Crypto Cloud

Filipe Duarte Bento Custódio
filipecustodio@tecnico.ulisboa.pt

Instituto Superior Técnico, Lisboa, Portugal

November 2017

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 [11, 10]. 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 [9] 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 [14, 12] protocols, for the establishment of secure communication channels between the interacting parties, and the OAuth [8] protocol, for access control and authorization of users' requests.
Keywords: Cloud Storage Services, Security and Privacy, Remote Key Management, Key Management Interoperability, Hardware Security Modules

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 accessibility, availability 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. 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 [11] 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). The first is responsible for implementing the Key Management Interoperability Protocol (KMIP) standard [9], which allows the remote access and management of user's cryptographic keys. The second certificates the users' keys, guaranteeing proper authentication when protecting and sharing sensitive files. The solution also follows widely used standards for web applications, such as the TLS/SSL [14, 12] or the OAuth [8] protocols.

1.1. 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.

1.2. Structure of this Document

This document is organised as follows: Section 2 presents an overview of the related work on the subject of secure cloud storage solutions; Section 3 describes the main aspects of the proposed solution; Section 4 describes the implementation details of the solution; Section 5 presents the evaluation of the proposed solution and the analysis of the obtained results; and Section 6 concludes this document.

2. Related Work

In this section, 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, a comparative analysis is presented.

BlueSky [16] 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 [7] 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 [11, 10] 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 Application Programming Interface (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 Storekeeper Directory Server (SDS), resulting in a decrease of the application's performance.

Table 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.

3. Crypto Cloud

This section presents an overview of the proposed solution for a secure cloud system. Crypto Cloud is based on the Storekeeper's concept, introducing two new components to the existing architecture, namely the Key Management Server and a PKI. The Key Management Server is a remote server that implements the KMIP protocol, and is responsible for the access, storage and management of the users' cryptographic keys. 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 public key. 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. 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.

3.1. Architecture

This section presents the Crypto Cloud's architecture, depicted in Figure 1. The system consists of five components: the client application,

System	Data Sharing	Data Confidentiality	Data Integrity	Data Versioning	Metadata Confidentiality	Metadata Integrity	Key Distribution	Multi-Cloud
BlueSky [16]		✓	✓	✓				
SPORC [7]	✓	✓	✓	✓	✓	✓	✓	
DepSky [4]		✓	✓	✓		✓	✓	✓
SCFS [5]	✓	✓	✓	✓			✓	✓
Storekeeper [11]	✓	✓		✓			✓	✓

Table 1: Comparison of existing solutions for secure cloud systems.

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; the Cloud Stores, which are responsible for providing users' storage space; and a PKI to certify and validate the users' public keys. The interaction between these components is secured using the TLS communication protocol.

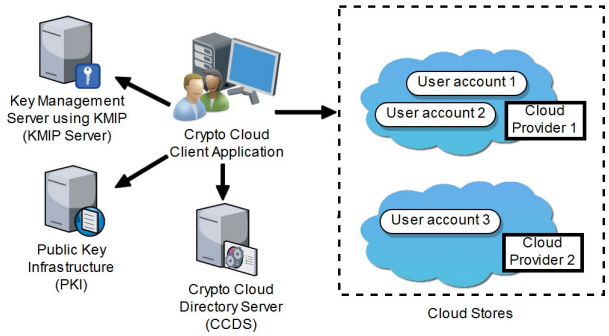

Figure 1: Crypto Cloud's system architecture.

3.1.1. 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.

The application provides an interface layer that allows the interaction with the user, through a command line console. This layer forwards the received input to a service layer that performs the required operations. The service layer is responsible for performing the users' operations, guaranteeing the system's functionality and security properties. 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.

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 and his managed files.

The communication manager is the module responsible for interacting with the CCDS. During the login process, it authenticates the user towards the CCDS, which provides an authorization token to access the user's resources.

The key manager controls the user's cryptographic key pair. It relies on the KMIP Client API to access the cryptographic keys and perform the cryptographic operations. Depending on the key usage, the module offers 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 and reconstruct the key. When using this mode, 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 user's key.

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, and a conflict directory, where the conflict files are moved.

3.1.2. Crypto Cloud Directory Server

The 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. The managed information is persistently stored using a structured database. 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. The access control mechanisms are checked on every request to prevent improper access to the managed information.

3.1.3. Key Management Server

The system includes a Key Management Server 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 [9], allowing users to generate, obtain, use and destroy their cryptographic keys. It requires the authentication of every request handled by the server. In order to serve its clients, our implementation for the KMIP protocol uses HSM hardware to generate, access and use managed cryptographic keys.

3.1.4. 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 an access token after proper authentication of the account's owner. These tokens are used to access the stores using the respective provider's API, allowing users to create, access, modify and delete stored files.

3.1.5. Public Key Infrastructure

The PKI component is 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. This mechanism allows clients to check if the certificate was revoked before using it.

3.2. User Authentication

The following describes the authentication mechanisms that each one of the system's components follow for the user authentication.

3.2.1. Crypto Cloud Authentication

In Crypto Cloud, clients authenticate themselves towards the CCDS using 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 authorization token to the client. After receiving the token, the client is allowed to use it by following the OAuth Protocol [8] in order to access resources from the CCDS. When the token's lifetime ends, the client has to repeat the authentication process in order to get a new one. This protocol is widely used by web applications and allows the use external authentication services (e.g. Facebook Login or Google Authentication Services).

3.2.2. Cloud Store Authentication

Our solution follows a similar approach as Storekeeper [10, 11] for the user authentication towards the Cloud Stores. The users authenticate themselves by login into the provider's website and allowing access to the Crypto Cloud application. This generates an access token that is used to access the user's cloud stores through the provider's API without compromising the users' credentials. Since the usage of the access token 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 access token on the CCDS, the client application uses a symmetric key (UK), to wrap and read-protect the token from the CCDS and the remaining users. The UK key is then wrapped using the user's public key (PU).

3.2.3. 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 [9], which requires user authentication on every request. The users authenticate themselves by providing their username and password credentials. These credentials are only handled by the KMIP Protocol itself, meaning that they are neither persistently stored at the CCDS nor at the Client Application.

3.3. 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. The following describes the algorithms responsible for supporting that functionality.

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

3.3.1. Base Algorithm

The functional process of our proposed solution involves management and protection of users' files. Thus, the client application has to ensure the data confidentiality and integrity before uploading the file to the cloud. The first step that our application takes is to generate two new symmetric keys, the File Key (FK) and Integrity Key (IK). Then, it uses the IK to perform a HMAC calculation over the file's content, producing a fixed-size hash result. After that, the algorithm performs the encryption of the file using the FK and a pseudo-random IV. The

produced hash result is concatenated alongside with the ciphered content, as illustrated in Figure 2, and the protected file is uploaded to the user's cloud store. In order to prevent illicit use of the keys, the FK and the IK keys are wrapped using the user's PU. To conclude, the file's metadata and the wrapped keys are stored at the CCDS and the file is registered in the system. When a user tries to read a file, the application performs the reverse operations.

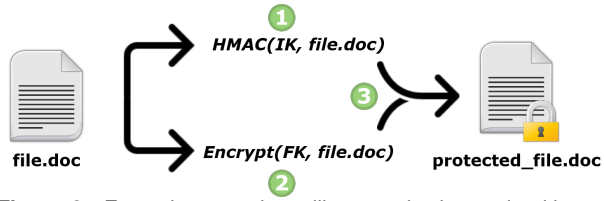

Figure 2: Example scenario to illustrate the base algorithm. First, (1) the HMAC of file.doc is calculated using IK. Then, (2) the content of file.doc is encrypted using FK. Lastly, (3) the result from the HMAC operation is merged with the result from the encrypt operation.

Considering this combined approach, our solution can guarantee the confidentiality and integrity of users' files, preventing the CCDS, the cloud providers, and other agents from reading or compromising user's private files.

3.3.2. File Sharing

Although the introduced base algorithm ensures the confidentiality and integrity of stored files, it lacks in providing sharing functionalities. A possible approach for this mechanism is to wrap the file's FK and IK using the *grantee*'s PU. By doing this, the *grantee* could unwrap this keys using his PK and access the file's content. However, revoking access to a *grantee* from a file requires the regeneration of the FK and IK keys and, consequently, the re-encryption and re-upload of the file. To overcome this issue and implement efficient access revocation, Crypto Cloud follows the Storekeeper's approach [11], avoiding re-encryption of the entire file and redistribution of both new keys, which consists of three new techniques:

1. **Readers have access to a Read Key (RK):** instead of granting direct access to the FK, an intermediary symmetric key is used, named the Read Key (RK). This RK is used to wrap the FK, which encrypts the file's content. Then, the RK is wrapped using each one of the *grantee*'s PU;

2. **Revocations produce a new RK:** every time a revocation occurs, a new RK has to be generated. This process includes the wrapping of the existing FK using the, which is distributed to the new set of members of the sharing group using their PUs;

3. **Updates generate a new FK:** every time an update to a file occurs, the file's FK is replaced with a new FK to encrypt the new file's version. The new FK is wrapped using the current RK in order to allow the members of the sharing group to continue reading future updates. Additionally, Crypto Cloud renews the file's IK to decrease its exposure.

By following this algorithm, it is possible to disrupt the access to the file's content from old group members. This approach presents a low overhead, since it introduces a new intermediary key, the 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.

3.3.3. File Homing

In order to handle the files on cloud stores, Crypto Cloud follows Storekeeper's Staging Space approach, which maintains staged files and respects a remote-read local-write policy [11]. 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.

Crypto Cloud counters these problems by employing a File Homing technique. This technique, similarly to Storekeeper [11], consists of periodically reallocate staged files back to its owner's cloud stores. The file homing process can only be performed by files' owners. This reallocation process consists of three tasks: (1) download the encrypted file, (2) upload the download file to one of the owner's cloud stores, (3) update the file's metadata at the CCDS with the new file's location.

However, after the reallocation, the old location still occupies someone else's precious cloud space. 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 later notified to remove the Ghost Files hosted on their cloud stores.

4. Implementation

The following describes the implementation of the proposed Crypto Cloud system. In Section 4.1, we present an overview of the implementation, describing the technologies chosen. Then, in Section 4.2, we present our implementation for the KMIP protocol, which is an important component of our solution.

4.1. Overview

Our prototype for a Crypto Cloud system was fully implemented in Java 8, and takes advantage of Spring Framework v4 [13], 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 configurable Java Beans to wire various components together, structuring the application into multiple layers, increasing the flexibility of the implementation. In 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 that allows the creation of a 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, decreasing the number of interactions during the execution of the Crypto Cloud's protocol. The authorization to the CCDS's resources is done through the OAuth protocol [8], a common authorization standard for web applications. The developed prototype couples the Authorization Server and the Resource Server inside the CCDS's machine. The users' passwords are encoded using the BCrypt [15], 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 database to securely store and manage the data. The access to the database is done through the Spring Data JPA framework, reducing the effort needed to access the database's tables.

The Client Application interacts with the CCDS through its REST endpoint. The requests are encoded in 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 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. Regarding cryptography, our prototype relies on JCA APIs to perform cryptographic operations. When dealing with symmetric cryptography, we use 256-bit AES keys, with a pseudo-random 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, were considered secure and not broken.

Crypto Cloud also implements logging mechanisms, both in the Client Application and in the CCDS, registering important information for future analysis of potential attacks or failures. 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.

4.2. KMIP

The KMIP protocol is a well-defined standard and widely accepted protocol for the remote use and management of cryptographic keys. However, there was no stable open source implementation of the protocol available for use. This section presents the implementation for the last published version of the KMIP (version 1.3) [9].

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, the Spring Framework v4 [13], the Log4J v2 framework [1] and the PostgreSQL DBMS. In overall, the proposed implementation for the KMIP protocol comprises 45000 lines of code.

4.2.1. Architecture

The architecture of the proposed KMIP implementation is composed of three main modules (see Figure 3): the client module, which exposes an API capable of create and send 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.

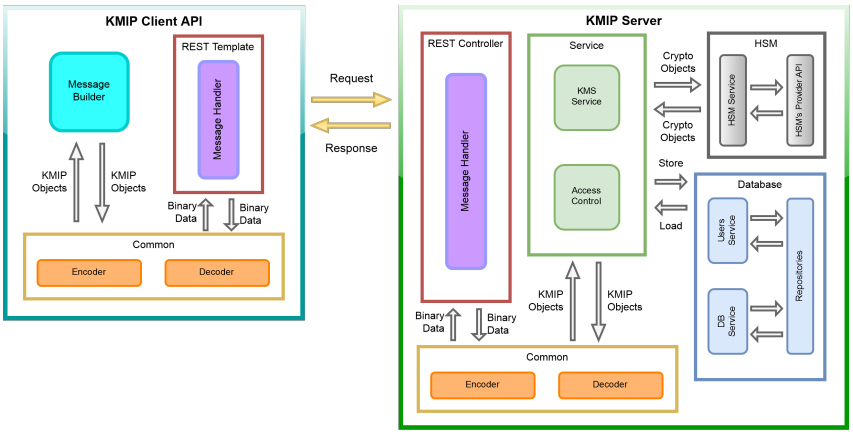

Figure 3: Main architecture of the KMIP protocol's implementation.

Client Module

The client module provides an API capable of interacting with a server endpoint using the KMIP protocol, as long as they follow a compatible protocol version and implement compatible profiles.

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 and set the request's payload. 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 the message handler, which is responsible for building the HTTPS message and sending it 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, which retrieves the desired information.

Server Module

The server module is responsible for providing the KMIP's service. 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, handling the received message to the common module to decode the request into KMIP objects. After this, the message handler forwards the decoded request message to the service component.

The service component represents the core of the KMIP server. To provide the KMIP functionality, the component relies on two services: the KMS service and the access control service. When the service component receives a request to a 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 allows the server to perform the clients' required tasks. When performing operations' procedures, this component interacts with the HSM and database components to perform cryptographic operations and access persistent data.

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. This component calls a 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.

After processing the requested operation, the KMS builds the operation's response and forwards it to the message handler. 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 offers a data representation for the KMIP objects, as specified in the protocol's specification [9]. The module provides data representation for attributes, managed objects, requests, responses and its fields. Each one of these data structures provide an encoding mechanism to transform its data into a TTLV scheme under binary form and vice-versa. When encoding a KMIP object, our implementation for the

encoder applies recursion to encode each one of the object's fields. 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 well defined interfaces, providing high modularity and allowing the creation of custom KMIP objects, such as custom attributes or other extensions.

4.2.2. Transport

The transport of the messages between the client and the server modules is done via HTTPS through TLS channels, providing bidirectional protection of the communication. Both modules rely on a keystore and a truststore to ensure mutual authentication in the secure channel. The KMIP users can further authenticate themselves towards the system using the protocol's authentication methods. In order to establish a secure channel, the implementation allows the use of *TLSv1.0*, *TLSv1.1* and *TLSv1.2* as secure communication protocols.

4.2.3. Profiles

The present implementation of protocol follows official KMIP profiles [9] that define rules and constraints for the client/server interaction. The following list describes the profiles that both our client and our server are in conformance with.

- **Baseline Basic Profile KMIP v1.3:** defines the basic KMIP functionality to implement, such as: the KMIP objects, the operations and the TTLV format as the default encoding scheme;
- **HTTPS Profile KMIP v1.3:** defines the format of the HTTPS messages to transport KMIP messages, specifying the mandatory HTTPS headers that have to be present in the message;
- **Baseline Profile TLS v1.2 KMIP v1.3:** defines the use and configuration of the TLS protocol to establish a secure channel between the server and the client;
- **Symmetric Key Lifecycle Profile KMIP v1.3:** defines the functionality required to create and manage symmetric cryptographic keys;
- **Asymmetric Key Lifecycle Profile KMIP v1.3:** defines the functionality required to create and manage asymmetric key pairs;
- **Basic Cryptographic Profile v1.3:** defines the functionality required to perform remote encrypt and decrypt operations using managed KMIP keys.

5. Evaluation

This section introduces the evaluation of Crypto Cloud's solution, describing the followed methodology and comparing the obtained results with the Storekeeper's [11] system.

5.1. Methodology

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 [6], which performs the instrumentation of the running code on a JVM. 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, and the Key Management Server were deployed in the same machine as the benchmarks. The HSM hardware was simulated using the Utimaco CryptoServer Simulator v5.4.6.

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 on the existing Storepeeker [11] prototype, in order to compare the obtained results. These operations were executed individually for 100 times, with approximately 10 seconds of interval, and its mean time and standard deviation were analysed. The experiments took place on September, 2017.

5.2. Crypto Cloud Performance

In order to evaluate the performance of Crypto Cloud, we obtained latency measures from each of the Crypto Cloud's operations using different file sizes: 100KB, 1MB and 10MB. The benchmarks were performed using the two Key Manager modes 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. Figure 4 depicts the latency measurements obtained from the performed operations.

The read operation consists of getting the file's content and metadata, unwrapping its keys, and decipher its content. From the obtained results

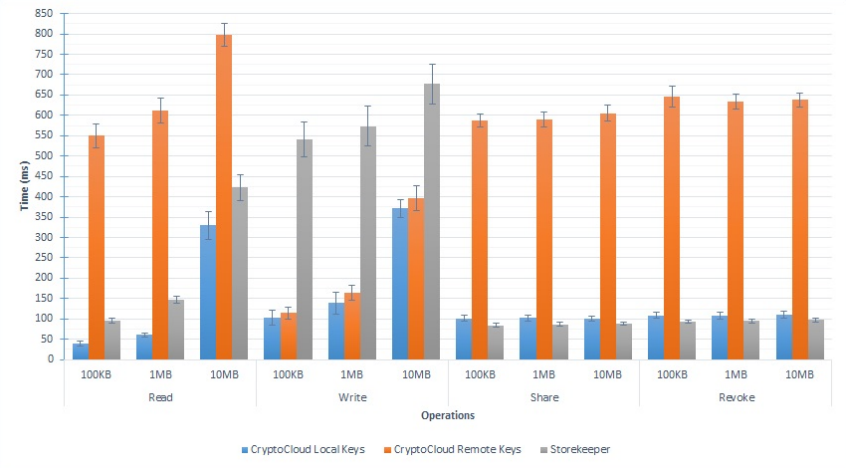

Figure 4: Crypto Cloud's mean latency and corresponding standard deviation per operation.

we can observe that the time of this operation increases with the rise of the file's size parameter. This is explained by the increase in the time during the decipher process. When comparing the results from our prototype with 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 of performance is explained by the improvements done in the CCDS's operation and in the Client Application's internal state, which keeps the user's 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.

The write operation also depends on the file's size parameter. This operation presents similar results for both local and remote modes, as the operation only relies on the user's PU that is kept in local memory. When comparing the obtained results, we can observe a large performance increase facing the results from the Storekeeper's prototype. In this case, the results reveal a reduce from 19% to 21% of time spent when writing 100KB files, 24% to 28% when writing 1MB files and 54% to 58% when writing 10MB files. This increase of 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.

The share operation consists of wrapping the file's RK with another user's PU. This operation only involves the file's metadata and does not deals with the file's content, which results in similar latency times for different file's sizes. When comparing the obtained results with Storekeeper's results, we can observe an increase of the operation's 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. As 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.

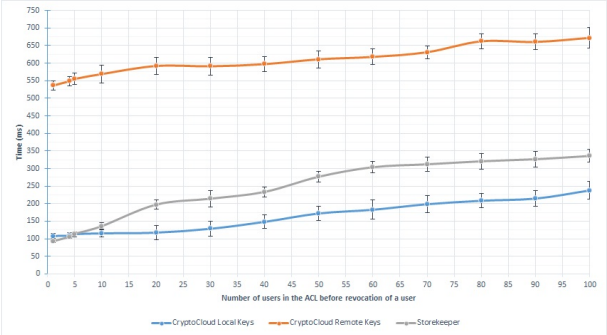

Figure 5: Evolution of revoke operation's latency in result of the file's ACL size.

The revoke operation consists of revoking the user access to a file and renew the file's RK. As similar to the share operation, this operation only deals with the file's metadata, presenting similar results for different file's sizes. However, this operations depends on the existing file's ACL size, as the new RK will have to be distributed to the file's group members. For 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 Store-

keeper'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, as the file's IK needs to be wrapped 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 5 depicts the 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 done at the CCDS (e.g. communication through REST interface, aggregation of requests, SQL database) outcome the loss inherent from the sharing algorithm.

6. Conclusions

The accentuated growth of users created new challenges for cloud providers, such as the security and privacy of users' resources. Recently, some systems emerged in order to overcome with the cloud security issues, however they all share a common limitation: users are required to give access to sensitive information (e.g. their cloud credentials). Storekeeper [11] addresses this problem by using authorization token 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 users' cryptographic keys..

The presented solution, Crypto Cloud, is a secure cloud system that focus on improving the performance and security of Storekeeper's solution. To achieve this, the proposed solution 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 open protocol, relying on corporate HSMs to remote access and manage the users' 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. Crypto Cloud implements newer authentication and authorization mechanisms based on OAuth open standard to prevent illicit access to users' resources. Regarding performance, Crypto Cloud enhances its overall operation and can reach gains up to 40% of the execution time when compared the same operation on Storekeeper.

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 to achieve this work.

I extend my gratitude to my parents and my colleagues at Multicert for their support, encouragement and motivation.

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

[7] 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.

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

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

[10] 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.

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

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

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

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

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

[16] 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.