A Secure Client Side Deduplication Scheme in Cloud Storage Environments

Nesrine Kaaniche, Maryline Laurent
Institut Mines-Telecom, Telecom SudParis, UMR CNRS 5157 SAMOVAR
9 rue Charles Fourier, 91011 Evry, France
Email: {nesrine.kaaniche, maryline.laurent}@telecom-sudparis.eu

***Abstract*—Recent years have witnessed the trend of leveraging cloud-based services for large scale content storage, processing, and distribution. Security and privacy are among top concerns for the public cloud environments. Towards these security challenges, we propose and implement, on OpenStack Swift, a new client-side deduplication scheme for securely storing and sharing outsourced data via the public cloud. The originality of our proposal is twofold. First, it ensures better confidentiality towards unauthorized users. That is, every client computes a per data key to encrypt the data that he intends to store in the cloud. As such, the data access is managed by the data owner. Second, by integrating access rights in metadata file, an authorized user can decipher an encrypted file only with his private key.**

***Keywords* – Cloud Storage, Data Security, Deduplication, Confidentiality, Proof of Ownership**

I. INTRODUCTION

Nowadays, the explosive growth of digital contents continues to rise the demand for new storage and network capacities, along with an increasing need for more cost-effective use of storage and network bandwidth for data transfer.
As such, the use of remote storage systems is gaining an expanding interest, namely the cloud storage based services, since it provides cost efficient architectures. These architectures support the transmission, storage in a multi-tenant environment, and intensive computation of outsourced data in a pay per use business model.
For saving resources consumption in both network bandwidth and storage capacities, many cloud services, namely Dropbox, wuala and Memopal, apply client side deduplication ([5], [10]). This concept avoids the storage of redundant data in cloud servers and reduces network bandwidth consumption associated to transmitting the same contents several times.

Despite these significant advantages in saving resources, client data deduplication brings many security issues, considerably due to the multi-owner data possession challenges [10]. For instance, several attacks target either the bandwidth consumption or the confidentiality and the privacy of legitimate cloud users. For example, a user may check whether another user has already uploaded a file, by trying to outsource the same file to the cloud.

Recently, to mitigate these concerns, many efforts have been proposed under different security models ([3], [8], [12], [13], [16]). These schemes are called *Proof of Ownership* systems (PoW). They allow the storage server check a user data ownership, based on a static and short value (e.g. hash value). These security protocols are designed to guarantee several requirements, namely lightweight of verification and computation efficiency.
Even though existing PoW schemes have addressed various security properties, we still need a careful consideration of potential attacks such as Data Leakage and poison attacks, that target privacy preservation and data confidentiality disclosure.

This paper introduces a new cryptographic method for secure Proof of Ownership (PoW), based on the joint use of convergent encryption [15] and the *Merkle-based Tree* [11], for improving data security in cloud storage systems, providing dynamic sharing between users and ensuring efficient data deduplication.
Our idea consists in using the *Merkle-based Tree* over encrypted data, in order to derive a unique identifier of outsourced data. On one hand, this identifier serves to check the availability of the same data in remote cloud servers. On the other hand, it is used to ensure efficient access control in dynamic sharing scenarios.

The remainder of this work is organized as follows. First, Section II describes the state of the art of existing schemes, introducing the general concept of PoW protocols and highlighting their limitations and their security challenges. Then, Section III introduces the system model. Section IV presents our secure PoW scheme and gives a short security analysis. Finally, a performance evaluation is presented, in Section VI before concluding in Section VII.

II. RELATED WORKS AND SECURITY ANALYSIS

The Proof of Ownership (PoW) is introduced by Halevi [8]. It is challenge-response protocol enabling a storage server to check whether a requesting entity is the data owner, based on a short value. That is, when a user wants to upload a data file (D) to the cloud, he first computes and sends a hash value $hash = H(D)$ to the storage server. This latter maintains a database of hash values of all received files, and looks up $hash$. If there is a match found, then D is already outsourced to cloud servers. As such, the cloud tags the cloud user as an owner of data with no need to upload the file to remote storage servers. If there is no math, then the user has to send the file data (D) to the cloud.
This client side deduplication, referred to as *hash-as-a-proof* [16], presents several security challenges, mainly due to the trust of cloud users assumption.
This Section presents a security analysis of existing PoW schemes.

A. Security Analysis

Despite the significant resource saving advantages, PoW schemes bring several security challenges that may lead to sensitive data.

- **Data confidentiality disclosure** – *hash-as-a-proof* schemes (e.g. Dropbox) introduce an important data confidentiality concern, mainly due to the static proof client side generation. For instance, if a malicious user has the short hash value of an outsourced data file, he could fool the storage server as an owner trying to upload the requested data file. Then, he gains access to data, by presenting the hash proof. As such, an efficient PoW scheme requires the use of unpredictable values of verifications.
- **Privacy violation** – sensitive data leakage is a critical challenge that has not been addressed by Halevi et al. in [8]. That is, cloud users should have an efficient way to ensure that remote servers are unable to access outsourced data or to build user profiles.
- **Poison attack** – when a data file D is encrypted on the client side, relying on a randomly chosen encryption key, the cloud server is unable to verify consistency between the uploaded file and the proof value $hash$. In fact, given a pair $(hash_D, Enc_k(D))$, the storage server cannot verify, if there is an original data file D, that provides a hash value $hash$. As such, a malicious user can replace a valid enciphered file with a poisoned file. So, a subsequent user looses his original copy of file, while retrieving the poisoned version.

B. Related Works

In 2002, Douceur et al. [4] studied the problem of deduplication in multi-tenant environment. The authors proposed the use of the convergent encryption, i.e., deriving keys from the hash of plaintext. Then, Storer et al. [13] pointed out some security problems, and presented a security model for secure data deduplication. However, these two protocols focus on server-side deduplication and do not consider data leakage settings, against malicious users.

In order to prevent private data leakage, Halevi et al. [8] proposed the concept of *Proof of Ownership* (PoW), while introducing three different constructions, in terms of security and performances. These schemes involve the server challenging the client to present valid sibling paths for a subset of a Merkle tree leaves [11].
The first scheme applies erasure coding on the content of the original file. This encoded version is the input for construction of the Merkle tree. The second purpose pre-possesses the data file with a universal hash function instead of erasure coding. The third construction is the most practical approach. Halevi et al. design an efficient hash family, under several security assumptions. Unfortunately, the proof assumes that the data file is sampled from a particular type of distribution. In addition, this construction is given in random oracle model, where SHA256 is considered as a random function.

Recently, Ng et al. [12] propose a PoW scheme over encrypted data. That is, the file is divided into fixed-size blocks, where each block has a unique commitment. The hash-tree proof is then built, using the data commitments. Hence, the owner has to prove the possession of a data chunk of a precise commitment, with no need to reveal any secret information. However, this scheme introduces a high computation cost, as requiring generation of all commitments, in every challenging proof request.
In [3], the authors presented an efficient PoW scheme. They use the projection of the file into selected bit-position as a proof of ownership. The main disadvantage of this construction is the privacy violation against honest but curious storage server. In 2013, Jia et al. [16] address the confidentiality preservation concern in cross-user client side deduplication of encrypted data files. They used the convergent encryption approach, for providing deduplication under a weak leakage model. Unfortunately, their paper does not support a malicious storage server adversary.

C. Threat Model

For designing a secure client-side deduplication scheme, we consider two adversaries: malicious cloud user and *honest but curious* cloud server.

- *malicious user adversary* – the objective of a malicious user is to convince the cloud server that he is a legitimate data owner. That is, we suppose that the adversary successes to gain knowledge of an arbitrary part of D. This information is then used as a challenging input to the POW protocol.
- *curious cloud server adversary* – this storage server honestly performs the operations defined by our proposed scheme, but it may actively attempt to gain the knowledge of the outsourced sensitive data. In addition, he may try to build links between user profiles and accessed data files.

III. System Model

Figure 1 illustrates a descriptive network architecture for cloud storage. It relies on the following entities for the good management of client data:

- *Cloud Service Provider (CSP)*: a CSP has significant resources to govern distributed cloud storage servers and to manage its database servers. It also provides virtual infrastructure to host application services. These services can be used by the client to manage his data stored in the cloud servers.
- *Client*: a client makes use of provider's resources to store, retrieve and share data with multiple users. A client can be either an individual or an enterprise.
- *Users*: the users are able to access the content stored in the cloud, depending on their access rights which are authorizations granted by the client, like the rights to read, write or re-store the modified data in the cloud. These access rights serve to specify several groups of users. Each group is characterized by an identifier ID_G and a set of access rights.

In practice, the CSP provides a web interface for the client to store data into a set of cloud servers, which are running in a cooperated and distributed manner. In addition, the web interface is used by the users to retrieve, modify and re-store data from the cloud, depending on their access rights. Moreover, the CSP relies on database servers to map client identities to their stored data identifiers and group identifiers.

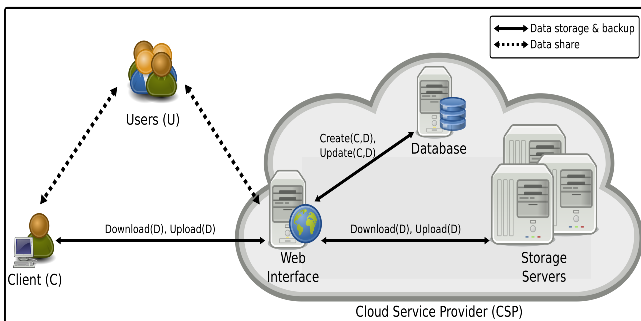

Fig. 1: Architecture of cloud data storage

IV. New Interactive Proof of Ownership Scheme

Our secure client-side data deduplication scheme is based on an original use of the convergent encryption [15].
That is, on one hand, when a data owner wants to store a new enciphered data file in remote storage servers, he has first to generate the enciphering key. This data encrypting key is derived by applying a one way hash function on data content.
After successfully encrypting the file data, the client has to generate the data identifier of enciphered data, in order to check its uniqueness in cloud database, before uploading the claimed file. This data identifier is computed by using a Merkle hash tree, over encrypted contents.
Then, for subsequent data outsourcing, the client is not required to send the same encrypted data. However, he has to substitute a client-server interactive proof scheme (PoW), in order to prove his ownership [7].

On the other hand, to protect data in public cloud servers from unauthorized entities, the client has to ensure that only authorized users are able to obtain the decrypting keys. As such, the data owner has to encrypt the data deciphering key, using the public key of the recipient user. This key is, then, integrated by the data owner in user metadata, ensuring data confidentiality against malicious users, as well as flexible access control policies.

To illustrate our solution for improving data security and efficiency, we first present the different prerequisites and assumptions. Then, we introduce three use cases for storing, retrieving and sharing data among a group of users.

A. Assumptions

Our solution considers the following assumptions.
First, we assume that there is an established secure channel between the client and the CSP. This secure channel supports mutual authentication and data confidentiality and integrity. Hence, after successfully authenticating with the CSP, these cloud users share the same resources in a multi-tenant environment.
Second, our solution uses the hash functions in the generation of the enciphering data keys. Hence, we assume that these cryptographic functions are strongly collision resistant, as it is an intractable problem to find the same output for different data files.

B. Prerequisities

- *Merkle Hash Tree* – a Merkle tree MT provides a succinct commitment, called the root value of the Merkle tree, to a data file. That is, the file is divided into blocks, called tree leaves, grouped in pairs and hashed using a collision resistant hash function. The hash values are then grouped in pairs and the process is repeated until the construction of the root value.
  The Merkle tree proof protocol requires that the prover has the original data file. That is, the verifier chooses a number of leaf indexes and asks the verifier to provide the corresponding leaves. As such, the verifier has to send these leaves with a sibling valid path.

- *Interactive Proof System [7]* – this proof system is an interactive game between two parties: a challenger and a verifier that interact in a common input, satisfying the correctness properties (i.e. completeness and soundness).

In the following, we introduce our client-side deduplication construction, based on three different scenarios: storage, backup and sharing schemes.

C. Cloud Data Storage

When a client wants to store a new data file f in the cloud, he derives the enciphering key k_f from the data contents, based on a one-way hash function $H()$. Note that data are stored enciphered in cloud servers, based on a symmetric algorithm. Hence, the data owner has to encipher the data file that he intends to outsource. Then, he generates the data identifier MT_f. That is, it is the *Merkle Tree* over encrypted data. This identifier, associated to the file, must be unique in the CSP database. Thus, the client starts the storage process by sending a *ClientRequestVerif* message to verify the uniqueness of the generated MT_f to his CSP.

1) New Data File Storage: The storage process consists in exchanging the four following messages:

- *ClientRequestVerif*: this first message contains the generated data identifier MT_f, associated to a nonce n. Note that the nonce is used to prevent from replay attack or potential capture of the data identifier. This message is a request for the verification of the uniqueness of the MT_f. The CSP replies with a *ResponseVerif* message to validate or unvalidate the claimed identifier. Note that if the sent identifier exists, the client has to perform a subsequent upload extra-proof procedure with the provider (cf, Section IV-C2). Once the verification holds, the cloud server asks the client to send only the access rights of authorized users.
- *ResponseVerif*: this acknowledgement message is generated by the CSP to inform the client about the existence of the requested MT_f in its database.
- *ClientRequestStorage*: this message is sent by the client. If the file does not exist in the cloud servers, the client sends the file that he intends to store in the cloud, and the data decrypting key k_f enciphered with the public keys of authorized users. Then, the enciphered k_f is included in the meta data of the file and it serves as an access rights provision.
- *ResponseStorage*: this acknowledgement message, sent by the CSP, is used to confirm to the client the success of his data storage.

This message contains the Uniform Resource Identifier (URI) of the outsourced data.

2) Subsequent Data File Storage: When a client wants to store a previous outsourced data file, he sends the data file identifier MT_f to the cloud provider. Since the claimed identifier in cloud database, the cloud has to verify that the requesting entity is a legitimate client. That is, the subsequent data storage procedure include these four messages:

- *ClientRequestVerif*: a subsequent data owner includes in this first message the generated data identifier MT_f, associated to a nonce n, in order to check its uniqueness in cloud database.
- *OwnershipRequest*: this message is sent by the CSP, to verify the client's data ownership. It contains random leaves' indices of the associated Merkle tree of the requested file. Upon receiving this message, the client has to compute the associated sibling path of each leaf, based on the stored Merkle tree, in order to prove his ownership of the requested file.
- *ClientResponseOwnership*: in his response, the client must include a valid sibling path of each selected leaf. The CSP verifies the correctness of the paths provided by the client. We must note that this data subsequent storage process stops if the verification fails.
- *ResponseStorage*: if the data ownership verification holds, the CSP sends an acknowledgement, to confirm the success of storage, while including the URI of the requested data.

D. Cloud Data Backup

The data backup process starts when the client requests for retrieving the data previously stored in the cloud. The data backup process includes the following messages:

- *ClientRequestBackup*: it contains the URI of the requested data that the client wants to retrieve. Upon receiving this client request, the CSP verifies the client ownership of the claimed file and generates a *ResponseBackup* message.
- *ResponseBackup*: in his response, the CSP includes the encrypted outsourced data $k_f(f)$.
  Upon receiving the *ResponseBackup* message, the client first retrieve the file metadata and deciphers the data decrypting key k_f, using his secret key. Then, he uses the derived key to decrypt the request data file.

E. Cloud Data Sharing

We consider the data sharing process, where the client outsources his data to the cloud and authorizes a group of users to access the data. Next, we refer to these user(s) as the recipient(s) and to the data owner as the depositor.

We must note that our proposal does not require the recipients to be connected during the sharing process. Indeed, recipients' access rights are granted by the data owner and managed by the CSP. That is, these access rights are also included in the meta data file. In addition, the CSP is in charge of verifying each recipient access permissions before sending him the outsourced data.

In practice, each recipient is assumed to know the URI of the outsourced data. This URI distribution problem can be solved in two ways. Either the depositor sends the URI to the recipient as soon as he stores data or a proxy is in charge of distributing the URIs. Once the depositor stored the data with the authorized access rights of the group, each member of the group can start the data sharing process based on the two following messages:

- *UserRequestAccess*: This message contains the URI of the requested file. When receiving this message, the CSP searches for the read/write permissions of the recipient, and then, he generates a *Response Access* message.
- *ResponseAccess*: the CSP includes, in its response, the enciphered file $k_f(f)$. Upon receiving this message, each recipient retrieves the data decrypting key from user metadata. That is, he deciphers the associated symmetric key with his own private key. Then, he performs a symmetric decryption algorithm to retrieve the plaintext. Our proposal provides a strong solution to improve the confidentiality of data in the cloud. In addition, the access to outsourced data is controlled by two processes. First, there is a traditional access list managed by the CSP. Second, the client has to own the private decrypting key to get the secret needed to retrieve the symmetric key compulsory needed to decipher data.

V. Security Discussion

In this section, we present a brief security discussion of our proposal. In addition, we expose its possible refinements to mitigate other threats.

- *Data confidentiality* – in our model, we propose to outsource encrypted data to remote storage servers. That is, data is stored enciphered in the cloud, based on a symmetric encryption algorithm using a per data key. This enciphering key is a content related information, ensuring data deduplication in remote servers. Thus, the confidentiality of outsourced data is twofold.
  First, we ensure confidentiality preservation against malicious users. On one hand, when a user wants to store new data in the cloud, he has to send the data identifier MT_f, based on the encrypted file. Then, he substitutes a nonce, while sending its *ClientRequestVerif*. Hence, this dual data identifier protection provides better secrecy to data outsourcing issue. On the other hand, if an attacker succeeds to retrieve the data identifier, he has to prove his ownership, based on a random challenge. We must note that our ownership proofs rely on the well-known Merkle tree lemma [11]. This lemma says that every prover that passes the Merkle tree protocol with high enough probability can be converted into an extractor that extracts mos of the leaves of the tree.
  Second, we enhance data confidentiality against curious servers. That is, the data owner outsource encrypted contents. Then, he enciphers the decrypting key relying on an asymmetric scheme, in order to ensure efficient access control. As such, the CSP is also unable to learn the contents of stored data in his his public servers.
- *Privacy* – based on a cryptographic solution to keep data content secret, sensitive information are generally included in metadata whose leakage is a critical concern in a multi-tenant environment. Thus, our model mitigates to such privacy violation issue.
  On one side, the CSP identifies clients as data owners, while outsourcing the same content in remote servers. However, the cloud server cannot bind the consistency between the plaintext information and these data owners, as he has only access to hashed identifiers and

encrypted contents. Consequently, he is unable to build user profiles, based on the received identifiers.
On the other side, searching for outsourced data may also endanger the privacy. That is, generally the backup process is based on keywords search. Indeed, our solution replaces the usage of keywords by the use of data identifiers which are derived by a Merkle tree hash computation over encrypted data.

- *Access control* – our client-side deduplication proposal provides forward and backward secrecy of outsourced data. It authorizes recipients to have access to data, based on their granted privileges, with respect to their private keys. That is, when a user wants to wants to access data, he has first to authenticate with the provider. That is to say, the access to data has been already strictly controlled by an authentication phase, before the verification of authorizations granted by the owner. In addition, even though a curious server or a malicious user can gain access to metadata, the enforcement of access contro is still safeguarded, since the decrypting key is enciphered using the public key of the authorized recipient.
  Besides, our scheme is well suited for the sharing process, as the client uses a different encryption key for each new data storage. As such, we avoid using the same key for enciphering all the outsourced data.

VI. A Swift-Client Data Deduplication Based System

In this section, we first present the context of our conducted experiments with OpenStack Object Storage, and then evaluate the system performances.

A. Context

In order to evaluate the performances of our proposal, we build a simulated cloud storage framework, based on OpenStack Storage system (Swift) [1]. Swift is a cloud based storage system, which stores data and allows write, read, and delete operations on them. To achieve security enhancement of Swift, we extend its functionalities with algorithms and protocols designed in our scheme.
We have designed our own architecture, performing an installation of swift. Indeed, our architecture consists in dividing the machine drive into four physical volumes. Then, each volume is divided into four logical volumes. In total, we obtain sixteen partitions, each one represents one logical storage zone.

The simulation consists of two components: the client side and the cloud side. We implement several cryptographic algorithms based on cryptographic functions from the OpenSSL library [14], the GMP library [6] and the Pairing Based Cryptography (PBC) library [2], with independent native processes.

B. Implementation Results

In order to evaluate the performances at the client side, we first conduct data encryption and decryption tests locally. Second, we evaluated the time consumption of random data upload in the cloud, or data download from remote servers. For our tests, we used 1000 samples in order to get our average durations. In addition, we conducted our experiments on an Intel core 2 duo, started on *single mode*, where each core relies on 800 MHz clock frequency (CPU).

1) Client-Side Computation: To evaluate client side computation costs, we conduct data symmetric encryption and decryption, using a symmetric cryptographic algorithm. We choose Advanced Encryption Standard (AES) scheme as our enciphering algorithm and implement the *CBC mode* of AES.
Figure 2 shows the computation overhead of data encryption and decryption at swift-client side, with different sizes of data contents.

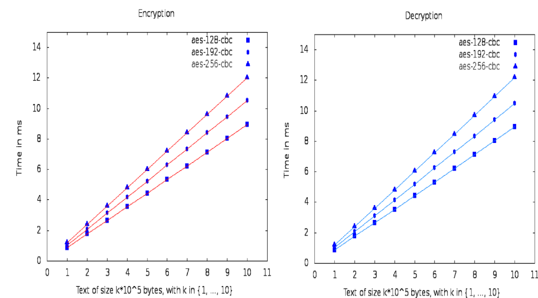

Fig. 2: Computation overhead of data encryption and decryption at the client side with different data size (from 10^5 to 10^6 bytes) (ms)

We can notice that the data encryption takes less than 12 ms, in order to encrypt $1MB$ data file. We can note that this computation cost remains attractive, as it provides better security to outsourced data and does not deserve the client resources.

2) Upload/Download : In this section, we evaluated the time consumption of random data upload in the cloud, or data download from remote servers. Our approach consists at first, to generate a random data file of a fixed size. Then, we encrypt the data file based on the AES-CBC algorithm. The length of the used enciphering key is 256 bits (AES-256-CBC). Afterwards, we upload the encrypted file and we download it from the cloud.

As such, we conducted some tests by choosing different files of different sizes. At each time, we computed the average time for uploading and downloading the encrypted file. Next, we present the results of average time computation to upload and download data file in the cloud.

TABLE I: Average time to upload and download file of size from 10 to 10^4 bytes, encrypted by AES-256-CBC

	Average Time in "s"		Standard Deviation σ	
Size in Bytes	Upload	Download	Upload	Download
10	0.338	0.193	0.231	0.067
10^2	0.329	0.192	0.210	0.060
10^3	0.339	0.189	0.233	0.027
10^4	0.326	0.194	0.191	0.080

By analyzing the results, we can conclude that:

- the time to upload a given data in the cloud is greater than the time to download it from remote servers.
- for data size less than 5×10^4 bits, the time needed to upload the file in the cloud (respectively download it

TABLE II: Average time to upload and download data file of size from 10^3 to 9×10^3 bytes, encrypted by AES-256-cbc

	Average Time in "s"		Standard Deviation σ	
Size in Bytes	Upload	Download	Upload	Download
1×10^3	0.340	0.190	0.230	0.021
2×10^3	0.328	0.190	0.202	0.035
3×10^3	0.340	0.189	0.260	0.018
4×10^3	0.333	0.191	0.235	0.059
5×10^3	0.333	0.193	0.260	0.127
6×10^3	0.324	0.188	0.200	0.020
7×10^3	0.337	0.188	0.241	0.026
8×10^3	0.325	0.191	0.176	0.026
9×10^3	0.328	0.192	0.194	0.025

TABLE III: Average time to upload and download data file of size from 10^6 to 9×10^7 bytes, encrypted by AES-256-CBC

	Average Time in "s"		Standard Deviation σ	
Size in Bytes	Upload	Download	Upload	Download
1×10^6	1.464	1.272	0.226	0.086
2×10^6	2.524	2.340	0.248	0.080
3×10^6	3.552	3.376	0.246	0.120
4×10^6	4.540	4.383	0.260	0.150
5×10^6	5.512	5.339	0.260	0.125
6×10^6	6.490	6.272	0.292	0.124
7×10^6	7.437	7.196	0.358	0.130
8×10^6	8.398	8.121	0.308	0.159
9×10^6	9.400	9.034	0.328	0.153
10^7	10.32	10.58	0.291	0.253

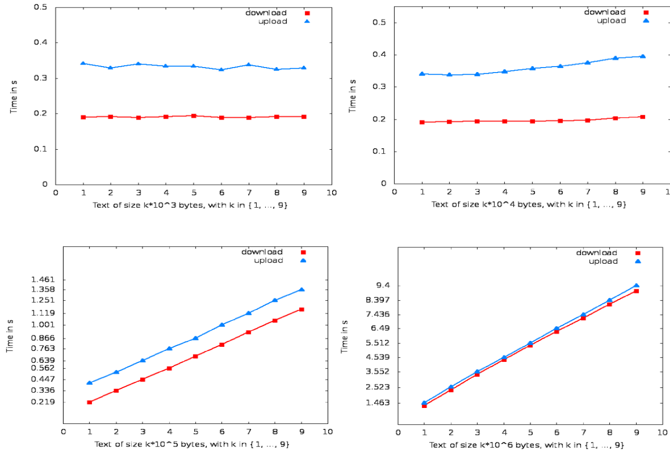

Fig. 3: Average time to upload and download file of different sizes

from the cloud) is almost constant and does not depend on the data size.

- for a given data size greater than 5×10^4 bits, the time needed to upload the file in the cloud (respectively download it from the cloud) increases by increasing the data size.

Finally, we conclude that the cryptographic operations, at the client side are acceptable compared to the upload operations and do not carry exhaustive computation capacities. For example, a $8 * 10^5$ bytes data size requires only 0.1 seconds to be enciphered, compared to 10 second be be uploaded. Therefore, the encryption procedures involve 1% from the OpenStack upload overhead.

3) Swift-Client Access Control Integration: In order to include the security procedures at the client side, we first choose an asymmetric algorithm based on the use of Elliptic Curve Cryptography (ECC) [9]. In order to use the Elliptic Curve Integrated Encryption Scheme (ECIES) asymmetric encryption scheme [9], we append encryption functions to upload swift command, in order to support encrypted data key in user metadata. As such, the cloud client has to apply the following command:

swift* -A** *http://ip:port/auth/v1.0* **-U** *account:login* **-K** *password* ***upload *container object* **-I** *ECIES_Encrypt*.

The *object* will be hashed using **SHA-256** and the result will be written in the file *object.AESkey*. Then, *object* will be encrypted using *AES-256-CBC* and the key in the file *object.AESkey*. After the generation of (public, private) keys using ECIES, we save the private and the public key. The key (in the file *object.AESkey*) will be encrypted using the ECIES encryption algorithm in the post method.

In order to support the proposed extension to the Swift environment, we add the following option to the post method options:

```
parser.add_option('-S','--AESKey',
dest='AESKey', help='To integrate the
encrypted AES key','the key length must
be : 32 bytes','AES key will be encrypted
by using the ECC encryption algorithm key')
```

The following code allows to integrate the encryption key in the meta-data after its encryption by using ECIES encryption algorithm:

```
if options.AESKey is not None:
  re_separateur=re.compile(r"[:]+")
  liste_0=re_separateur.split(options.meta[0])

  Input = open(liste_0[1], "r")
  while 1:
    ligne = Input.readline()
      if not ligne:
        break
      ecc_pkey = ligne
  Input.close()

  options.meta[0]=liste_0[0]+":"+ecc_pkey
  liste_1=re_separateur.split(options.meta[1])

  # Generate the curve
  crypto = ECIES();
  # Upload the AESKey from the file
  Input = open(options.AESKey, "r")
  while 1:
  ligne = Input.readline()
    if not ligne:
       break
    my_AES_key = ligne
  Input.close()

  # Encrypt my AES key by public key
  eccCurve = "p384"
  secrect_key = crypto.eccEncrypt(ecc_pkey,
  eccCurve, my_AES_key)
  S = base64.urlsafe_b64encode(secrect_key)
  options.meta[1] = liste_1[0]+":"+S
```

The command-line which allows to integrate the encrypted key in meta-data is:

***swift* -A** *http://ip:port/auth/v1.0* **-U** *account:login* **-K** *password* ***post* -m** *EcPkey:object.ECPkey*
-m *Enckey:"*****" container object* **-S** *object.AESkey.*

- -m: is an option to integrate some informations in the meta-data.
- EcPkey: is a static word meaning the elliptic curve public key.
- object.ECPkey: is a file containing the elliptic curve public key of the object.
- Enckey: is a static word meaning the encrypted key.
- "****": is a non-zero random string.
- -S: is an option for the AES key.
- object.AESkey: is the file containing the AES key (the output of SHA-256 applied to the object).

In our implementation, we have chosen *p384* as an elliptic curve [9]. The encryption key will be encrypted by the ECIES encryption algorithm and it will be encoded in 64 bit-basis.

In this part, we discuss the case when the recipient wants to retrieve data which are stored in the cloud by another user. In particular, when the data owner uses the upload and post methods to encrypt data and encipher the related data key (the result of data encryption is $data.enc$ and the encrypted key is $k.enc$). The recipient must obtain k.enc from the meta-data and then decrypt it using its private key. That is, the ECIES encryption algorithm uses the public key for the recipient to encrypt k.

We add three functions in swift code, in particular in the download method. The first one performs the ECIES decryption algorithm. The second uses the private key of the recipient. The third uses the deciphering data key.

```
parser.add_option('-I', '--EC_IES',
dest='ECIES_mode',help='To introduce the
elliptic curve mode :''the object is stored
crypted in the cloud, It will be decrypted'
'by using AES_256_cbc and the AES_key will
be decrypted by using''ECIES which is
Elliptic Curve Integrate Encryption Scheme')

parser.add_option('-S', '--EccSkey',
dest='EccSkey', help='To decrypt the AES_key,
which is encrypted by elliptic curve public
key using ECIES')

parser.add_option('-H', '--Hashkey',
dest='Hashkey', help='To use the Hash-Key,
which is the AES_key encrypted by using
elliptic curve public key, the Hash-Key exists
as information of Meda-data object')
```

The following code is used to decrypt the enciphering key $k.enc$:

```
if (options.ECIES_mode and options.EccSkey
and options.Hashkey):
  crypto = CryptoWrapper();
  Input = open(options.EccSkey, "r")
    while 1:
         ligne = Input.readline()
         if not ligne:
            break
         ecc_skey = ligne
   Input.close()
   encryptedMessage = options.Hashkey
   encryptedMessage = base64.urlsafe_b64decode
                       (encryptedMessage)
    eccCurve = "p384"
    my_AES_key = crypto.eccDecrypt
                (ecc_skey, eccCurve,
                 encryptedMessage)
    ciphertext = args[1]
    n = len(ciphertext)-4
    plaintext = ciphertext[:n]
    AESKey = plaintext+".AESkey"
    File = open(AESKey, "w")
    File.write("%s" %my_AES_key)
    File.close()
```

The following code is used to decrypt the outsourced data:

```
if (options.ECIES_mode and options.EccSkey
and options.Hashkey):
  ciphertext = args[1]
  n = len(ciphertext)-4
  plaintext = ciphertext[:n]
  os.system("openssl enc -AES-256-CBC -d -in
          "+ciphertext+" -out "+plaintext+"
          -k "+AESKey)
  os.system("rm "+ciphertext)
```

The following command-line, will be used to perform the data backup:

swift* -A** *http://ip:port/auth/v1.0* **-U** *account:login* **-K** *password* ***download *container object.enc* **-I** *ECIES* **-S** *object.ECSkey* **-H** *k.enc*

- download: is the download method for swift.
- object.enc: is the encrypted data stored in the cloud.
- -I: is an option to indicate the ECIES.
- ECIES: is the Elliptic Curve Integrated Encryption Scheme, used to call the ECIES decryption algorithm.
- -S: is an option to indicate the private key of the recipient.
- object.ECSkey: is a file which contain the private key of the recipient.
- -H: is an option to indicate the encrypted key stored in the meta-data.
- k.enc: is the encrypted key.

4) Swift-Client Sharing scenario Implementation: In this section, we present a cloud data sharing scenario between two users Alice and Bob. We suppose that Alice needs to receive data from Bob. So that, she sends her request (the data identifier and his public key) to Bob. When Bob receives the request, he prepares the data file. Then, he uploads the encrypted data as data.enc (encryption of data using the AES-256-CBC using a 256 bit size key) to the cloud servers. Bob performs this operation using the command-line:

swift* -A** *http://ip:port/auth/v1.0* **-U** *account:login* **-K** *password* ***upload *container object* **-I** *ECIES_Encrypt.*

After encrypting the key k using the ECIES encryption algorithm, Bob integrates k.enc in the meta-data of data.enc, this operation is performed by using the command-line:

swift* -A** *http://ip:port/auth/v1.0* **-U** *account:login* **-K** *password* ***post* -m** *EcPkey:object.ECPkey* **-m** *Enckey:"**" container object* **-S** *object.AESkey.*

Alice accesses the meta-data of data.enc, she recovers the encrypted key k.enc and decrypts it by using ECIES decryption algorithm. Then, she uploads data.enc and she encrypts it by using AES-CBC. To perform this operation, Alice uses the following command-line:

swift -A *http://ip:port/auth/v1.0* ***-U*** *account:login* ***-K*** *password* ***download*** *container object.enc* ***-I*** *ECIES* ***-S*** *object.ECSkey* ***-H*** *k.enc*.

In summary, our performance evaluation presents the efficiency of our proposal in content processing and delivery. At the client side, our scheme brings acceptable computation costs. The overhead of the implemented security mechanisms does not affect the client resources. At the cloud side, our deduplication proposal preserves the efficiency of content upload/download operations with a smaller overhead of cryptographic operations.

VII. Conclusion

The growing need for secure cloud storage services and the attractive properties of the convergent cryptography lead us to combine them, thus, defining an innovative solution to the data outsourcing security and efficiency issues.

Our solution is based on a cryptographic usage of symmetric encryption used for enciphering the data file and asymmetric encryption for meta data files, due to the highest sensibility of these information towards several intrusions. In addition, thanks to the *Merkle tree* properties, this proposal is shown to support data deduplication, as it employs an pre-verfication of data existence, in cloud servers, which is useful for saving bandwidth. Besides, our solution is also shown to be resistant to unauthorized access to data and to any data disclosure during sharing process, providing two levels of access control verification.

Finally, we believe that cloud data storage security is still full of challenges and of paramount importance, and many research problems remain to be identified.

Acknowledgements

This work is part of ODISEA project and is financially supported by the Conseil Regional d'Ile de France.

References

[1] https://github.com/openstack/swift.
[2] L. Ben. On the implementation of pairing-based cryptosystems, 2007.
[3] R. Di Pietro and A. Sorniotti. Boosting efficiency and security in proof of ownership for deduplication. In *Proceedings of the 7th ACM Symposium on Information, Computer and Communications Security*, ASIACCS '12, pages 81–82, New York, NY, USA, 2012. ACM.
[4] J. R. Douceur, A. Adya, W. J. Bolosky, D. Simon, and M. Theimer. Reclaiming space from duplicate files in a serverless distributed file system. In *In Proceedings of 22nd International Conference on Distributed Computing Systems (ICDCS*, 2002.
[5] M. Dutch. Understanding data deduplication ratios. SNIA White Paper, June 2008.
[6] T. G. et al. GNU multiple precision arithmetic library 4.1.2, December 2002.
[7] O. Goldreich. *Foundations of Cryptography: Basic Tools*. Cambridge University Press, New York, NY, USA, 2000.
[8] S. Halevi, D. Harnik, B. Pinkas, and A. Shulman-Peleg. Proofs of ownership in remote storage systems. In *Proceedings of the 18th ACM conference on Computer and communications security*, CCS '11, pages 491–500, New York, NY, USA, 2011. ACM.
[9] D. Hankerson, A. J. Menezes, and S. Vanstone. *Guide to Elliptic Curve Cryptography*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2003.
[10] D. Harnik, B. Pinkas, and A. Shulman-Peleg. Side channels in cloud services: Deduplication in cloud storage. *IEEE Security And Privacy*, 8(6):40–47, 2010.
[11] R. C. Merkle. A digital signature based on a conventional encryption function. In *A Conference on the Theory and Applications of Cryptographic Techniques on Advances in Cryptology*, CRYPTO '87, pages 369–378, London, UK, UK, 1988. Springer-Verlag.
[12] W. K. Ng, Y. Wen, and H. Zhu. Private data deduplication protocols in cloud storage. In *Proceedings of the 27th Annual ACM Symposium on Applied Computing*, SAC '12, pages 441–446, New York, NY, USA, 2012. ACM.
[13] M. W. Storer, K. Greenan, D. D. Long, and E. L. Miller. Secure data deduplication. In *Proceedings of the 4th ACM International Workshop on Storage Security and Survivability*, StorageSS '08, pages 1–10, New York, NY, USA, 2008. ACM.
[14] The OpenSSL Project. www.openssl.org, April 2003.
[15] C. Wang, Z. guang Qin, J. Peng, and J. Wang. A novel encryption scheme for data deduplication system. In *Communications, Circuits and Systems (ICCCAS), 2010 International Conference on*, pages 265–269, 2010.
[16] J. Xu, E.-C. Chang, and J. Zhou. Weak leakage-resilient client-side deduplication of encrypted data in cloud storage. In *Proceedings of the 8th ACM SIGSAC symposium on Information, computer and communications security*, ASIA CCS '13, pages 195–206, New York, NY, USA, 2013. ACM.