is still recent. This section presents the implementation for the last published version of the Key Management Interoperability Protocol (KMIP) (version 1.3) [18]. Identically to the Crypto Cloud's implementation, our implementation for the KMIP protocol benefits from tools widely used in the development of Multicert's products, such as the Java 8 programming language and the Spring Framework v4 [26]. Overall, the implementation for the KMIP protocol comprises 45000 lines of code, revealing higher complexity and programming effort than the proposed implementation for the Crypto Cloud system.

The KMIP Server consists of a remote server responsible for managing cryptographic keys, which must be deployed in a secure environment. This component runs on a Spring Boot application, which provides a REST servlet, and takes advantage of a HSM hardware to manage cryptographic objects and perform the cryptographic operations. Regarding the server's database, the server relies on PostgreSQL [31] DBMS, provided by Multicert, and the access to the database is done through the Hibernate framework [13], facilitating the mapping between the protocol's objects and the entities stored. Similarly to the Crypto Cloud's implementation, the users' passwords are encoded using BCrypt algorithm [30] before being stored in the system's database.

The KMIP Client consists of a simple component that provides access to the remote cryptographic resources from the KMIP server. This functionality is provided through an API interface, that provides methods to build KMIP requests, interact with the server and decode

the responses. Regarding logging mechanisms, our implementation relies on the Log4J engine [1] to log all the operations performed through the protocol. This helps troubleshooting possible problems and facilitates audits to the system.

In Section 5.4.1, we present the architecture of our implementation for the KMIP protocol and describe its components. In Section 5.4.2, we detail the process of establishing secure communication channels between all the KMIP's components and handling the exchanged messages. After that, in Section 5.4.3, we specify the official KMIP profiles which the proposed implementation is in conformance with. To conclude, in Section 5.4.4, we describe decisions that were taken during the implementation of the protocol.

5.4.1 Architecture

The architecture of the proposed KMIP implementation is composed of three main modules, as depicted in Figure 5.2: the client module, which exposes an API capable of creating and sending the protocol requests; the server module, responsible for serving clients and managing the cryptographic objects; and the common module, that is common to both client and server modules and is responsible for providing data representations of the objects of the KMIP protocol. There is also an additional module, the admin module, with a simple web user interface that provides a back-office administration of the system's users. The following subsections detail the system's modules.

Client Module

The client module provides an API capable of interacting with a server endpoint using the KMIP protocol. The module is able to communicate with any implementation of a KMIP server, as long as they follow a compatible protocol version and implement compatible profiles (described in Section 5.4.3).

The message builder component represents the core of this module and has the responsibility of building the protocol's requests. This component provides methods to set the address of the server's host, set the protocol's headers (e.g. version and authentication headers) and set the requests' payload (e.g. batch items). After building the request, the message builder calls the common module to encode the KMIP objects into binary data. Then, the data is forwarded to

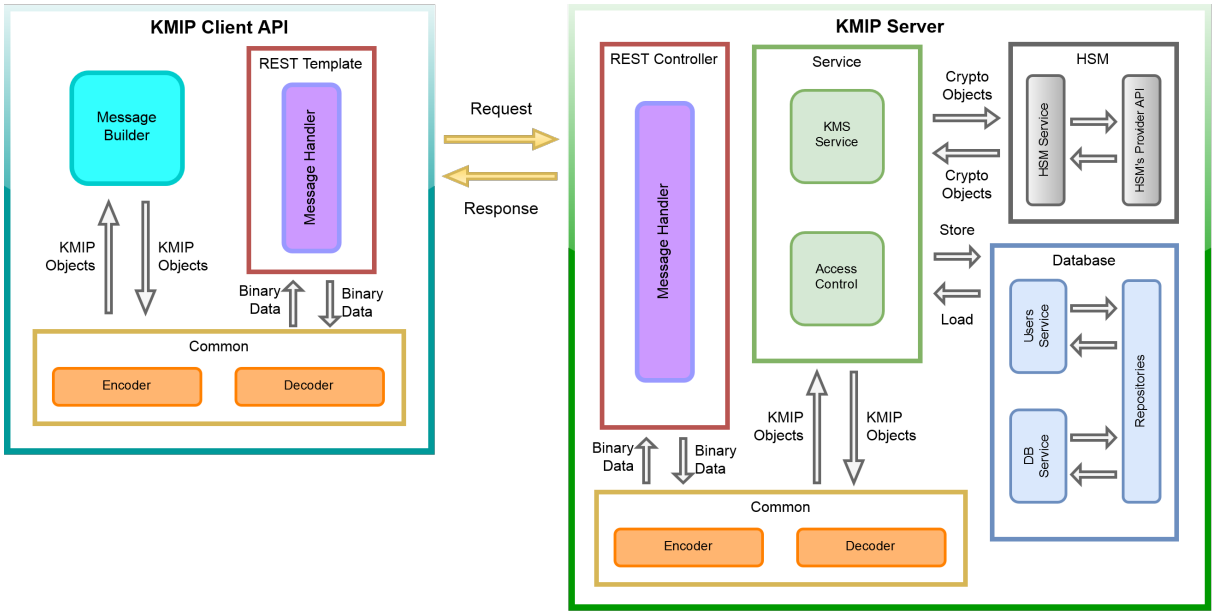

Figure 5.2: Main architecture of the KMIP protocol's implementation.

the message handler. This component is responsible for setting special headers (e.g. content type, content length and others) of the Hyper Text Transfer Protocol Secure (HTTPS) message and creating a REST template that will be used to send the message to the server endpoint.

When the client receives a KMIP response, the message handler forwards the received binary data to the common module, which decodes it into KMIP objects. These objects are then sent to the user's application by the message builder, allowing the application to retrieve the desired information.

Server Module

The server module is responsible for providing the KMIP's service, answering to requests from multiple clients that follow the KMIP protocol. The module is composed of four components: the REST controller, the service component, the database component and the HSM component.

The REST controller provides a web entrance for the service. This component handles the received messages and requests the common module to decode received binary data into KMIP objects. After the decoding process, the message handler checks the validity of the message's headers (e.g. authentication header) and forwards the decoded request to the service component.

The service component represents the core of the KMIP server. This component is responsible for implementing the main KMIP business logic and functionality. To provide such functionality, the component relies on two services: the Key Management Service (KMS) and the access control service. When the service component receives a request to a certain managed object, the access control checks the user's permission for that object. If the request is allowed, the KMS service proceeds with the operation. The KMS implements the protocol's functionality, which allows the server to perform the clients' required tasks. When performing the operations' actions, this component interacts with the HSM and database components to perform cryptographic operations and access persistent data. This component also executes scheduled tasks to, periodically, activate and deactivate managed objects when the required conditions are triggered (e.g. time conditions, such as the object's activation or deactivation dates).

The database component acts as a repository to store KMIP objects in a persistent state. The objects are mapped to entities and stored in SQL tables. These tables maintain similar constraints and relations as the KMIP objects.

The HSM component is responsible for performing cryptographic operations (e.g. generate/export cryptographic keys and cipher/decipher data). This component calls the provider's proprietary API in order to interact with the remote HSM hardware. The KMS relies on this component to protect sensitive data (e.g. cryptographic key's material) before storing it in the database. This cryptographic process is entirely performed inside the HSM hardware, which maintains a sensitive and non-exportable symmetric key that was previously generated for that purpose. The HSM component is also responsible for supporting the KMS component when performing cryptographic operations using KMIP managed keys.

After processing the requested operation, the KMS builds the operation's response and forwards it to the message handler. The message handler builds the response message, setting its headers and batching the operations' responses from the KMS service. After this process, the response is encoded under the form of binary data, with the help of the common module, and sent back to them client by the REST controller.

Common Module

This module creates a data context and provides a data representation for the KMIP objects, as specified in the KMIP specification document [18].

The common module provides data representation for attributes, managed objects, requests, responses and its fields. This module maintains special representation for primary data structures (e.g. strings, enumerations, booleans, integers, big integers, and others) as defined in the protocol's specification [18]. Each one of these data structures provide an encoding mechanism to transform its data into a Tag Type Length Value (TTLV) scheme and vice-versa. When encoding a KMIP object, the proposed implementation for the encoder applies recursion to encode each one of the object's fields, which leads to the encoding of primary data structures and reduces the complexity of the encoding process. The decoding of the data follows the same logic, taking advantage of the tags of the TTLV format to determine which object is being decoded.

The common module uses interfaces to provide high modularity and allow the implementation of custom KMIP objects, such as custom attributes or other extensions. To implement a custom attribute, the developer has to define the object's structure (i.e. its fields) and define the encoding and decoding methods, which recursively call their fields' encoding mechanisms.

Admin Module

The admin module is out of scope of the protocol's specification. This module is responsible for offering a back-office management of the server host, allowing administrators to manage the users of the system. The module interacts with the database component, providing methods for the system's administrators to create, list, enable and disable users. The access to this module can be done through the browser via a web interface, illustrated in the Figure 5.3. To allow access, the module requires special authentication from the administrators, as described in Section 5.4.2.

5.4.2 Transport

The transport of the KMIP messages between the client and the server modules is done via HTTPS and supported by a TLS channel, providing bidirectional protection of the communication. Both modules rely on a keystore and on a truststore to ensure mutual authentication in the secure channel. The client module uses an asymmetric key pair, stored in the module's keystore and recognized by the KMIP server, to authenticate the application towards the KMIP server. The KMIP users can authenticate themselves towards the system using the user authentication

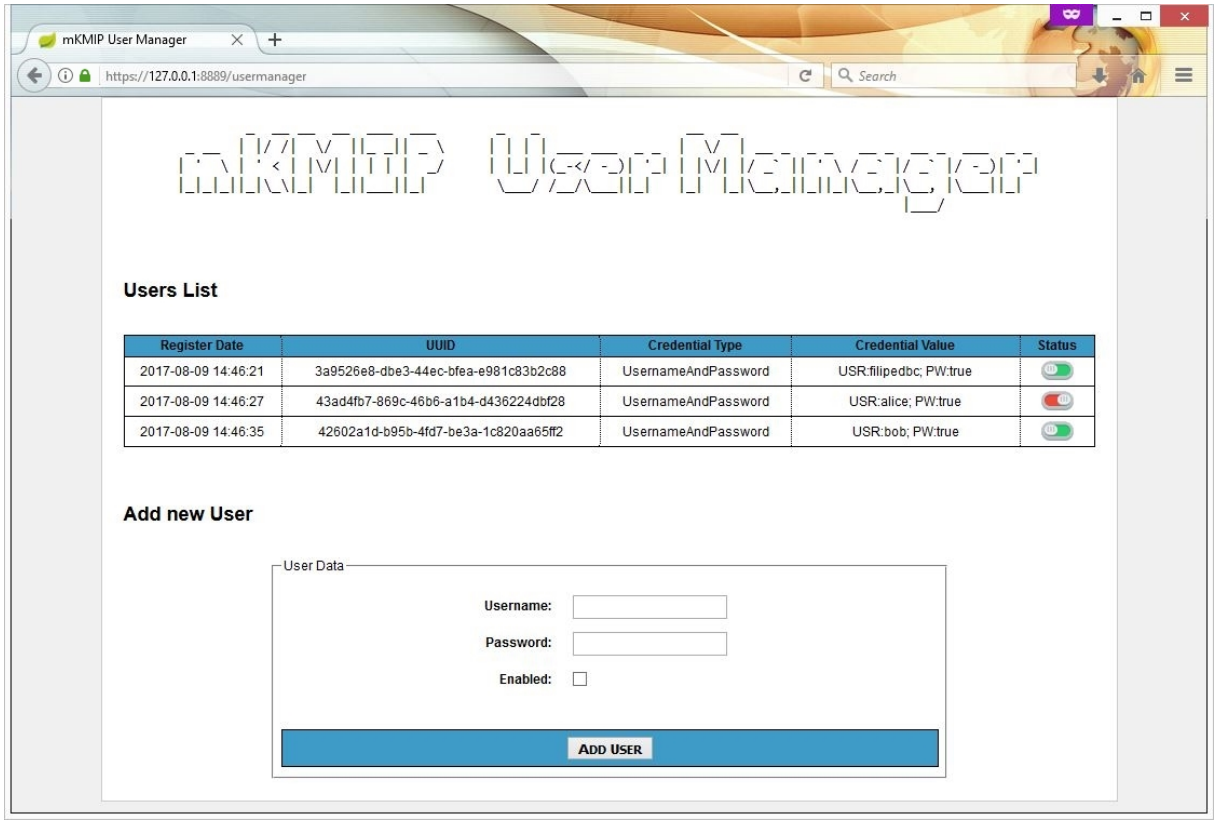

Figure 5.3: KMIP User Manager interface provided by the admin module.

methods provided by the KMIP protocol. The server module uses an exclusive keystore, which contains a private key that is only used in the protocol's context.

In order to establish a secure channel, the proposed implementation allows the use of *TLSv1.0* and *TLSv1.1* protocols for backward compatibility with legacy KMIP implementations and the use of *TLSv1.2* as recommended secure communication protocol. The implementation allows the use of the *TLS RSA WITH AES 256 CBC SHA256*, the *TLS RSA WITH AES 128 CBC SHA256* and the *TLS RSA WITH AES 128 CBC SHA* cipher suites, as recommended by the KMIP profiles described in the next Section.

The communication to the admin module is done via HTTPS channels, using the TLS protocol to establish a secure channel. As similar as the server module, this module also relies on a keystore and a truststore. The administrators authenticate themselves using certificates that are recognized and trusted by the module's truststore. The administrator's browser has to support mechanisms for the correct use of their certificates and the respective private keys.

5.4.3 Profiles

The proposed implementation of the KMIP protocol follows the official KMIP profiles [18] that define the rules and the constraints of the client/server interaction. These profiles are responsible for providing the interoperability of the protocol, allowing the interaction between different implementations of the KMIP protocol, as long as they implement compatible versions of the KMIP profiles. The KMIP clients can perform *query* requests, defined in the protocol's specification, to interrogate KMIP servers and determine their capabilities and the supported profiles.

The following describes the profiles that both the Client API and the KMIP Server from our implementation are in conformance with:

- **Baseline Basic Profile KMIP v1.3:** defines the basic KMIP functionality to implement, including the KMIP data objects, the operations and the TTLV format as the default encoding scheme for exchanged messages;
- **HTTPS Profile KMIP v1.3:** defines the format of the HTTPS messages to transport KMIP messages. This profile specifies the mandatory HTTPS headers that have to be present in the message, such as: the content-type, the content-length, the media type and the caching mechanism;
- **Baseline Profile TLS v1.2 KMIP v1.3:** defines the use of TLS protocol to establish a secure channel between the server and the client. This profile defines the configuration that has to be followed to negotiate a secure communication channel, including the protocol version, the cipher suites or the algorithms used;
- **Symmetric Key Lifecycle Profile KMIP v1.3:** defines the functionality required to create and manage symmetric cryptographic keys, including the allowed algorithms, the key sizes, and the key format types to store the symmetric key;
- **Asymmetric Key Lifecycle Profile KMIP v1.3:** defines the functionality required to create and manage asymmetric key pairs, including the allowed algorithms, the key sizes, and the key format types to store private and public keys;
- **Basic Cryptographic Profile v1.3:** defines the functionality required to perform remote encrypt and decrypt operations using managed KMIP keys, including the cryptographic

parameters used during the execution of these operations, such as the block cipher mode, the padding mode or the cryptographic algorithm used.

5.4.4 Decisions

The following describes the decisions that were made during the implementation of the KMIP protocol.

Authentication: although all the credential objects were implemented, we only considered the use of username and password credentials to authenticate the users of the proposed implementation. When a client wants to authenticate himself towards the KMIP server, a single username/password pair must be specified. The other credential types were not used, as they are out of context for this work.

Templates: the template objects were implemented but not used in this implementations, as they are considered deprecated. This decision was taken with the upgradability of the implementation in mind, as recommended in the KMIP specification [18].

Key Material: this implementation supports all key material types specified in the current version. The generated keys from create operations are saved using transparent key material structures. This decision allows an ease of the reconstruct operations on the managed cryptographic keys.

Scheduled Tasks: the present implementation executes scheduled tasks with a configurable time interval. This allows the KMIP server to automatically activate and deactivate KMIP managed objects when the time conditions are triggered (e.g. activation date and deactivation date).

Encrypt/Decrypt operations: the present implementations only considers keys, block cipher modes and padding methods that are supported by the HSM, which includes: AES and RSA keys; ECB, CBC and OFC block cipher modes; None, PKCS1 and PKCS5 padding methods.

5.5 Summary

In this chapter, the selected technologies and the implementation details for the proposed solution were described. The proposed Crypto Cloud implementation was built in Java and uses the Spring Framework [26] to wire the various components together and structure the application into different layers. The CCDS uses different technologies to provide the desired service, including: a Tomcat servlet that provides a REST endpoint to access the service; a PostgreSQL [31] database to store and manage the system's data; and a Log4J engine [1] to log the interaction between the users and the system. The Client Application supports the integration with network proxies and public clouds, such as: Dropbox and Google Drive.

Regarding the Key Management Service (KMS), we proposed an implementation for the last published version (v1.3) of the KMIP protocol. Our implementation of the KMIP Server relies on HSM hardware to perform cryptographic operations. The server also takes advantage of a PostgreSQL database and the Hibernate Framework [13] to facilitate the mapping between the KMIP objects and the database entities. The KMIP Client provides the desired functionality through an API interface. Both these components rely on the Log4J framework to log the workflow of the operations of the protocol.

6 Evaluation

This chapter presents the evaluation of the proposed Crypto Cloud solution. We start this evaluation by presenting the overall performance of our solution, describing the followed methodology and comparing the obtained results with Storekeeper's [21] system. In Section 6.2, we provide a security analysis of our solution, describing the countermeasures applied to mitigate the previously identified threats. Then, in Section 6.3, we describe how the design and implementation details of our solution satisfy the previously defined requirements. Lastly, in Section 6.4, we summarize the main aspects of the evaluation of the proposed solution.

6.1 Performance Evaluation

To evaluate the performance of our solution, several benchmark tests were carried out. The latencies measured from benchmarks were obtained using a profiler software, called JProfiler v10.0 [11], which performs the instrumentation of the running code on a Java Virtual Machine (JVM) and traces its information (e.g. threads use, sockets, method statistics, and others). This technique has a relatively low overhead associated [11]. The experiments have been performed over an Intel(R) Core(TM) i5 3230M CPU running at 2.60GHz with TurboBoost technology enabled, with 8GB of DDR3 memory running at 1600MHz, and 500GB of HDD running at 7200rpm with 32MB of cache. The machine was connected to the internet by an enterprise fiber-network, with 250Mbps of download and 100Mbps of upload speed. The OS used was Microsoft Windows 8.1 (x64) running standard services. For all experiments, the Client Application, the CCDS, the Key Management Server and the PostgreSQL databases were deployed in the same machine as the benchmarks. The HSM hardware was simulated using the Utimaco CryptoServer Simulator v5.4.6 software and deployed on the same machine as the previous components.

The benchmark tests consist of the Client Application performing several operation requests to the system. These operations includes: reading a file from the cloud, writing a new file to

the cloud, sharing a file with a user and revoking a user from accessing a file. Additionally, we also performed the same benchmarks using the existing Storepeeker's prototype [21], in order to compare the obtained results. To evaluate the performance of the Key Management Server in our system, we carried out tests that consist of performing encrypt and decrypt operations using remote stored keys. All performed benchmarks measured the latency times of each operation on our system. These operations were executed individually for 100 times, with approximately 10 seconds of interval between them, and its mean time and standard deviation were analysed. The experiments took place on September, 2017.

6.1.1 Crypto Cloud Performance

In order to evaluate the performance of the Crypto Cloud system, we obtained latency measures for each of the Crypto Cloud operations using different file sizes: 100KB, 1MB and 10MB. The same benchmarks were performed using the two Key Manager modes available from our solution: local keys and remote keys modes. During these benchmarks, we did not considered the latency times obtained from Cloud upload and download operations, since these operations are performed outside of our controlled environment and we cannot guarantee that all cloud operations are performed under the same conditions (e.g. the packets follow same network routes and use same provider's nodes). Figure 6.1 depicts the latency measurements obtained from the performed operations.

As stated in Section 5.3, the read operation consists of getting the file's content and metadata, unwrapping its keys, and decipher its content. From the obtained results, depicted in Figure 6.1, we can observe that the computation time of this operation increases with the rise of the file size parameter. This is explained by the increase in the time during the decipher process. When comparing the results from our prototype with the Storekeeper's prototype, we can observe an improve in time performance when running the application in local keys mode. In this case, the operation's time was reduced to 41% when reading 100KB files, 40% when reading 1MB files and 78% when reading 10MB files. This increase in performance is explained by the improvements done in the CCDS's operation (e.g. communication through REST interface and SQL database) and in the Client Application's internal state, which keeps the user's Private Key (PK) in-memory when running in local mode. When running the application in remote keys mode, we can observe an overhead that results from the remote use of the user's PK. This

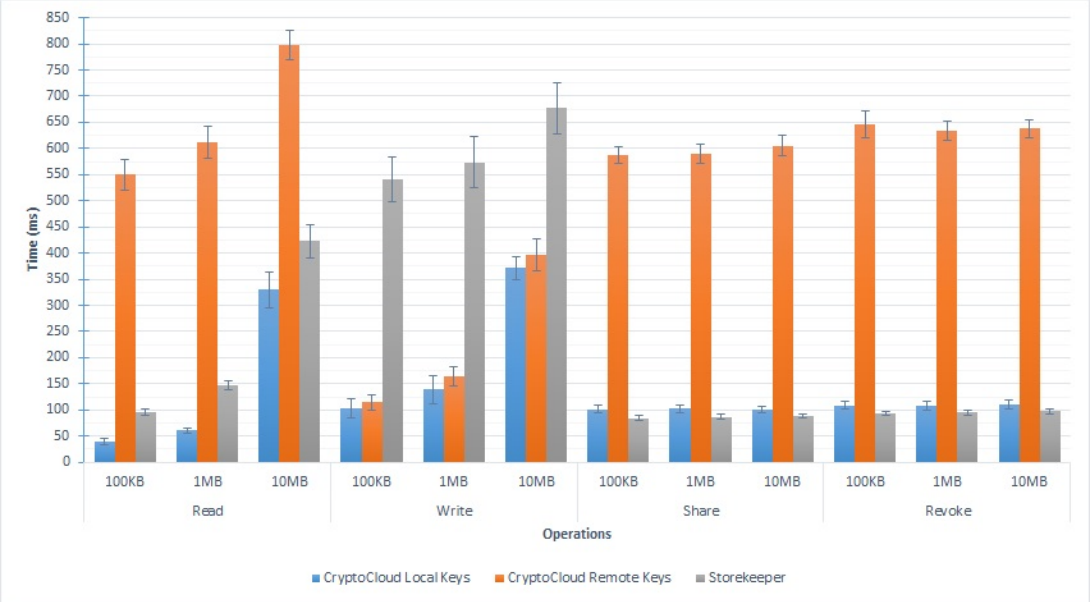

Figure 6.1: Crypto Cloud's mean latency and corresponding standard deviation per operation.

overhead originates an average increase of 477 ±7 milliseconds, as detailed in Section 6.1.2, over the whole read operation.

Similar to the read operation, the write operation also depends on the file size parameter. In this operation, both local and remote modes present similar results, as this operation only relies on the user's Public Key (PU), which is always kept in local memory. When comparing the obtained results, we can observe a larger performance increase facing the results from the Storekeeper's prototype. In this case, the results reveal a reduction of 19% to 21% of the time spent when writing 100KB files, 24% to 28% when writing 1MB files and 54% to 58% when writing 10MB files. This increase in performance results mainly from the fact that Crypto Cloud's Client Application initializes the user's Cloud Stores during the login process while Storekeeper performs the initialization on every write operation. Also, Crypto Cloud uses the most recent Clouds' APIs versions and the CCDS's operation was improved (as previously explained).

The share operation consists of wrapping the file's Read Key (RK) with another user's PU. This operation only involves the file's metadata and does not deal with the file content, which results in similar latency times for different file sizes. When comparing the obtained results with Storekeeper, we can observe an increase of the operation time. When running the application in local keys mode, Crypto Cloud takes 12% to 20% more time than Storekeeper's approach. This

decrease of performance is related to the fact that our solution verifies the validity of the user's public certificate before using them to share the file. Similar to the read operation, there is an overhead associated to this operation when running the application in remote keys mode, which results from the remote use of the user's PK to unwrap the file's RK, as detailed in Section 6.1.2.

The revoke operation consists of revoking the user access to a file and renew the file's RK. Similar to the share operation, this operation only deals with the file's metadata, presenting similar results for different file sizes. However, this operations depends of the existing file's ACL size, as the new RK will have to be distributed to the file's group members. In this case, we considered revoking a user from a file shared by two users. When comparing the results, our approach presents a decrease of performance in comparison with Storekeeper's approach. For instance, when running the application in local keys mode, the operation takes 13% to 16% more time to execute than in Storekeeper's approach. These results are explained by the fact that Crypto Cloud guarantees the integrity of its files, which adds complexity to the sharing algorithm (remember Section 4.5.3), resulting in the need to also wrap the file's Integrity Key (IK) using the new RK. Also, the application verifies the validity of users' public certificates before wrapping the new RK. When running the application in remote keys mode it is observed, once again, that a huge overhead exists. As similar to previous operations, this overhead is originated by the use of the user's PK to unwrap the file's RK before renewing it. Figure 6.2 depicts the

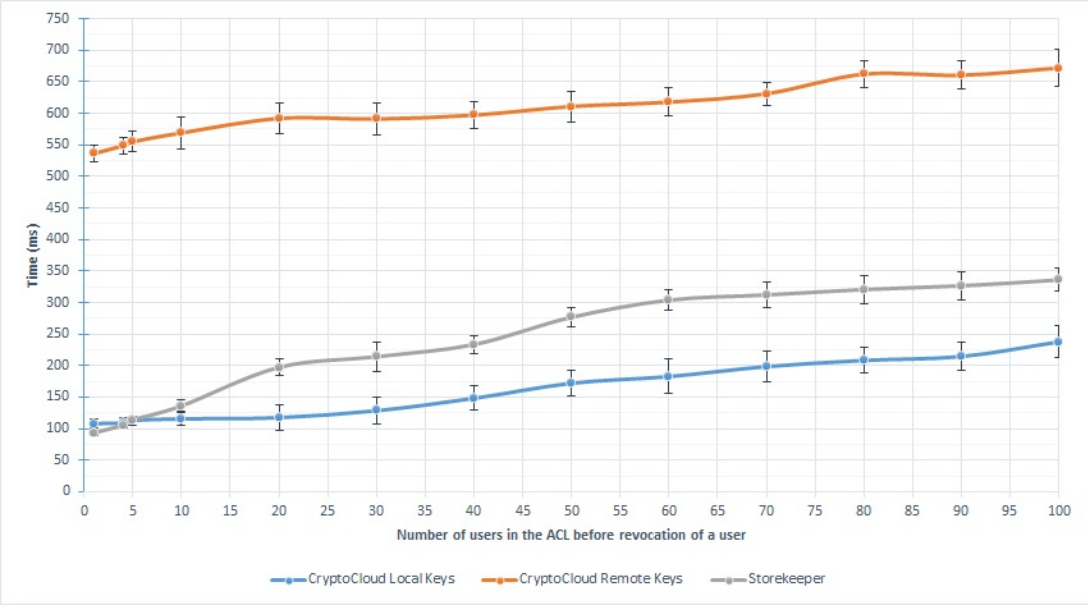

Figure 6.2: Evolution of revoke operation latency depending on the file's ACL size.

behaviour of the revoke operation for different number of users in the sharing group (1 to 100 users). Looking in detail, we can see that when dealing with files shared by more than 4 users and running the application in local keys mode, Crypto Cloud starts performing better than Storekeeper. The best gain is achieved when dealing with ACLs with 20 to 30 users, where the operation of revoking a user only takes 60% of time compared to the Storekeeper's results. This gain of performance results from the fact that the improvements performed at the CCDS (e.g. communication through REST interface, aggregation of requests, SQL database) outcome the loss inherent from the sharing algorithm.

6.1.2 KMIP Performance

In order to evaluate the performance of our prototype for the KMIP protocol, more precisely the use of remote cryptographic keys, we obtained latency measures from encrypt and decrypt operations. These operations consist of wrapping and unwrapping a 256-bit AES key using 2048-bit RSA key pairs, reproducing operations requested by the Crypto Cloud Client Application when running in remote keys mode. The experiments consist of repeating the same request operation from 1 to 10 times. The requests to the KMIP Server were sent in two different ways: sequentially and aggregated. In the first, the KMIP Client sends the request sequentially and waits for their response. In the second, the KMIP Client uses concept of KMIP Batch Item (see the KMIP specification [18]) and aggregates multiple request operation into a single KMIP request. Additionally, the same experiments were performed using JCA to serve only as reference points.

Figure 6.3 depicts the behaviour of the encrypt operation. It can be observed that a single encrypt operation takes, on average, 398 $\pm$9 milliseconds. When performing more than one encrypt operation, we can observe that bulking the operations in a single KMIP request results in a gain of performance. Looking in detail, the bulk of 2 encrypt operations represents a gain of 13%, while a bulk of 10 encrypt operations can achieve a gain in performance of 32%. Figure 6.4 depicts the behaviour of the decrypt operation. Looking in detail, we can observe that one decrypt operation takes, in average, 477 $\pm$7 milliseconds. When performing more than one decrypt operation, we can observe a similar behaviour to the encrypt operation, despite the fact that the decrypt operation is slower. In this case, performing 2 decrypt operations in a single KMIP request, can represent a gain of 17% in the time spent during execution, while

joining 10 decrypt operations can represent a gain of 31%.

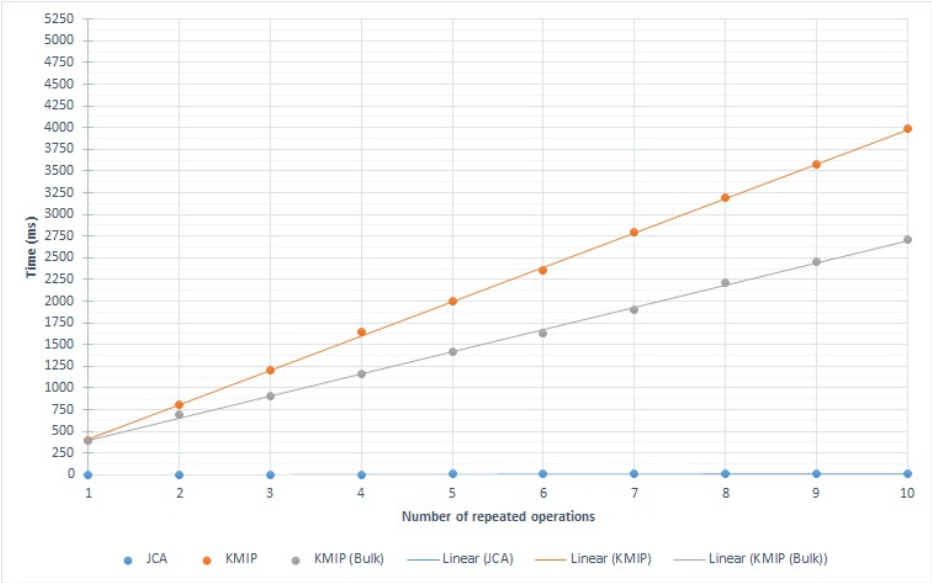

Figure 6.3: Evolution of KMIP encrypt operation's latency in result of the number of requests.

According to the discussed results, we can observe that the remote use of cryptographic keys through the KMIP protocol can be costly to an application in terms of execution time. However, the aggregation of multiple requests into a single KMIP request can decrease this cost. Also, we have to consider the fact these results were obtained on a simulated HSM and the use of real HSM hardware can lead to better results.

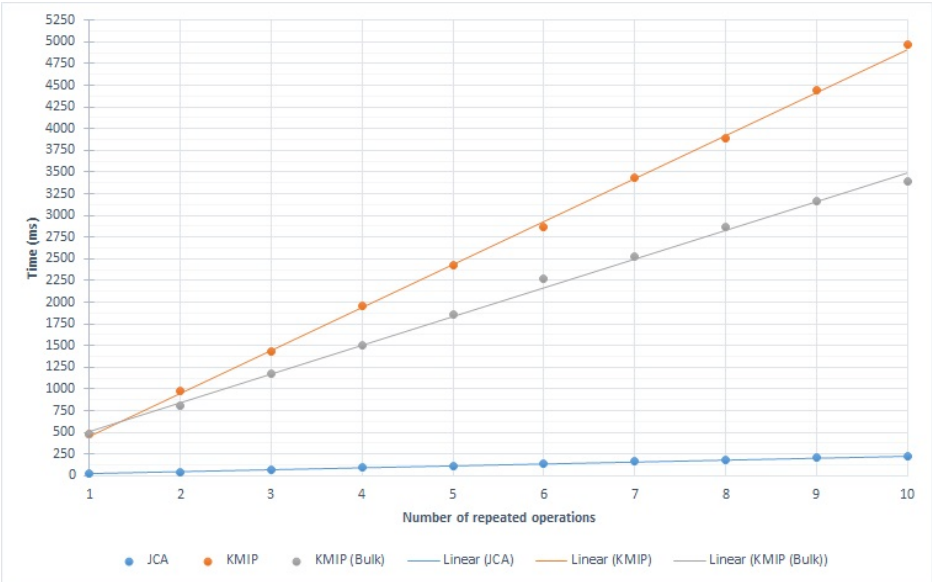

Figure 6.4: Evolution of KMIP decrypt operation's latency in result of the number of requests.

6.2 Security Analysis

The implementation of Crypto Cloud follows different mechanisms in order to mitigate the previously identified threats, in Section 4.3.2.

The **Spoofing Identity** threats consist of the violation of the system's authentication properties, where a malicious entity poses as an authorised one. In our application, an attacker could explore this category by brute-forcing token values, providing false certificates or presenting himself as a trustable certification entity. Our implementation deals with these problems by limiting the lifetime of the token to 1 hour, renewing its value, and decreasing its attack window. Also, Crypto Cloud only accepts signed certificates, which are verified before being used. The system also relies on a well-known PKI infrastructure that is assumed to be trustworthy.

Regarding threats that involve **Tampering with Data**, violating integrity properties of the system, an attacker (e.g. malicious cloud) may try to modify users' files. Our solution deals with this by implementing an integrity mechanism that relies on HMAC calculations to authenticate the content of the files.

In order to mitigate **Repudiation** threats, where a user denies performing an action over a file, our implementation counters this threats by maintaining a log of every operation performed by the system's users. This fraud prevention measure is applied in both components of the system.

Information Disclosure threats involve the violation of the system's confidentiality properties, where sensitive information could be exposed to agents (i.e. users) that are not authorized to see it. For instance, an attacker may try to read users' files or eavesdrop Crypto Cloud's messages. Our solution deals with these threats by applying cryptography over these objects, more precisely, encrypting the files' content before uploading them to the clouds and exchanging messages over secure communication channels (i.e. TLS channels). During the design of the solution, we assumed that the client's device is trustworthy (Section 4.3.1), thus we do not mitigate threats that target the client's device and expose sensitive information.

In case of **Denial of Service** threats, our implementation does not completely mitigate these threats, as the availability properties were not considered as part of the requirements for this system. However, there are possible countermeasures to mitigate these threats, such as: performing regular backups of the files' metadata, improve the infrastructure where the system

is deployed, or deploy the system on infrastructures with elastic characteristics (i.e. cloud computing infrastructures).

Regarding the **Elevation of Privilege** threats, our solution deals with this problem by implementing strict access control mechanisms based on ACLs that verify user's permission over the files on every request. Similar mechanisms are also implemented in the Key Management Server, preventing users to access other users' private objects (i.e. private keys).

6.3 Compliance with Requirements

This section describes aspects of the proposed solution and details of the implementation that lead to satisfying the previously defined requirements in Section 1.3.

Functional requirements:

1. **Crypto Cloud must support the usage of multiple cloud storage providers:** our implementation of the Crypto Cloud Client Application currently integrates Dropbox and Google Drive clouds, satisfying this requirement;

2. **Users should be able to store, access and share files without compromising their storage credentials:** by design, the client application uses an Access Token (AT) to access to the user's cloud storage space. This approach does not compromise the user's cloud credentials, as Crypto Cloud never has access to these credentials;

3. **The system must provide a secure way of storing the users' private keys:** by design, the solution relies on a Key Management Server to store and manage the users' private keys. This component follows the KMIP standard that defines how the system should work. Thus, the keys maintained at the CCDS are wrapped by the user before being sent to the CCDS;

4. **The system must provide a mechanism to validate the public keys and associated identities of the users:** by design, the system relies on a trusted PKI to certify and validate the public keys of the users. This component is responsible for issuing digital certificates, binding the user's identity with his public key. This operation is performed during the user registration process;

Non-functional requirements:

5. **The system must be able to handle a large number of users:** from the results of the performance tests, we can conclude that although there is an overhead associated, the system is capable of handling large number of users;

6. **Users must authenticate themselves in order to use the system:** by design, the users have to authenticate themselves towards the CCDS before start using the system. The users prove their entity by providing a valid username/password pair. After successful authentication, the users can use the system by following the OAuth protocol;

7. **The communication channels between the Crypto Cloud's components must be secure:** our implementation for Crypto Cloud relies on the TLS protocol to establish a secure communication channel between the system's components. In order to prove its identity, the CCDS provides a valid certificate during the establishment of the secure channel;

8. **The implementation should follow open standards, serving a wide spectrum of enterprise equipment and applications:** our prototype was built in Java, which runs in a wide spectrum of equipment. Thus, our solution follows open standards, such as, the OAuth protocol, for authorization, and the KMIP protocol, that aims at the interoperability of multiple key management systems. In addition, our implementation for the KMIP protocol takes advantage of enterprise equipment, such as HSMs, to perform cryptographic operations.

6.4 Summary

In this chapter, the evaluation of the developed solution based on the solution's performance, security and the conformance with the previously established requirements is presented.

In terms of performance, we first presented an overall performance of the Crypto Cloud, evaluating each one of the system's main operations and comparing the obtained results with Storekeeper's [21] results. The best results are achieved when the Client Application is running in local keys mode, where the user's PK is kept in-memory. When running in this mode, the read

of small files (e.g. 100KB and 1MB files) only took near 40% of the time, while the read of big files (e.g. 10MB files) took 78% of the time when comparing the same operation in Storekeeper. When performing the write of new files, our solution only took 19% to 58% of Storekeeper's time, depending on the file size. On the other hand, the extra complexity implemented on our functional algorithms to guarantee file's integrity and validity of users' public keys lead to a decrease of performance in share and revoke operations. When sharing a file, Crypto Cloud spent 12% to 20% more time than Storekeeper's and when performing a revoke operation over a file with 2 group members, the operation took 13% to 16% more time than in Storekeeper. However, when issuing a revoke over a file with more than 4 users, our application performs better than Storekeeper, as the improvements performed at the CCDS outcome the loss inherent from our functional algorithms. When running the Client Application in remote keys mode, all operations other than the write operation took longer time to execute. This is explained by the fact that the remote use of the user's PK delays the execution, on average, by 477 milliseconds and the write operation only uses the user's PU, which is stored in local memory, to wrap the file's keys.

Regarding security, our solution implemented different mechanisms to mitigate the previously identified threats. Our solution guarantees the confidentiality and integrity of stored files. Even if the attacker compromises clouds' infrastructures, the users' files remain secure. Crypto Cloud also implements authentication and authorization mechanisms to prevent illicit access. The users' identities are also verified by a trustworthy PKI infrastructure. In order to protect the exchange of messages, all the communication channels are established using secure communication protocols (i.e. TLS channels).

Through the assessment of the Crypto Cloud solution and based on its design and implementation details, we can conclude that the proposed solution meets all the previously established requirements and achieves the initially proposed goals.

7 Conclusions

7.1 Conclusions

Motivated by the need for higher portability and availability of their resources, users started migrating their services to cloud infrastructures, creating new challenges for the cloud providers, such as the security and privacy of users' resources. To address these issues, several secure cloud systems started to emerge. However, they share a common limitation: users are required to give access to their cloud credentials. Storekeeper [21] addresses this problem by using authorization tokens to access users' cloud resources. However, Storekeeper's design presents some fragilities, such as not providing integrity properties over stored files and following a weak key management scheme for the users' keys. To overcome these limitations, Crypto Cloud was proposed.

Crypto Cloud is a secure cloud system that focus on improving the performance and security of Storekeeper's solution. To achieve this, Crypto Cloud proposes several enhancements in the existing operational protocol and introduces two new components to the existing architecture: a key management server and a PKI infrastructure. The key management server decouples the CCDS from the management of the cryptographic keys and follows the KMIP protocol, providing remote management of the users' cryptographic keys. The proposed implementation for the KMIP protocol relies on various technologies, including corporate HSMs, provided by Multicert, to securely use and manage the cryptographic keys. The PKI infrastructure acts as a trusted third-party entity responsible for certification and validation of users' public keys. The protection of stored files is also enhanced by implementing proper integrity verification mechanisms. The proposed Crypto Cloud system implements newer authentication and authorization mechanisms, based on the OAuth standard, to prevent illicit access to the users' resources. Regarding performance, Crypto Cloud enhances its overall operation and can reach gains up to 40% in the execution time when comparing the same operations on Storekeeper.

The work of this thesis was developed as part of a partnership between Multicert and IST.

7.2 Future Work

The solution, herein proposed, was properly implemented and achieves its proposed goals. However, there are still a few improvements that can be taken into account in future versions of this work, such as:

- **File Versioning:** although Crypto Cloud implements version control mechanisms, we do not consider any fallback mechanism to recover old versions of files, as mentioned in Section 4.5.2. If a certain file is lost or becomes inaccessible, the owner or a writer are responsible for re-uploading the file to the system. This could be a problem if none of these users has the last version of the file stored locally on their device. One possible approach to address this problem could be maintaining old versions of the file in the owner's cloud store. However, the rotation of the file's keys should not affect the old versions, introducing new challenges that need to be addressed;

- **File Conflict:** occasionally, some file conflicts may occur during the execution of our application, namely filename conflicts and file version conflicts. Our implementation deals with this problem by moving the conflict file to a certain directory out of the user's workspace and waits for the user interaction. However, this approach may not be ideal when dealing with a large number of files. A possible solution for this problem could consists of applying an abstraction layer over the user's workspace, creating different levels based on the file's owner, file's version and filename (e.g. Alice's *file.doc* on Bob's workspace will have the following path: */shared/alice/v1/file.doc*);

- **GUI Client Application:** our implementation for the Client Application's user interface is entirely console-based, meaning that users have to input text commands in order to interact with the system. A possible improvement for the user experience could be implementing a Graphical User Interface (GUI) for the Client Application with the basic functionality. The application could also monitor the workspace directory in the user's device and automatically replicate the operations performed over the local files to the remote files, as similar to Dropbox and Google Drive client agents.

References

[1] Apache. Apache Log4J 2. https://logging.apache.org/log4j/2.x/. Online; Accessed: August 2017.

[2] BBC. Apple confirms accounts compromised but denies security breach. http://www.bbc.com/news/technology-29039294. Online; Accessed: December 2016.

[3] BBC. Dropbox hack affected 68 million users. http://www.bbc.com/news/technology-37232635. Online; Accessed: December 2016.

[4] Alysson Bessani, Miguel Correia, Bruno Quaresma, and Paulo Sousa. DepSky: Dependable and Secure Storage in a Cloud-of-Clouds. *EuroSys '11: Proceedings of the Sixth Conference on Computer Systems*, 2011.

[5] Alysson Bessani, Ricardo Mendes, Tiago Oliveira, Nuno Neves, Miguel Correia, Marcelo Pasin, and Paulo Verissimo. SCFS: A Shared Cloud-backed File System. *Proceedings of the 2014 USENIX conference on USENIX Annual Technical Conference*, 2014.

[6] Amar Ramesh Buchade and Rajesh Ingle. Key Management for Cloud Data Storage: Methods and Comparisons. *2014 Fourth International Conference on Advanced Computing & Communication Technologies*, 2014.

[7] Jolyon Clulow. On the Security of PKCS #11. *CHES'03: Proceedings of the 5th International Workshop on Cryptographic Hardware and Embedded Systems*, 2003.

[8] Miguel Correia. *Clouds-of-Clouds for Dependability and Security: Geo-replication Meets the Cloud*. Springer Berlin Heidelberg, Berlin, Heidelberg, 2014.

[9] Túlio Cícero Salvaro De Souza, Jean Everson Martina, and Ricardo Felipe Custódio. Audit and Backup Procedures for Hardware Security Modules. *Proceedings of the 7th symposium on Identity and trust on the Internet*, 2008.

[10] Stéphanie Delaune, Steve Kremer, and Graham Steel. Formal analysis of PKCS#11. *2008 21st IEEE Computer Security Foundations Symposium*, 2008.

[11] EJ Technologies. Java Profiler - JProfiler. https://www.ej-technologies.com/products/jprofiler/. Online; Accessed: August 2017.

[12] Ariel J Feldman, William P Zeller, Michael J Freedman, and Edward W Felten. SPORC: Group Collaboration using Untrusted Cloud Resources. *OSDI'10 Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, 2010.

[13] Hibernate. Hibernate: Everything data. http://hibernate.org/. Online; Accessed: August 2017.

[14] Internet Engineering Task Force. RFC 6749 - The OAuth 2.0 Authorization Framework. https://tools.ietf.org/html/rfc6749. Online; Accessed: May 2017.

[15] Daewon Kim, Yongsung Jeon, and Jeongnyeo Kim. A Secure Channel Establishment Method on A Hardware Security Module. *2014 International Conference on Information and Communication Technology Convergence (ICTC)*, 2014.

[16] Microsoft. Bitlocker drive encryption overview. https://technet.microsoft.com/en-us/library/cc732774. Online; Accessed: December 2016.

[17] Microsoft. Threat Modeling Again, STRIDE. https://blogs.msdn.microsoft.com/larryosterman/2007/09/04/threat-modeling-again-stride/. Online; Accessed: August 2017.

[18] OASIS. Key Management Interoperability Protocol Technical Committee. https://www.oasis-open.org/committees/kmip/. Online; Accessed: November 2016.

[19] OASIS. PKCS 11 Technical Committee. https://www.oasis-open.org/committees/pkcs11/. Online; Accessed: November 2016.

[20] Sancha Pereira, André Alves, Nuno Santos, and Ricardo Chaves. Storekeeper: A Security-Enhanced Cloud Storage Aggregation Service. *IEEE 35th Symposium on Reliable Distributed Systems (SRDS)*, 2016.

[21] Sancha Cipriano Pereira. Storekeeper: A Security-Enhanced Cloud Storage Aggregation Service. Master's thesis, Instituto Superior Técnico, 2016.

[22] Wolfgang Rankl and Wolfgang Effing. *Smart Card Handbook*. John Wiley & Sons, 3rd edition, 2004.

[23] Ivan Ristić. *Bulletproof SSL and TLS*. Feisty Duck Digital, 6 Acantha Court, Montpelier Road, LDN, UK, 2015.

[24] Adi Shamir. How To Share a Secret. *Communications of the ACM (CACM)*, 1979.

[25] Dinkar Sitaram, Sudheendra Harwalkar, Utkarsh Simha, Sreekanth Iyer, and Shiv Jha. Standards Based Integration of Advanced Key Management Capabilities with Openstack. *2015 IEEE International Conference on Cloud Computing in Emerging Markets (CCEM)*, 2015.

[26] Spring. Spring Framework. https://projects.spring.io/spring-framework/. Online; Accessed: August 2017.

[27] William Stallings. *Network Security Essentials: Applications and Standards*. Prentice Hall Press, Upper Saddle River, NJ, USA, 4th edition, 2010.

[28] William Stallings and Lawrie Brown. *Computer Security: Principles and Practice*. Pearson, 2nd edition, 2011.

[29] Michael Subi. An Anchor of Trust in a Digital World: Risk Management Strategies for Digital Processes - White Paper, 2015.

[30] Sudhir Bansal. Securing Passwords with Bcrypt Hashing Function. http://thehackernews.com/2014/04/securing-passwords-with-bcrypt-hashing.html. Online; Accessed: August 2017.

[31] The PostgreSQL Global Development Group . PostgreSQL: The world's most advanced open source database. https://www.postgresql.org/. Online; Accessed: August 2017.

[32] Danan Thilakanathan, Shiping Chen, and Rafael A Calvo. *Secure Data Sharing and Collaboration in the Cloud*. PhD thesis, University of Sydney, 2016.

[33] Michael Vrable, Stefan Savage, and G.M. Gm Voelker. BlueSky: A Cloud-Backed File System for the Enterprise. *Fast'12 Proceedings of the 10th USENIX Conference on File and Storage Technologies*, 2012.

Appendices

A Protocol Procedures

The following details the procedures taken during the execution of the protocol of the Crypto Cloud system. We start by describing the initialization procedure that has to be followed before during the system set-up. Then, we detail procedures of the protocols responsible for performing the available Crypto Cloud's operations: read, write, delete, share, revoke, sync and regen.

A.1 Initialization Procedure

The Crypto Cloud system entails a set of procedures that have to be followed, by both clients and administrators, before the service became available. Both Client Application and CCDS work on specific directories, where the configuration and other files required for the software to operate are stored (e.g. keystores, truststores).

The initialization procedure for the CCDS involves simple steps that need to be performed by a system's administrator. Once these steps are performed, the service becomes available for all clients. The following list describes the needed steps:

1. **Database Setup:** the prototype requires a database system in order to persistently store the managed metadata. Before starting the CCDS, a PostgreSQL database has to be created using the Crypto Cloud's database schema;

2. **Certificate Setup:** administrators have to obtain a valid digital certificate that will be used to establish a secure communication channel between the CCDS and the Client Application, ensuring the channel's authentication;

3. **CCDS Setup:** after installing the CryptoCloud's software, administrators have to set some configuration variables to access the database and keystore that were previously configured.

On the other hand, Crypto Cloud's users have to follow a longer procedure before start using the system. The following describes the operations of this procedure:

1. **Client Application Setup:** users have to install the Crypto Cloud's software on their devices. It is also necessary to configure some properties of the client application, such as: the CCDS's IP address, the KMIP Server's IP address, the KMIP's keystore and truststore in use, the Key Manager work mode and the key sizes (in bits). For default, it is assumed that the Key Manager works on local mode and uses 256-bit AES keys and 2048-bit RSA pairs. Additionally, users can also define their workspace path and proxy settings;

2. **Account Registration:** users have to register themselves through the application in order to access the service. Before registering, each user generates a new asymmetric key pair, containing a PK and a PU, through the KMIP Server. Following this process, the application requests the PKI for the certification of the user's PU. Done that, the client application generates the User Key (UK), which is wrapped using the user's PU, and the User IV. After that, the client application sends a registration request, containing the user's information to the CCDS;

3. **Account Activation:** before procced, Crypto Cloud's users need to be manually approved by an administrator. Administrators can perform this operation through a specific Crypto Cloud Admin Console, which allows an administrator to list current users and activate or deactivate their accounts;

4. **User Login:** after activation, Crypto Cloud's users can start using the service. They authenticate themselves towards the CCDS, using their private credentials. After successfully authenticated, the client application retrieves user's private information (e.g. private key identifier, public certificate, wrapped UK, user IV) from the CCDS. Depending on the Key Manager's work mode, the application unwraps the UK using the user's PK. Since the client application does not maintain any persistent state, the Sync protocol is triggered. Once this operation ends, the user can proceed with his interaction;

5. **Cloud Store Registration (optional):** before start uploading new files, each user has to register at least one cloud store. The user starts by authenticating himself towards the Cloud Provider. After successful authentication, the provider generates an authorization code, which the Crypto Cloud's client application uses to retrieve the AT from the

provider's API. After receiving the AT, the client application encrypts it using the user's UK and IV. After securing the AT, a cloud store registration request is built and sent to the CCDS.

Once the previous operations were performed, the user has access to the entire set of functionalities provided by Crypto Cloud.

A.2 Read Protocol

The read operation consists of update the user's local workspace with the most recent versions available from the cloud stores. The following list describes the steps involved in this operation for each file (see Figure A.1):

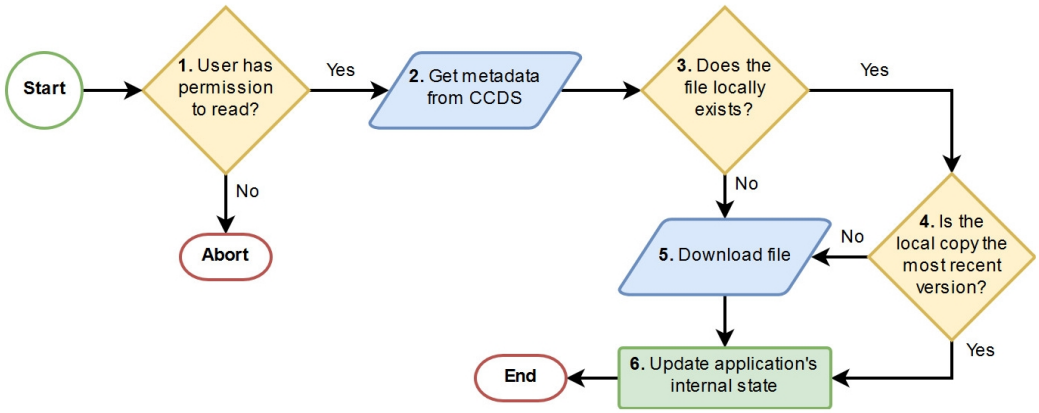

Figure A.1: Read protocol flowchart.

Step 1: Consult the user-related ACLs' entries and check if the user has, at least, read permission for that file, otherwise abort the operation for that file;

Step 2: Obtain the file's metadata from the CCDS;

Step 3: Check if a local copy of the same file exists. If exists, proceed with the next step, otherwise jump to Step 5;

Step 4: Compare the version of the local copy with the obtained metadata and check if the local copy represents the most recent version of the file. If both versions match, jump to Step 6, otherwise proceed with the next step;

Step 5: Obtain the file from the cloud store and read its content;

Step 6: Update the application's internal state with the metadata received from the CCDS.

A.3 Write Protocol

The write protocol comprises create and update operations, which consist of uploading a local copy of a file to a registered cloud store. Figure A.2 illustrates the write protocol's flowchart. The following describes the steps executed during this protocol:

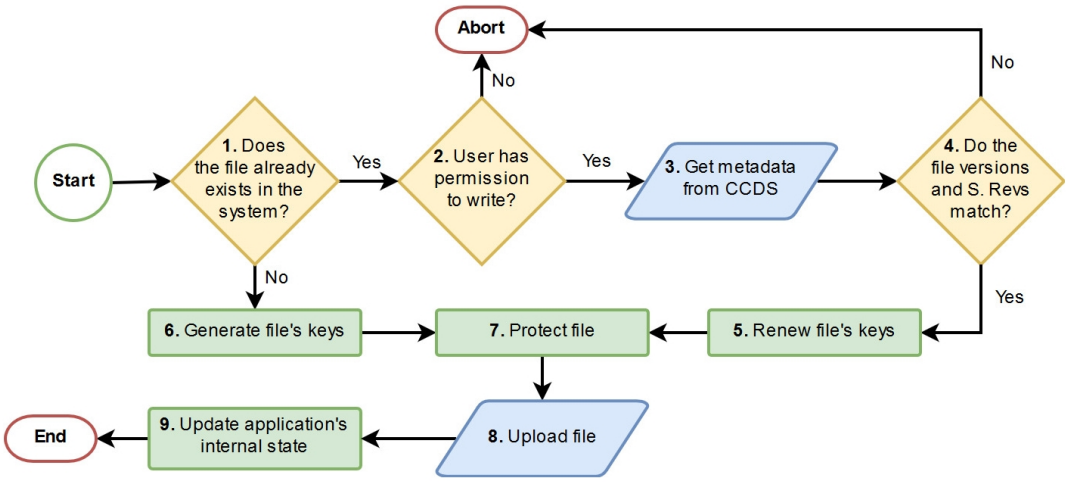

Figure A.2: Write protocol flowchart.

Step 1: Check if the file is already managed by the system. If the file is new in the system, jump to Step 6, otherwise procced to the next step;

Step 2: Consult the ACLs' entries related to the user and check if the user has permission to update the file, otherwise abort the operation;

Step 3: Obtain the file's metadata from the CCDS;

Step 4: Check if the file's metadata maintained by the client application's internal state match with the metadata received from the CCDS. During this process, the file's version number and the SR are checked. If these values don't match, the procedure is aborted;

Step 5: Renew the file's keys, which comprises the FK, its IV, and the IK. After this renewal, proceed to Step 7;

Step 6: Generate the file's cryptographic keys and IVs necessary to its protection;

Step 7: Protect the new version of the file using the keys generated on the previous steps;

Step 8: Upload the newly encrypted file to the cloud store and update the corresponding metadata on the CCDS;

Step 9: Update the application's internal state with the new file's metadata.

A.4 Delete Protocol

The delete operation can be issued to remove a file and its corresponding metadata from the system. During this operation, the following steps are executed:

Step 1: Access the ACLs entries and verify if the user has, at least, write permissions over the chosen file, otherwise abort the operation;

Step 2: Delete the file's metadata maintained at the CCDS;

Step 3: Check if the last version of the selected file is stored on a cloud store owned by the user, otherwise jump to Step 5;

Step 4: Remove the file content from its respective cloud store through the corresponding API. Done this, the operation terminates;

Step 5: Create a new ghost file structure with the information necessary to claim the cloud space from the stale file (i.e. remote filename and cloud store identifier).

A.5 Share Protocol

The share operation, is an operation that allows a user to share a certain file with other user, called grantee user. The following list describes the steps taken during the execution of this operation:

Step 1: Access the ACLs entries and verify if the user has, at least, share permissions on the selected file, otherwise abort the operation;

Step 2: Obtain the grantee user's public certificate from the CCDS;

Step 3: Check the validity of the grantee's digital certificate. Abort if the certificate is not valid;

Step 4: Encrypt the file's current Read Key (RK), using the public key obtained from the grantee's certificate;

Step 5: Update the file's ACL, creating a new entry for the grantee user, with the assigned permission and previously encrypted RK.

A.6 Revoke Protocol

The revoke operation allows a user, with share permissions over a file, to revoke access from a member of the sharing group. The revoke protocol performs the following sequence of steps:

Step 1: Consult the ACLs entries and check if the user has share permissions or is the owner of the chosen file, otherwise abort the procedure;

Step 2: Obtain the digital certificates from the members of the current file's sharing group (excluding the revoked user), which are maintained by the CCDS;

Step 3: Check the validity of the obtained certificates. Abort the operation if an invalid certificate is detected;

Step 4: Generate a new file's RK to replace the existing one;

Step 5: Retrieve the public keys from the valid certificates and use them to encrypt the new RK for each member of the sharing group;

Step 6: Remove the ACL entry corresponding to the revoked user and update the other members' entries with the newly encrypted read keys.

A.7 Sync Protocol

The sync operation is a complex process responsible for maintaining the consistency of the client application's internal state and local files. This operation also triggers the file homing and garbage collector mechanisms. During this process, the following steps are performed (see Figure A.3):

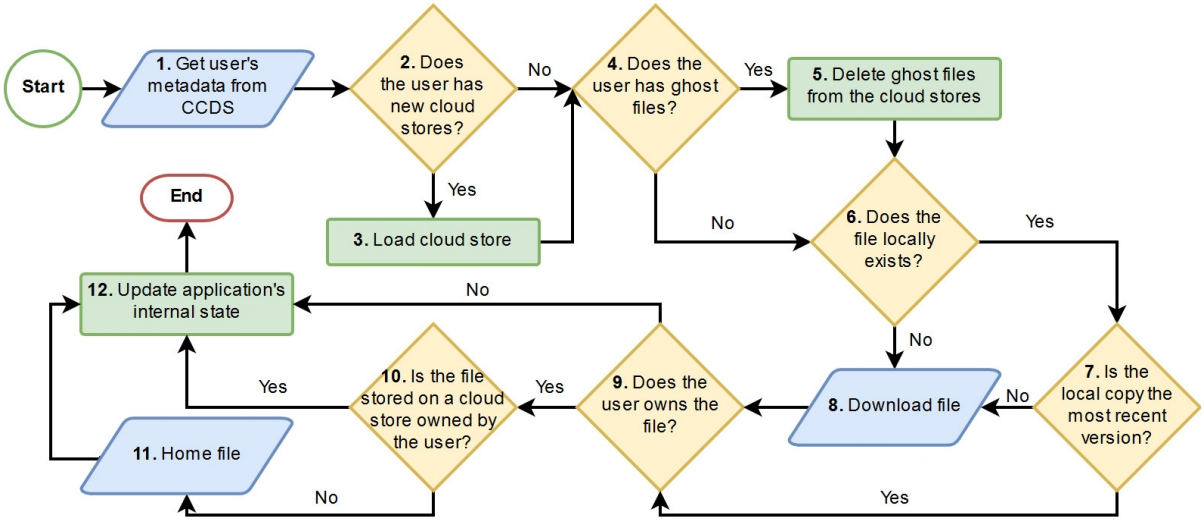

Figure A.3: Sync protocol flowchart.

Step 1: Obtain the user's metadata from the CCDS, including user information, cloud stores and files' metadata;

Step 2: Verify if the received metadata contains information about newly registered cloud stores. Jump to Step 4 if no recent cloud stores were detected, otherwise proceed with the next step;

Step 3: Load and login onto the new cloud stores using their respective AT;

Step 4: Check if the user has ghost files to delete from his cloud stores. Jump to Step 6 if no ghost files were found;

Step 5: Delete the ghost files from their corresponding cloud stores;

Step 6: Check if a local copy of each one of the received files exists. If exists, proceed with the next step, otherwise jump to Step 8;

Step 7: Compare the version of each local copy with the received metadata and check if each copy is up to date. If both versions match, jump to Step 9, otherwise proceed with the next step;

Step 8: Download the remote file from the cloud and read its content;

Step 9: Verify if the user is the owner of the current file. If not, proceed to Step 12;

Step 10: Check if the remote file is stored on a cloud store owned by the user. Proceed if not, otherwise, jump to Step 12;

Step 11: Trigger the file homing algorithm to reallocate the file to a cloud store owned by the user;

Step 12: Update the application's internal state with the received metadata.

A.8 Regen Protocol

The regen operation follows a special procedure that allows a user to generate a new pair of asymmetric keys to be used inside the application's context. This procedure is composed of the following steps:

Step 1: Contact the CCDS in order to obtain the cryptographic keys of files that the user has access;

Step 2: Unwrap each file's RK using the user's current PK;

Step 3: Contact the Key Manager Server (KMIP Server) to request the generation of a new asymmetric key pair (i.e PK and PU);

Step 4: Request the PKI to generate a digital certificate of the new user's PU;

Step 5: Wrap the UK using the new user's PU;

Step 6: Wrap the previously obtained RKs using the new PU;

Step 7: Update the corresponding user's metadata, maintained at the CCDS, with the new PK identifier, PU certificate and wrapped keys (i.e. wrapped UK and wrapped RKs);

Step 8: Log off the current user, clearing the application's internal state.