Secure Storage as a Service in Multi-Cloud Environment

Riccardo Di Pietro[1(✉)], Marco Scarpa[2], Maurizio Giacobbe[2], and Antonio Puliafito[2]

[1] Department of Mathematics and Computer Science, University of Catania, Viale Andrea Doria, 6, 95125 Catania, CT, Italy
rdipietro@unict.it

[2] Department of Engineering, University of Messina, Contrada Di Dio, 98158 Sant'Agata, Messina, ME, Italy
{mscarpa,mgiacobbe,apuliafito}@unime.it

Abstract. Nowadays, Cloud computing is rapidly evolving due to social and cultural influences that are changing our everyday life. Innovative research in security-enabling techniques and architectures bodes to satisfy the increased interoperability among different vendors. An increasing number of service providers use Cloud computing to adapt their products to customer needs by addressing storage requirements in a secure way. The goal of our proposal is to provide a secure storage service able to store data in Multi-Cloud environment. The proposed secure storage architecture is oriented to guarantee confidentiality and integrity issues concerning information and data which are disseminated and stored in worldwide distributed Cloud environments. In this work, the SSME architecture is presented and discussed. Moreover, in order to evaluate our proposal, we conducted several experiments by considering the implementation of the SSME application as a Service in a real scenario.

Keywords: Cloud storage · Multi-Cloud · Confidentiality · Integrity · Security · AES-256 · RSA · OpenStack

1 Introduction

The growing number of requests to store and share files in Cloud environments results in an also growing number of user-friendly Cloud services in order to satisfy the above requests.

Cloud storage allows to store data in multiple remote sites usually owned by "top" companies and running their owner solutions, e.g., Google Drive, Dropbox, Amazon Simple Cloud Storage Service (S3). Besides the above-mentioned proprietary solutions, other open source solutions exist for providing Cloud storage services. For example, users can use the Swift OpenStack Object Storage to store lots of data efficiently and cheaply.

The development of the Cloud storage services marketplace particularly depends on the ability to build economies of scale. Within the *Digital Single*

Market Strategy, the *European Union* establishes a free flow of data in Europe, by facilitating data portability and switching of Cloud service providers. The study *"SMART 2013/0043 - Uptake of Cloud in Europe"* [4] indicates that Cloud developments could lead to the growth of the European Cloud market from €9.5bn in 2013 to €44.8bn by 2020 (i.e., almost five times the market size in 2013). Approximately the 19% of EU enterprises used Cloud computing in 2014, mostly did it for hosting their e-mail systems and storing files in electronic form. Moreover, further estimates of this study highlight that four out of ten companies (39%) using the Cloud reported the risk of a security breach was the main limiting factor in the use of Cloud computing services.

In such a scenario, by moving data to the Cloud, users can avoid (i) costs (i.e., money) to build and maintain a private storage infrastructure, (ii) unauthorized access by third parties, and at the same time it is important (iii) to reduce legal uncertainly. In fact, in view of easy file storage and sharing services widely accessible by several typologies of customers and contexts (e.g., businesses, academies, and many others), more and more personal and confidential data can be exposed to privacy and security vulnerabilities.

In this paper we present the **S**ecure **S**torage in **M**ulti-Cloud **E**nvironment **(SSME)** architecture which addresses the confidentiality and integrity issues concerning data store and dissemination in worldwide distributed Cloud environments. The architecture implements a novel solution which mainly uses encryption at client-side, and a worldwide distributed middleware for data splitting, dissemination and retrieval.

The reminder of this paper is organized as follow: in Sect. 3 we give a brief overview of existing work about the use of multi-Cloud storage using cryptographic data splitting. In Sect. 2, we describe the motivation about the solution we propose. The proposed architecture is described in Sect. 4, and a detailed explanation of the implementation is in Sect. 5. In Sect. 6, we describe the results of the performance analysis. Section 7 concludes the presented work, with a discussion of some improvements planned as future work.

2 Motivations

Cloud Computing Services usually provide resource management's capabilities which ensure security on accessing services and on communicating data, but they often lack of data protection from direct malicious accesses at system level. It is mandatory to guarantee data protection mechanisms in order to deny data intelligibility to unauthorized users, even when they are (local) system administrators.

The goal of our work is to provide a software mechanism capable to store data in multi-Cloud environment in a secure way, by giving an answer to the confidentiality and integrity issues of information and data disseminated and stored in remote distributed machines all around the world. With SSME, we intend to introduce the above-mentioned data protection mechanism by combining both symmetric (AES256) and asymmetric encryption (RSA). Moreover, users can use a dynamic management of both the fragmentation schema and the pool of Cloud storage services to use to create their own multi-Cloud environment whenever they want to store a file on the Cloud. In our proposal, all client-side configurations are totally invisible on the server-side. Each run of the SSME application is tracked nowhere on the system, neither on the physical disk of servers. By using the *Trusted Control Service* (TCS, see the Sect. 4.1), users are able to *integrate* Cloud services they trust for the significant Cloud computing functionalities of *authentication* and *backup-storage*. This greatly increases their confidence in using the SSME application.

Many works in literature deal with Cloud and multi-Cloud storage systems using cryptographic data splitting. In the next Section we present the background and how the already existing approaches and solutions differ from the SSME.

3 Related Work

In [2] the authors present an architecture for a cryptographic storage service. It consists of four components: a server which processes and encrypts (AES256) data before it is sent to Cloud, a private Cloud that holds the meta-data information, and two Clouds that respectively archive one half of each user's file. The authors assume that the remote server is trusted without specifying any information on "how" this trustiness is implemented. The meta-data information (e.g., passwords, secret keys of each files, encrypted access paths) are securely stored in the private Cloud. If compared with our dynamic approach which allows to specify the size of the split fragments, here data splitting is statically fixed on half of each user's file.

In [1] the authors propose a new method for securing the user's data using the multi-Clouds in an untrusted mobile Cloud environment. This method splits data into segments that are successively encrypted, compressed and distributed via multi-Clouds while keeping one segment on the mobile device memory. Keeping one segment in the user's device will prevent any attempt to recover the distributed data, thus to avoid the grabbing of all the segments together with the key by possible unauthorized users. In contrast to our approach, their solution requires a Mobile Cloud Computing (MCC) architecture and keeps a segment in each device.

In [9] the authors presents an architecture that makes the data splitting of the uploaded file size by three. Their architecture consists of a System Database and a Middleware for data slicing and merging, and for data encryption and decryption. Also in this work data splitting is statically fixed on a third of each user's file. Another difference is they use the System Database to store the information necessary for the Middleware operations, without to specify its content (i.e., the type of information).

In [8] the authors propose a reliable storage system, *TrustyDrive*, that takes care of both the document anonymity and the user anonymity. The system architecture consists of three layers: the end users which use a storage system as a service that splits and encodes the files; the dispatcher, which provides an entry point for both the end users and the Cloud providers; Cloud providers which provide different storage space in terms of costs and performances. At client side (the end user) documents are split in blocks and for each block meta-data are associated (user meta-data). The user breaks its meta-data into blocks to be sent to the dispatcher which, in turn, computes missing information to store meta-data blocks on Cloud providers. However, the authors do not provide a description about the secure communication between end user-dispatcher and dispatcher-Cloud provider. Moreover, differently from their architecture, our mechanism of splitting and dissemination of the fragments is configurable by the user, which is able to specify the size of the fragments and the pool of Cloud providers.

In [5] the authors present three different approaches for the parallel exploitation of different Cloud storage providers using Redundant Residue Number System (RRNS) in order to enforce long-term availability, obfuscation and encryption. They conduct several experiments considering a real testbed in which a client interacts with three different providers (i.e., Google Drive, Copy, Dropbox). In contrast to our approach, their solution permits user to retrieve files even if one of the Cloud storage providers previously used is not temporarily or permanently available. On the other hand, similarly to our approach, providers cannot access the files stored within them.

The TwinCloud client-side encryption solution presented in [3] focuses on the *secure sharing* on Clouds without explicit key management. To this end, they highlight the Public Key Infrastructure (PKI)-based solution problems, i.e., costs due to get a certificate from a Certification Authority (CA) and PKI-infrastructure maintenance. Differently from the TwinCloud solution, our architecture uses both symmetric and asymmetric cryptography.

All the papers do not give specific information on how the keys are managed, at least it is not described where the key are stored and how and who can access them. Moreover, in the above-mentioned contributions, none of virtual access points (e.g., gateway, dispatcher, server) to the various multi-Cloud environments which represent the main nodes of the processing offer a scheme easily and dynamically configurable by users. Neither adopts and/or describes the use of a secure communication protocol for its communications nor works only in volatile memory. It means that the processed data are not protected against insider attacks [6].

4 Proposed SSME Architecture

Figure 1 shows a general scheme of the SSME architecture. It consists of a JAVA client-server application which uses a stateless RESTfull approach for its communication, where a client cooperates with the middleware where most of the computation is done. The *SSME middleware* is, in turn, made up two main components: a "Trusted Cloud Service" (TCS) implementing all the fundamental interfacing functions with SSME clients, and a "Server" where all the middleware file manipulations are provided.

An SSME compliant application can work in two different modes: *Upload Mode* and *Download Mode*.

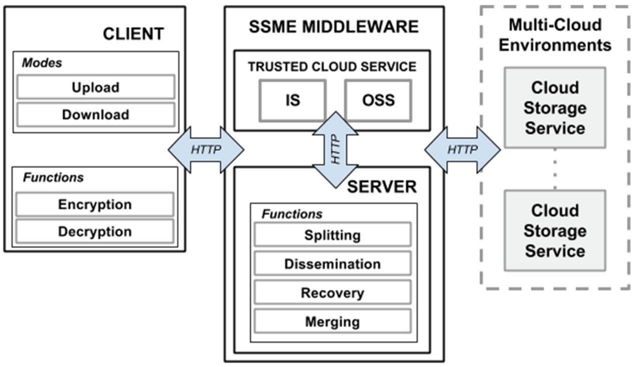


Fig. 1. SSME architecture general scheme.

4.1 SSME Middleware

Trusted Cloud Service. The **"Trusted Cloud Service" (TCS)** is the architectural component that provides some significant Cloud computing functionalities to the SSME client-server application. It is composed in turn into a two different sub-services: the **"Identity Service" (IS)** and the **"Object Storage Service" (OSS)**. As the name implies, the IS identifies the own trusted Identity Service the user may want to adopt by running the SSME client-server application. The IS, by exploiting its own token service mechanism, provides authentication and authorization on Cloud for the SSME service requests. In the same way, the OSS identifies the own trusted Object Storage Service that the user may want to adopt by using the SSME Application. The OSS provides temporary backup needed during the SSME internal operations. The OSS can also be used to store the encrypted JSON file that results as output of the SSME application. This output file represents the only *guiding light* to retrieve that particular file from the Cloud. We adopt the approach of externalizing the main function of Identity and Object Storage because each SSME user could have his/her own trusted Cloud services, for example personal or provided by his company, and may want to use these in the SSME scenario. In our testbed scenario, we adopt an OpenStack [7] compliant TCS.

SSME Server. All processing of the server are carried out without leaving any trace on local storage (hard disk) because it works only in volatile memory. In brief, the functions performed by the server are the following:

– uploading and downloading files to and from the TCS;
– fragmentation, recovery and merging of fragments to and from the Cloud services which are part of the multi-Cloud environment used at the moment;
– decryption of the information contained in the Headers of the HTTP requests received from the client;
– creation and encryption of the *json-encrypted-file* file which represents the output of the *Upload Mode*. This file represents the only way to recover the file from the Cloud.

4.2 SSME Client

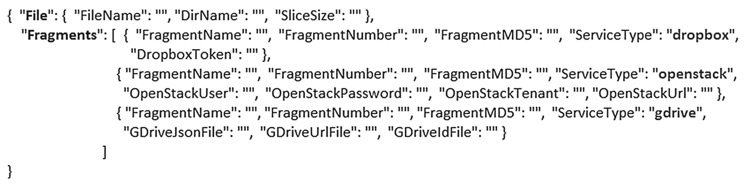
In order to work, the client needs the presence of a mandatory JSON configuration file called *json-conf-file* file. For smooth functioning of our service, the *json-conf-file* file must be filled in the proper way. As the name implies, the *json-conf-file* file contains the information about the configuration of the client, in particular:

– its internal settings (including the symmetric encryption key used);
– the communication with the Server (including the public key of the Server);
– the services that make up the TCS;
– the Cloud storage service providers which are part of the multi-Cloud environment the user want to use.

In brief, the functions performed by the client, when working in either *Upload Mode* or *Download Mode*, are the following:

– encryption of the file you want to send to the multi-Cloud environment;
– decryption of the file you want to receive from the multi-Cloud environment;
– encryption of the information contained in the Headers of the HTTP requests sent from both the server and the TCS;
– decryption of the information contained in the Headers of the HTTP requests received from both the server and the TCS.

Upload Mode. During the *Upload Mode*, the client reads the information inside the *json-conf-file* file in order to instruct itself on how to contact the service. According to the reported information, the client starts the processing. The client sends to the server some HTTP requests according to the rule described in Sect. 4.3. Once the server received all the needed information, it elaborates them and returns to the client an AES256 encrypted JSON file, we called *json-encrypted-file* file. The *json-encrypted-file* file contains all the relevant information in order to retrieve and rebuild all the fragments scattered among the Cloud storage services. Referring to the classical mythology, this file serves as a modern *Ariadne's thread*. In the absence of the *json-encrypted-file* file, or if it is damaged, it is not possible to recover and rebuild the fragments stored in the Cloud. The *Upload Mode* will return a *json-encrypted-file* file for each file uploaded in the Cloud. The template structure of the *json-encrypted-file* file is shown in Fig. 2.



```
{ "File": { "FileName": "", "DirName": "", "SliceSize": "" },
  "Fragments": [ { "FragmentName": "", "FragmentNumber": "", "FragmentMD5": "", "ServiceType": "dropbox",
                   "DropboxToken": "" },
                 { "FragmentName": "", "FragmentNumber": "", "FragmentMD5": "", "ServiceType": "openstack",
                   "OpenStackUser": "", "OpenStackPassword": "", "OpenStackTenant": "", "OpenStackUrl": "" },
                 { "FragmentName": "", "FragmentNumber": "", "FragmentMD5": "", "ServiceType": "gdrive",
                   "GDriveJsonFile": "", "GDriveUrlFile": "", "GDriveIdFile": "" }
               ]
}
```

Fig. 2. The *json-encrypted-file* file template structure.

The structure is composed of two main parts: the field "File" and "Fragments". The field "File" contains the name of the original file (i.e. FileName), the name of the container/directory where the fragments are stored inside the Cloud storage services (i.e. DirName) and the dimension of the single fragment (i.e. SliceSize). The field "Fragments" contains all the information needed to retrieve all the fragments. This field is structured as a JSON vector. In the actual implementation, the elements of this vector can assume three different typology: *dropbox*, *openstack*, *gdrive*. Each of these, represents the typology of the Cloud storage services used by our SSME testbed.

Download Mode. During the *Download Mode*, the client reads the information inside the *json-encrypted-file* file to follow the *Ariadne's thread*. The *json-encrypted-file* file is related to the original file that the client wants to retrieve from the Cloud. The client decrypts *json-encrypted-file* file by using the symmetric key specified in the *json-conf-file* file and extracts the information contained to process it. The client sends to the server all the information contained in the *json-encrypted-file* file according to the rule described in Sect. 4.3. These information are needed to the server in order to retrieve and rebuild all the fragments of the file the user want to download from its own multi-Cloud environment. These fragments were previously scattered among the Cloud storage services during the *Upload Mode*.

After the *Download Mode*, the client will obtain its original file (still encrypted). At this time, firstly the client deletes the *json-encrypted-file* file related to the file just receive, then in order to remove all the fragments stored in the multi-Cloud environment, the client sends some `delete-fragment` requests to the server. In the end the client will decrypt the original file.

4.3 Communication Between the Client and the Server

All the HTTP communications between the client and the server are authenticated using the token service provided by the chosen IS. In our testbed, we used the token service provided by the OpenStack Identity Service v2.0 (Keystone). Each HTTP request sent by the client embeds some fields that contain the information necessary to carry out a given task at the server side. All these fields are symmetrically encrypted by using the AES256 algorithm. The key to decrypt these fields is also embedded in each request inside a field called "key".

The client encrypts this field by using the RSA asymmetric algorithm with a key of 2048 bytes, by using the server's public key. The server, through its own private key, can decrypt the "key" field and using it to decrypt the content of all the other fields stored in the HTTP request body. The server uses the symmetric key received from the client to encrypt all the fragments before sending them to the Cloud storage services. It is worth noting that the public key of the server is known and is freely downloadable from the web.

5 Implementation Details

In this section deeper details on how client and server interact in the proposed architecture are presented. Using the communication protocol here described it is possible to design a client compliant with a public service based on SSME.

5.1 Client-Server Communication Protocol in "Upload Mode"

The communication phase between the client and the server is done in seven different steps: (see Fig. 3)

1. First of all, in order to verify if the server is alive, the client sends to it an HTTP request (**1.A**). If the test is successfully done (**1.B**), then the client can move to the next step (**2.A**).
2. In order to be authenticated by the "Trusted Control Service" (TCS), the client sends an authentication request to the "Identity Service" (IS) (**2.A**). If the IS returns the token (**2.B**), which means that it is successfully authenticated, then the client can move to the next step (**3.A**).

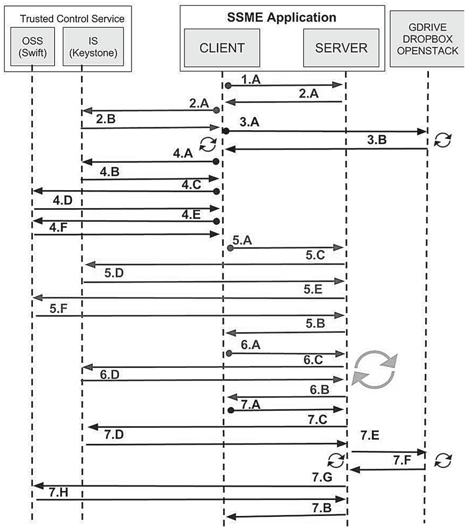


Fig. 3. Client-server communication protocol in "Upload Mode".

3. In order to test the validity of the Cloud storage services described in the *json-conf-file* file, the client checks each of them. The client sends an HTTP request to each of the Cloud storage services included in the list (**3.A**). If a Cloud storage service returns a positive response (**3.B**), the client sends another Cloud storage service test request (**3.A**), repeating the checking until the end of the storage service list.
4. In order to put the file to store in our multi-Cloud environment on the OSS:
   (a) firstly, the client sends an authentication request to the IS (**4.A**). If the IS returns the token (**4.B**), then the client can move to the next step (**4.C**);
   (b) then, the client sends an HTTP request to the OSS (**4.C**). If the container creation is successfully done (**4.D**), then the client can move to the next step (**4.E**);
   (c) finally, the client sends an HTTP request to the OSS **(4.E)**. If the the create-object request was successfully done (**4.F**), and the client can move to the next step (**5.A**).
5. To start the server-side processing phase, the client sends an HTTP request to the server (**5.A**). The server checks the received request by interrogating the TCS (**5.C**), and if the request is successfully authenticated (**5.D**), then the server starts the download of the file the user want to put on the multi-Cloud environment (**5.E**) from the OSS. When the server completes the download with success (**5.F**), it returns a response (**5.B**), and the client moves to the next step (**6.A**).
6. In order to transfer all the information about the Cloud storage services to the server, the client uses an HTTP request for each Cloud storage service (**6.A**). Each request contains the information about a particular Cloud storage service the client wants to transfer at that moment. The server checks the received request by interrogating the TCS (**6.C**) and, if the request is successfully authenticated (**6.D**), the server returns a positive response (**6.B)**, and the client sends another Cloud storage service request, repeating the process until the end of the Cloud storage service list.
7. To complete the server side processing phase, the client sends a `split` HTTP request to the server (**7.A**). The server checks the received request by interrogating the TCS (**7.C**) and, if the request is successfully authenticated **(7.D)**, the server starts the splitting phase of the file just downloaded from the OSS. After the splitting, the server sends each fragment to a different Cloud storage service by following a random mechanism (**7.E**). If the upload-request for a fragment is successfully done (**7.F**), the server sends another upload-request, and so on, until the end. When the server has sent all the fragments to the Cloud storage services, a `delete-fragment` HTTP request is sent to the OSS (**7.G**) and, if successful (**7.H**) the server moves to the next step (**7.B**). At the end, the server returns to the client the *json-encrypted-file* file (**7.B**).

5.2 Client-Server Communication Protocol in "Download Mode"

The communication phase between the client and the server is done in nine different steps (see Fig. 4):

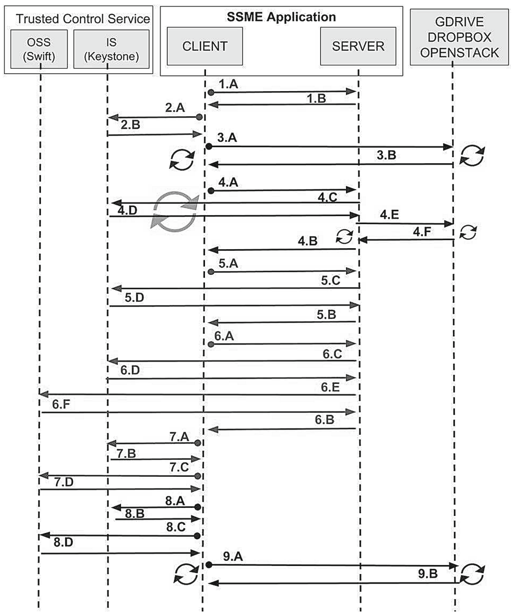


Fig. 4. Client-server communication protocol in "Download Mode".

1. First of all, in order to verify if the server is alive, the client sends to it an HTTP request **(1.A)**. If the test is successfully done (**1.B**), then the client can move to the next step (**2.A**) otherwise it completes.
2. In order to be authenticated by the "Trusted Control Service" (TCS), the client sends an authentication request to the "Identity Service" (IS). The client sends to IS an HTTP request with a JSON file containing its credentials (**2.A**). If the IS returns the token (**2.B**) the client is successfully authenticated and it can move to the next step (**3.A**).
3. In order to test if the Cloud storage services described in the *json-encrypted-file* file are active and available, the client checks each of them. It sends an HTTP request to each of the Cloud storage services included in the list. All these requests (**3.A**) contain the information about a particular Cloud storage service that the client wants to check. If the Cloud storage service returns a positive response (**3.B**), which means that the Cloud `storage-service-test` request was successfully done, the client sends another Cloud `storage-service-test` request, and so on, until the end of the list.
4. To start the server side processing phase, the client transfers to the server all the information about the fragments described in the *json-encrypted-file* file. The client uses one HTTP request for each fragment. All these requests contain encrypted parameters inside the headers (**4.A**). The server checks the received request by interrogating the TCS (**4.C**). If the request is successfully authenticated (**4.D**), the server uses the information contained in the request to download fragment from the related Cloud storage service (**4.E**). If the

Cloud storage service returns a positive response and a file (i.e. the fragment) (**4.F**), meaning that the Cloud storage service `download` request was successfully done, the server confirm the successful completion to the client (**4.B**). Then the client sends another `slice` request, and so on, until the end of the fragments and the client can move to the next step **(5.A)**.

5. To continue the server side processing phase, the client sends an HTTP request to the server (**5.A**). The server checks the received request by interrogating the TCS (**5.C**). If the request is successfully authenticated (**5.D**), the server start the merge operation of the fragments previously gathered server side. When the server successfully completes the merge operation, it returns a response (**5.B**).
6. To complete the recover of the file from the Cloud, the client sends an HTTP request to the server (**6.A**). The server checks the received request by interrogating the TCS (**6.C**). If the request is successfully authenticated (**6.D**), the server uploads the file just merged on the OSS ((**6.E**)). When the server completes the upload with success (**6.F**), it returns a response (**6.B**), and the client can move to the next step (**7.A**).
7. In order to download the file from the "Object Storage Service" (OSS):
   (a) firstly, the client sends an authentication request to the IS (**7.A**). If the IS returns the token **(7.B)**, the client can move to the next step (**7.C**);
   (b) then, the client sends an HTTP request to the OSS (**7.C**). If the the `download-object` request is successfully done (**7.D**), the client can move to the next step (**8.A**).
8. Once the client receives the file from the OSS:
   (a) firstly, the client sends an authentication request to the IS (**8.A**). If the IS returns the token (**8.B**), the client is successfully authenticated;
   (b) then, the client sends an HTTP request to the OSS (**8.C**). If the the `delete-object` request is successfully done (**8.D**), the server moves to the next step (**9.A**).
9. At the end, the client takes care to delete all the fragment scattered on the Cloud storage service constituting the multi-Cloud environment. The client sends an HTTP request to delete each of the fragments which are present in the *json-encrypted-file* file. All these requests (**9.A**) contain the information about a particular fragment the client wants to delete at that moment. If the Cloud storage service returns a positive response (**9.B**), the client sends another `delete-fragment` request, and so on, until the end. Then the client can finish.

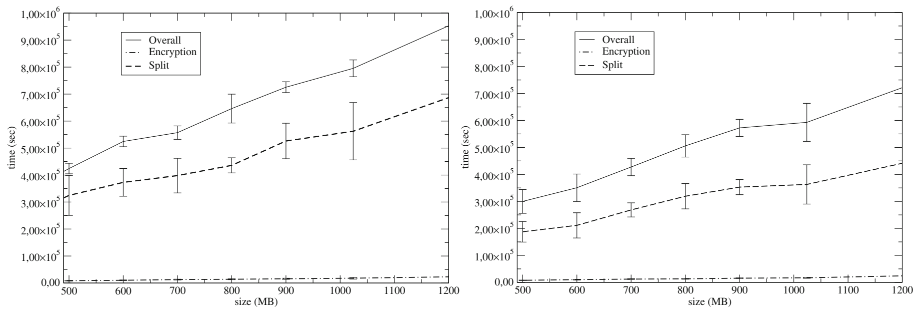
6 Performance Analysis

In order to evaluate the proposed system, we conducted several experiments by considering the implementation of the SSME application as a Service in a real scenario. The system was developed using the JAVA programming language for both client and server sides. We started the SSME application server on a virtual machine equipped with Ubuntu Server 14.04 and hosted on an IBM BladeCenter LS21. Instead, the client machine used in our experiments is equipped with the following hardware configuration: a CPU Intel(R) Core(TM) i7-4700MQ 2.4 GHz Dual-Core, 16 GB of central memory, Linux Ubuntu server 14.04.5 LTS 64 bit operating system and a SATA HD with 1 TB of disk storage. The middleware interacts with eight different Cloud storage services among three different Cloud storage providers: Google Drive (one instance), Dropbox (four instances) and OpenStack Swift (three instances). In our experiments, we considered different file sizes in each performed test: specifically, we used files from 10 MB to 2 GB. Therefore, we split each of them in fragments with two different size: 10 MB and 100 MB (this latter with large files at least 500 MB).

In *Upload Mode*, performance analysis consists in evaluating: the system response time (*Overall*), the encryption time (*Encryption*) and the time due to splitting and fragment dissemination (*Split*). In *Download Mode*, performance analysis consists in to evaluate: the system response time (*Overall*), the time to receive and merge all the fragments to recompose the original file (*Merge*), and the decryption time of the file just recomposed (*Decryption*).

We repeat each experiment 30 times and analyzed the collected data considering 95% as confidence level.

Figures 5 and 6 show a graphical representation of the monitored times, from 500 MB to 1 GB as file size, in *Upload Mode* and the *Download Mode* respectively. We did not depict the results obtained with lower file size only to make clearer the graphs without introducing any further information. As can be noted the system response time linearly increases with the file size both in *Upload Mode* and *Download Mode* and independently on the fragment size used for splitting the files. Both in *Upload Mode* and *Download Mode*, the impact of encryption and decryption, respectively, turn out to be negligible with respect the overall response time. When in *Download Mode* also the merge phase does not affect the performance, instead the splitting phase has an impact on the overall time in *Upload Mode*. In particular, in *Upload Mode* (Fig. 5)



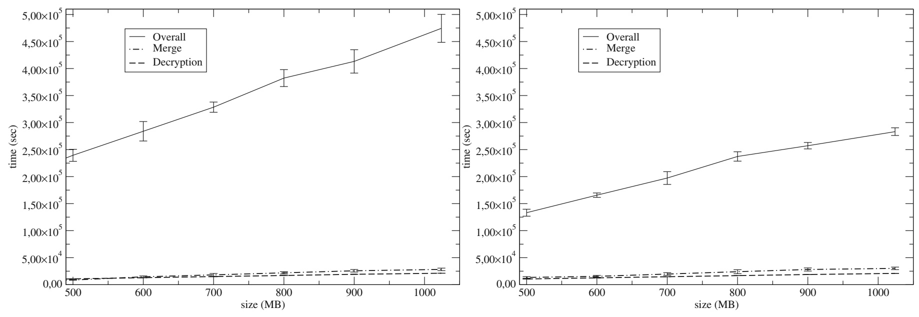
(a) Fragment Size = $10MB$ (b) Fragment Size = $100MB$

Fig. 5. Measured times in *Upload Mode*.

- encryption assumes a value between 1.5% and 5% of the overall time depending on the file size,
- Splitting and Fragments Dissemination is between 49% and 74% of the overall time depending on the file size.

In *Download Mode* (Fig. 6),

- Receive and Merging affects the Overall Time between 2.7% and 11% depending on the original file size.
- Decryption affects the Overall Time between 3.8% and 8% depending on the original file size.



(a) Fragment Size = $10MB$ (b) Fragment Size = $100MB$

Fig. 6. Measured times in *Download Mode*.

In both cases, the results show that the Overall Time improves in performance when greater fragment sizes are used. For example, in *Upload Mode*, this trend is confirmed for 800 MB file size: the average value is 646 s for 10 MB fragment sizes and 505 s for 100 MB fragment sizes. We had a similar trend in *Download Mode* using the same file: the average response time is 382 s when 10 MB fragment size is used and 237 s with 100 MB.

7 Conclusion and Future Work

This work presents and describes the SSME architecture to provide Secure Storage in multi-Cloud Environment. The architecture proposes to combine symmetric and asymmetric cryptography by offering a dynamic fragmentation schema to users which guarantees protection against insider attacks. In order to generate encrypted data while storing on multi-Cloud Environment, the symmetric cryptography is directly applied to the data on the client-side. The asymmetric cryptography (server public key) is used to encrypt sensitive information of HTTP requests Headers, which is being exchanged in the various steps described in the

Sect. 4.3. The SSME architecture has been developed and evaluated by considering the implementation of the SSME application as a Service in a real scenario. In future, we plan to release the SSME application as a public service and, based on it, to realize an Android Mobile Application and desktop application as SSME clients in order to show the effectiveness and the flexibility of our approach.

References

1. Alqahtani, H.S., Sant, P.: A multi-cloud approach for secure data storage on smart device. In: 2016 Sixth International Conference on Digital Information and Communication Technology and its Applications (DICTAP), pp. 63–69, July 2016
2. Balasaraswathi, V.R., Manikandan, S.: Enhanced security for multi-cloud storage using cryptographic data splitting with dynamic approach. In: 2014 IEEE International Conference on Advanced Communications, Control and Computing Technologies, pp. 1190–1194, May 2014
3. Bicakci, K., Yavuz, D.D., Gurkan, S.: Twincloud: a client-side encryption solution for secure sharing on clouds without explicit key management. CoRR abs/1606.04705 (2016)
4. Bradshaw, D., Cattaneo, G., Lifonti, R., Simcox, J.: Smart 2013/0043 - uptake of cloud in Europe. Technical report, IDC EMEA, June 2015
5. Celesti, A., Fazio, M., Villari, M., Puliafito, A.: Adding long-term availability, obfuscation, and encryption to multi-cloud storage systems. J. Netw. Comput. Appl. **59**, 208–218 (2016)
6. Gunasekhar, T., Rao, K.T., Basu, M.T.: Understanding insider attack problem and scope in cloud. In: 2015 International Conference on Circuits, Power and Computing Technologies (ICCPCT-2015), pp. 1–6, March 2015
7. OpenStack: Official web site. https://www.openstack.org/. Accessed 28 June 2017
8. Pottier, R., Menaud, J.M.: Trustydrive, a multi-cloud storage service that protects your privacy. In: 2016 IEEE 9th International Conference on Cloud Computing (CLOUD), pp. 937–940, June 2016
9. Vaidya, M.B., Nehe, S.: Data security using data slicing over storage clouds. In: 2015 International Conference on Information Processing (ICIP), pp. 322–325, December 2015