Towards Data Confidentiality and Portability in Cloud Storage

Ebtesam Ahmad Alomari and Muhammad Mostafa Monowar

Faculty of Computing and Information Technology,
King AbdulAziz University,
Jeddah, Saudi Arabia
ebtesam07@yahoo.com, mmonowar@kau.edu.sa

Abstract. As of now, *cloud computing* has become a hot topic in the global technology industry. Users become able to store their data in cloud storage and have ubiquitous access at any time. In spite of the enormous advantages of cloud storage, one of the greatest challenges is ensuring the security. In this paper, we address the problem of data confidentiality in cloud storage. Further, we consider the portability and secured file sharing issues in cloud storage. Our proposed solution consists of four different modules: *Encryption/Decryption provider (EDP*) that performs the cryptographic operations, *Third party auditor (TPA)* that traces and audits the EDP, *Keys storage provider (KSP)* which performs key management and *Data storage provider (DSP)* which stores user files in an encrypted form. We design a prototype to facilitate the process of secured data storage on DSP and KSP. The proposed mechanism ensures the data confidentiality, supports portability, and also provides secured sharing of files among users.

Keywords: Cloud Computing, Storage, Confidentiality, Portability.

1 Introduction

Cloud storage is a service that allows users storing files online in order to access them at any time via Internet. The most popular cloud storage options are Dropbox, Google Drive and SkyDrive. Most of them are free up to a certain number of gigabytes and provide uploading, downloading and simple sharing of files. Further, there are several benefits of using cloud storage, which are:

- Most cloud storage provide desktop folder to enable users managing their files easily.
- Cloud storage avoids the effort of emailing files to individuals.
- The data can be accessible from anywhere.
- It offers disaster recovery.
- It is economic, where there is no need for large storage maintenance and backup.
- In contrast, there are several disadvantages of using cloud storage, which are:

A. Marcus (Ed.): DUXU 2014, Part II, LNCS 8518, pp. 38–49, 2014.

- Using drag and drop to desktop storage folder will permanently move the file to cloud storage location.
- User cannot retrieve files if he does not have Internet connection.
- User needs to download the service on all devices if he wants to be able to handle file locally through multiple devices.
- Data security may be violated.

Among the pitfalls of cloud storage, ensuring security and confidentiality is considered as one of the most challenging problems. Users cannot depend on the security provided by cloud provider. In addition, users do not have any direct control on security and confidentiality of their data. Usually, the most common mechanism for ensuring data confidentiality is encryption. Two issues need to be considered for data encryption in the cloud: What is the best encryption algorithm and who is responsible for key management.

Devising a suitable algorithm for data encryption in the cloud requires considering the processing speed and computational efficiency since the cloud stores enormous data that need to be encrypted. In this regard, the researchers suggest for using symmetric encryption algorithm than the asymmetric ones [1]. Further, separate key management functionality is required since the user is not expert enough to manage the keys and the cloud provider cannot be trusted to provide the responsibility. Therefore, several researchers focused on providing a solution to enhance confidentiality, and protect data. However, these solutions still have several limitations.

Furthermore, there is a need for secure key management and file sharing approaches. In addition, the solution should consider the portability, which mean users should be able to encrypt/decrypt their data from any place and at any time.

This research will focus on these limitations and provide solutions to address them. The main contributions of this paper can be summarized as the following three aspects:

1. Propose an approach to provide data confidentiality on the cloud.
2. Introduce techniques to enable secure file upload, download, sharing and ensure that revoked user will not access the data.
3. Ensuring portability and does not restrict users to use single machine.

The organization of this paper is as follows; Section 2 discusses the related works, Section 3 presents the proposed mechanisms, Section 4 presents the implementation and finally Section 5 concludes the paper with future work.

2 Related Works

A number of state-of-the-art works address the data confidentiality issues for cloud storage. Puttaswamy et al. [2] proposed an approach called silverline that focuses on maintaining the confidentiality of the database on the cloud. The approach automatically recognizes the data that are not included in the computation. Then, those data will be encrypted before being stored in cloud storage. Further, it enables clients to store and use their keys safely while preventing cloud providers from stealing the keys. They suggest storing keys in an organization. Therefore, user will retrieve plaintext data from the cloud and keys from the organization. Then, the user is responsible for encrypting and decrypting the data in his side. However, the limitation in this approach is that not all data in the cloud are encrypted. In addition, even if they propose automatic partitioning of applications and moving the sensitive data, it increases computation in client machine.

Diallo et al. [3] proposed a policy-based privacy middleware called *CloudProtect* to enhance the protection of user data. They implemented it as an HTTP proxy server. This work focuses on two problems: how encrypted data can be stored in the cloud without changing the implementation of the application and how the functionality that needs to be in plaintext can be available to the users' even data is encrypted. However, the main limitation in their solution is that the privacy middleware is implemented as a desktop application that the user has to install it on a local machine. Then, he is enforced to use this machine whenever he needs to access his data because the application, the encryption/decryption keys, as well as the other information are stored on this machine along with the application. Although their implementation is easy to use, it is not very suitable for portability.

Saleh and Meinel [4] proposed an approach called *HPISecure* to achieve data confidentiality. The authors implemented it as an HTTP proxy that could be installed on the client's machine. They also evaluated it against Google Docs and Google Calendar. In addition, they suggested using *Facilitator* that could be the company that the user works for, or another cloud provider. The idea of using the facilitator is to store the cryptographic keys to solve the limitation in [4]. However, the drawbacks in *HPISecure* approach are it limits the sharing and collaboration of encrypted documents. In addition, it is implemented as a desktop application, which violates the concept of portability in the cloud.

Moreover, Huang et al. [5], provided a complete service called *SSTreasury+* to store and protect data in order to prevent them to be unauthorized, during the transmission to the cloud or in the cloud storage itself. Their system consists of three components: i) client-side application named *SSGuard* that is responsible for encryption and decryption, ii) a processing server called *SSManager* for storing keys, and iii) storage servers known as *SSCoffers* to store encrypted files. Furthermore, they considered that the decryption keys should be portable. However, the limitation in their solution is that the *SSGuard* application is a desktop application, which should be installed in each machine that is used to access their proposed system.

Hwang et al. [7] suggested separating the encryption and decryption system from the storage and application systems. Further, they suggested a third system called *CRM system* that is responsible to get a request from the user and send the data in plaintext to an *encryption/ decryption service* system for performing the encryption. However, the main drawback in their approach is that the data move in plaintext across the two service systems, which are *CRM* and *encryption/ decryption* service.

Table 1 summarizes the proposed mechanisms of the state-of-the-art approaches.

Table 1. Summarization of the Related Work

Solution	Focus	Encryption/ Decryption mechanism	Manage keys in other side	Support sharing data	Easy portability
Silverline [2]	Cloud Application data.	Encryption and decryption in client machine.	Yes	-	-
CloudProtect [3]	File that is editable online in the web.	HTTP proxy installed in client machine.	-	Yes	-
HPISecure [4]	File that is editable online in the web.	HTTP proxy installed in client machine.	Yes	-	-
SSTreasury+ [5]	File that is just stored in the cloud storage.	Using desktop application.	Yes	Yes	-
Encryption/ Decryption as an independent service [6]	File that is just stored in the cloud storage.	Using separate cloud service provider.	Yes	-	Yes

Most of the suggested solution does not provide easy portability. They also do not provide secure sharing of files among users. Further, most of the works consider that user should take the responsibility for Encryption/ Decryption.

Although the proposed mechanisms provide security, however these are inefficient in terms of portability and overhead on client side considering the cloud environment. This paper addresses the limitations of these researches to provide confidential, portable and secure approach in an efficient way.

3 Proposed Mechanisms

In this paper, we address the problem of confidentiality in cloud storage. We propose an approach to help users storing their data in cloud storage provider in an encrypted form to protect it from unauthorized access. The approach also considers portability and does not restrict users to use a single machine. Moreover, we propose a mechanism for secure sharing of files among users.

3.1 Basic Architecture and Preliminaries

Our proposed architecture consists of the following modules as shown in figure 1:

Encryption/ Decryption provider (EDP): It is responsible for key generation, encryption and decryption process. It should not store any data about user. It uses RSA algorithm to create the user's public and private keys. In addition, it generates random AES key, and use it to encrypt and decrypt user's files. The main benefits the module provides are:

- To enable user perform encryption/decryption operation from any machine and at any time (solve portability problem).
- To reduce load in client side (support thin client).

Third Party Auditor (TPA): It is responsible for tracing and auditing the EDP.
Data storage provider (DSP): It stores user files in an encrypted form.
Key storage provider (KSP): It is used to store the following information:

- The public key.
- Keys and information of user files that are stored in DSP. For each file in DSP, there is a file in KSP that contains its encryption/ decryption key and information like name of the file, size and the list of users that we share the file with them.

To enhance confidentiality, these services should be from different service providers. Further, after the completion of encryption or decryption operation, the EDP should remove all keys and data. All files in KSP will be in cipher text as they contain keys.

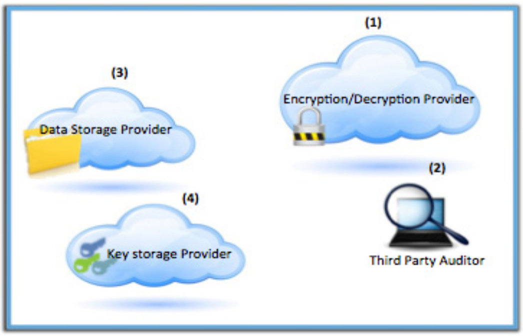

Fig. 1. Proposed modules

In this paper, we used several notations while describing the mechanisms. Table 2 summarizes the notations along with their meaning and usages.

Table 2. Definition and Notation

Notation	Definition	Used to/ stored in
F	User File	Stored in DSP
A_k	Random AES key	Used to encrypt/ decrypt F (user file).
E_k	User Public Key	Used to encrypt A_k (the Key of the file).
D_k	User Private key	Used to decrypt A_k (the Key of the file).
$Fi(A_k)$	File contains the A_k (Random AES key) and main information about F (User File).	Stored in KSP

Each F (user file) in DSP (data storage provider) should have Fi(Ak) in KSP (key storage provider). Fi(Ak) should contain symmetric key Ak, which is used to encrypt and decrypt F. There are two main advantages from generating new AES key for each F, which are:

1. Increase the security level: when the attacker compromises one key for a particular file, the other files will not be affected.
2. Provide secure sharing: when user A shares file F1 with user B, user B will know the key of F1 only, but he can not decrypt other files using this key.

3.2 Registration Phase

This section illustrates how the user registers an account for services. Figure 2 illustrates the steps, which are as follows:

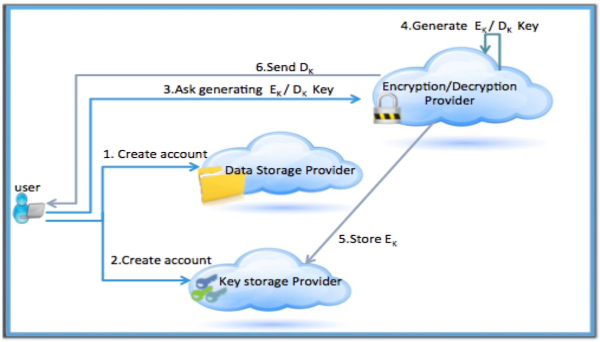

Fig. 2. Registration

1. User creates an account at DSP.
2. User creates an account at KSP.

3. User sends a request to EDP to generate keys.
4. EDP generates public and private keys.
5. The public key will be sent to store at KSP.
6. The private key will be sent to the user. The user should not lose the private key, as EDP will not store keys or any information about users.

3.3 Encryption Phase

In this phase, user files are stored in an encrypted form at DSP. Figure 3 shows the steps, which are as follows

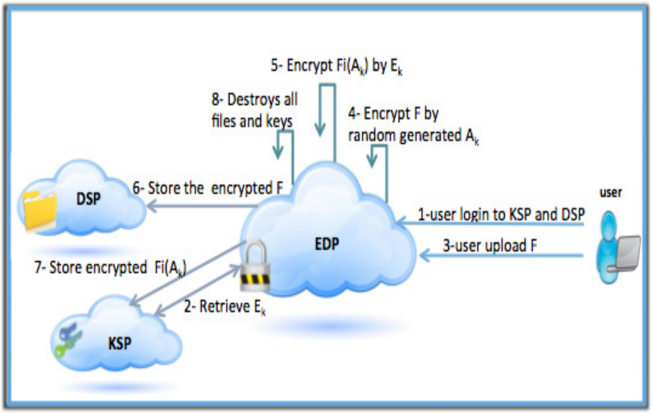

Fig. 3. Encryption

1. Initially, user login to KSP and DSP through EDP. In this case, there is a need to use OAuth authorization protocol, which enables a third-party application to get limited access to an HTTP service [7].
2. The EDP retrieves E_k (the public key) from KSP.
3. User uploads F (the file that he wants to store in cloud storage).
4. EDP generates A_k (random AES key). Then, the uploaded file is encrypted using $A_{k.}$
5. The uploaded file information (such as the size and type), and the corresponding A_k will be stored in one new file $Fi(A_k)$, and encrypted by user using E_k (public key).
6. EDP sends the F (the user encrypted file) to DSP.
7. EDP sends the encrypted $Fi(A_k)$ to KSP.
8. After the successful completion of the processes, the EDP should destroy all files and keys.

3.4 Decryption Phase

In this phase, the encrypted user file in DSP will be retrieved by the user after decryption process. Figure 4 illustrates the steps, which are as follows:

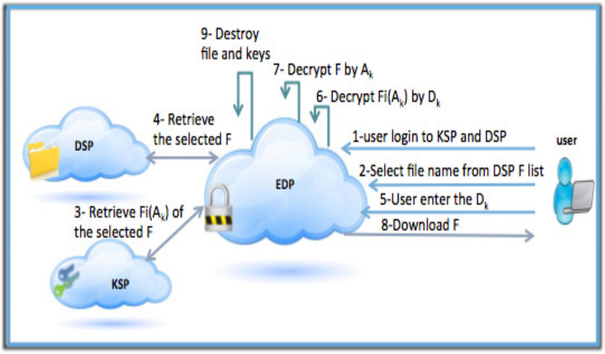

Fig. 4. Decryption

1. The user login to KSP and DSP through EDP. In this case, there is a need for OAuth authorization protocol.
2. User selects F, which is the file that he wants to download it from the DSP files list.
3. The EDP retrieves E_k (the public key) of this user and the Fi(A_k), which contains the information and key of F (the selected file).
4. The EDP retrieves F (the selected file) from the DSP.
5. Both the Fi(A_k) and F are in encrypted form. The EDP needs to decrypt the Fi(A_k) by the private key of the user. Hence, the user enters the D_k (his private key).
6. EDP uses the D_k to decrypt the Fi(A_k) to get the A_k (file's key) in plaintext.
7. Then EDP uses the A_k to decrypt F (the retrieved file from DSP).
8. Now the user can download the file in plaintext.
9. After the successful completion of the decryption process, the EDP destroys all files and keys.

3.5 Sharing Files

User A can give permission to user B to share a specific file by using EDP. In this case, user A should have the id (email) and public key of user B. Suppose that user A already have files in DSP, and know the public key of user B. Then, user A login to DSP and KSP through EDP. After login to DSP, the list of files will appear. Then user A can give permission to user B by performing the following steps as shown in figure 5:

1. User A selects the name of the file F that he wants to share it with user B.
2. Then, user A enters his own private key D_k and information of user B (public key and email).

3. EDP retrieves the public key of user A and the $Fi(A_k)$ of the selected file.
4. In EDP, the retrieved $Fi(A_k)$ should be decrypted by user A private key, D_k.
5. Then, EDP encrypts it using user B public key before sending the A_k to user B.
6. EDP informs user B that he can access the file by the A_k that was encrypted by his public key.
7. EDP should add the information of user B in the same file $Fi(A_k)$ that consist the key. Then, it stores again in KSP.
8. EDP should destroy all files and keys.

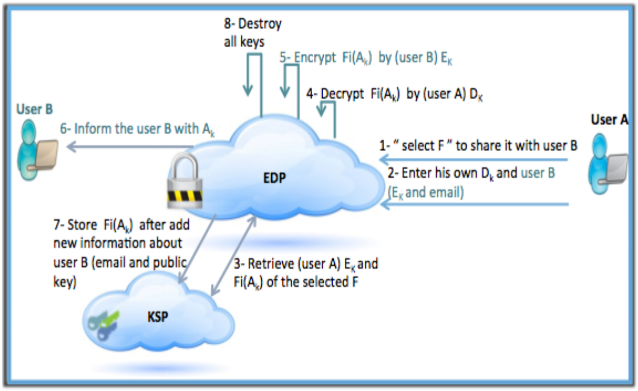

Fig. 5. Allowing a user to share file

Besides giving permission to share a file, there is also need for revoking a user from sharing the file. For that, user A needs to use EDP to re-encrypt the shared file by the new key so user B cannot decrypt it again by the old key. First, user A login to the DSP and KSP through EDP, and the list of files are appeared. Then user A can revoke permission from user B by performing the following steps as shown in figure 6:

1. User A selects the name of the file that he wants to prevent user B from accessing it.
2. User A enters his own private key D_k
3. EDP retrieves from the KSP both the public key of user A and the $Fi(A_k)$ of the selected file.
4. EDP also retrieves the selected files F from the DSP.
5. Then, EDP decrypts the $Fi(A_k)$ by user A private key, D_k.
6. EDP uses A_k (the key that is retrieved after decryption in step 5) to decrypt F (user file).
7. EDP generates new A_k to encrypt F and encrypt F by new A_k.
8. The encrypted file, F in step 7 will be stored in DSP to overwrite the previous file.
9. *Then, EDP also stores the new $Fi(A_k)$ at KSP, which contains the new key to overwrite the previous $Fi(A_k)$.*
10. EDP should destroy all files and keys.

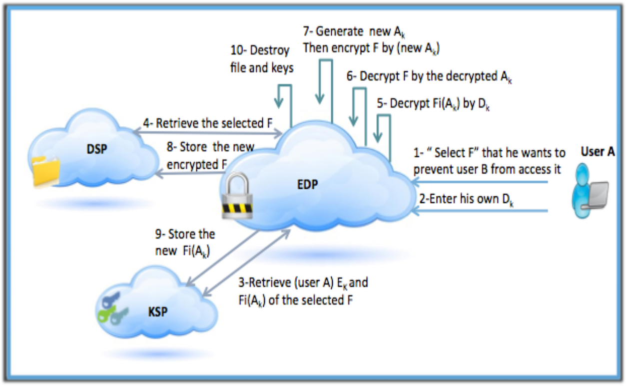

Fig. 6. Revoke user

3.6 Enhancing Security, Trust and Confidentiality

In this section we will discuss how our proposed approach enhances security, confidentiality and trust. We suggest keys to be stored in KSP in ciphertext format. Therefore, getting these data will not help the attacker even KSP itself, because the private key is with authorized user. Moreover, this approach presents secure sharing technique that prevents the revoked user from access to the file. The only module that can know the plaintext data and keys is EDP. Users should not depend only on reputation of service developer or company. Therefore, to enhance trust on EDP, we suggest government bodies setting policies and imposing penalties on breach of trust in the cloud. The EDP will be announced as one of the trusted service providers and hence certified if and only if it maintains the policies and applies the rules set by the respective bodies.

Furthermore, we suggest using a third party auditor (TPA) that will be responsible for tracing and auditing the EDP. It traces the log files and ensures that the EDP does not violate the policies. However, the TPA should not get access to plaintext data or keys. EDP should start logging all the transactions and access history of the files from the time it starts receiving the files and to the time it finishes destroying those. Then, the log files will be sent to the TPA, which verifies whether the EDP has fulfilled the agreement, or not. In the other side, EDP should use digital signature to enhance accountability and prevent log data from being compromised and modified by the attackers. Once TPA records a breach and finds a violation of policies, it immediately suggests for cancellation of the certificate or degrades the rating of the EDP to the respective bodies. Thus, we suggest to have a global or international security organization that maintains a database to store the rating from both TPA and customer side, in order to provide periodic ranking of the trusted EDP's.

4 Implementation

We develop a proof-of-concept to validate our approach. We select Google App Engine (GAE) to build a prototype of EDP using java. GAE enables developers to build an application easily that can run on Google's infrastructure. The application that we build is responsible for generating keys and encrypting/ decrypting files. Further, we need the EDP to interact with two storages to store the encrypted keys and files. Therefore, we select Dropbox and Google Drive because they offer APIs. Therefore, we can communicate with these storages to upload and download files from our java application. These APIs support using OAuth protocol for authentication and authorization. Further, when a login screens for these storages appear, user needs to authorize the EDP application with his Dropbox or Google drive account. Moreover, as shown in figure 7, Google Drive is used as data storage and Dropbox as key storage.

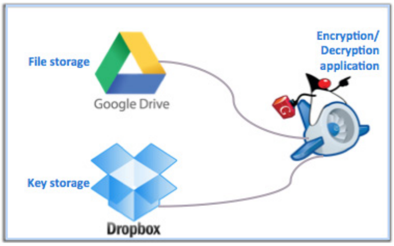

Fig. 7. Prototype implementation

In this paper, we implement the encryption phase only. First, the public and private keys are generated for the user. Then, a random AES key is generated and used to encrypt user text file. The encrypted text file is stored on Google Drive as shown in figure 8. Further, as shown in figure 9, AES key is stored on Dropbox after encrypted using user public key.

Fig. 8. User text file is encrypted by AES key and stored in Google Drive

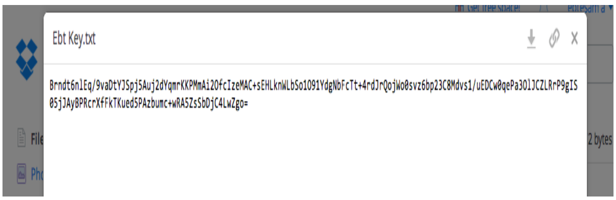

Fig. 9. AES key is encrypted by public key and stored in Dropbox

5 Conclusion and Future Work

Nowadays, ensuring confidentiality of data stored within the cloud storage is considered as one of the most challenging problems. Even though several state-of-the-art tried to solve this problem; they do not provide portability. Therefore, in this paper, we address the problem of confidentiality of data storage in the cloud that also supports portability. Furthermore, we also propose a mechanism for securely sharing of files among users.

We implement a prototype to prove the feasibility of the approach by developing Java application on Google App Engine (GAE). In addition, we use Google Drive and Dropbox to store user files and keys as ciphertext in separate storages. In future, we will try to address the confidentiality and portability for the files that are editable online in the web, such as Google Doc.

References

1. Chen, D., Zhao, H.: Data Security and Privacy Protection Issues in Cloud Computing. In: 2012 International Conference on Computer Science and Electronics Engineering (ICCSEE), vol. 1, pp. 647–651 (2012)
2. Puttaswamy, K.P.N., Kruegel, C., Zhao, B.Y.: Silverline: Toward Data Confidentiality in Storage-intensive Cloud Applications. In: Proceedings of the 2nd ACM Symposium on Cloud Computing, New York, NY, USA, pp. 10:1–10:13 (2011)
3. Diallo, M.H., Hore, B., Chang, E.-C., Mehrotra, S., Venkatasubramanian, N.: CloudProtect: Managing Data Privacy in Cloud Applications. In: 2012 IEEE 5th International Conference on Cloud Computing (CLOUD), pp. 303–310 (2012)
4. Saleh, E., Meinel, C.: HPISecure: Towards Data Confidentiality in Cloud Applications. In: 13th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid), pp. 605–609 (2013)
5. Huang, K.-Y., Luo, G.-H., Yuan, S.-M.: SSTreasury+: A Secure and Elastic Cloud Data Encryption System. In: 2012 Sixth International Conference on Genetic and Evolutionary Computing (ICGEC), pp. 518–521 (2012)
6. Hwang, J.-J., Chuang, H.-K., Hsu, Y.-C., Wu, C.-H.: A Business Model for Cloud Computing Based on a Separate Encryption and Decryption Service. In: 2011 International Conference on Information Science and Applications (ICISA), pp. 1–7 (2011)
7. Hammer-Lahav, E.: The oauth 1.0 protocol (2010)