ESPRESSO: An Encryption as a Service for Cloud Storage Systems

Seungmin Kang[1], Bharadwaj Veeravalli[1], and Khin Mi Mi Aung[2]

[1] National University of Singpore, Singapore
{kang_seungmin,elebv}@nus.edu.sg
[2] Data Storage Institute, A*STAR, Singapore
mi_mi_aung@dsi.a-star.edu.sg

Abstract. Cloud storage systems have become the primary storage space for cloud users' data. Despite the huge advantages and flexibility of the cloud storage services, many challenges are hindering the migration of users' data into the cloud. Among them, the data privacy needs to be considered. In this paper, we design and implement an encryption service namely ESPRESSO (Encryption as a Service for Cloud Storage Systems) to protect the users' data by using advanced encryption algorithms. The flexible design and the standalone property of ESPRESSO allow cloud storage service providers to easily integrate it without heavy modification and implementation of their infrastructures. ESPRESSO was integrated into two open-source cloud storage platforms: OpenStack/Swift and Nimbus/Cumulus. The real experiments were conducted, and the results assess the performance and effectiveness of ESPRESSO.

Keywords: Cloud storage system, encryption, data privacy protection.

1 Introduction

With significant investment, many cloud storage systems are providing cloud users high data availability and the flexibility in data management, and they become the primary storage space for users' data. Thus, instead of storing and managing data in local servers, most of users nowadays are moving their data into the cloud and paying for storage and management service by a pay-per-use model. In this sense, cloud users are sharing a common storage space offered by Cloud Service Providers (CSPs). This characteristic raises several challenges which are hindering the migration of users' softwares and data into the cloud [1]. Among them, the security and data privacy are the most important challenges needed to be solved to attract users [2]. While consumers have been willing to trade privacy for the convenience of cloud storage services, this is not the case for enterprises and government organizations. This reluctance can be attributed to several factors that range from a desire to protect mission-critical data to regulatory obligations to preserve the confidentiality and integrity of data. The latter can occur when the customer is responsible for keeping personally identifiable information (PII), or financial and medical records [3]. Driven by the

© IFIP International Federation for Information Processing 2014

need to secure growing cloud data storage systems as well as high profile security breaches, data protection in cloud storage systems has become a hot topic in both academia and industry [4,5,6,7]. While the current approaches, which rely on a user-centric authentication service, can be broken by authentication attacks, encryption emerged as one of the most effective means to protect sensitive data no matter where it lives [8].

With an encryption tool, users can encrypt data on their local machine before uploading the encrypted data to a cloud. However, this approach introduces an additional burden for users to manage the encryption key and operate the encryption tool. Furthermore, users are required to equip local machines which are able to handle such a compute-intensive task that incurs the time overhead. These issues make the user-side encryption approach difficult to realize, especially when users are using scarce resource devices such as smartphones. A server-side encryption approach is therefore needed. On one hand, CSPs can provide the encryption to users as an added value service with minimal additional cost. On the other hand, this encryption can be offered as a free charge service. It then becomes a competitive advantage of a CSP against other CSPs to attract users and increase the CSP's reputation.

Among existing CSPs, only two commercial CSPs: Google Cloud Storage [9] and Amazon S3 [10] offer a server-side encryption service. However, the encryption services developed by Google and Amazon cannot be adopted by many other CSPs which want to offer the server-side encryption to users such as Microsoft Azure [11], GoGrid [12], RackSpace [13]. This observation inspires us to design and implement a standalone encryption service, ESPRESSO, for such CSPs to integrate without heavy modification and implementation of their infrastructures. Furthermore, we aim at providing a configurable and flexible encryption service for both CSPs who can choose the encryption algorithm based on their preference, and users who can specify the critical level of their data. The data with higher critical level needs to be more securely protected. Last but not least, we aim at providing ESPRESSO as a transparent encryption service which makes users perceive no difference between with and without the encryption service in terms of latency and complexity of data management operations.

The paper is organized as follows. Section 2 presents the problem statement including threat model and design goals of ESPRESSO whose the architecture is described in Section 3. Section 4 presents the detailed implementation of ESPRESSO. The integration of ESPRESSO into the Swift and Cumulus storage systems is presented in Section 5. Section 6 presents the experimental results which assess the performance and effectiveness of ESPRESSO. The related work is discussed in Section 7 followed by the conclusion of the paper in Section 8.

2 Problem Statement

2.1 The System and Threat Model

In this paper, we consider the CSPs which provide a data storage service. To protect the data privacy, an encryption service is used to encrypt the data before being stored in the cloud, and decrypt the data whenever users need to access the data. Depending on the deployment location of the encryption service, different threat models can be introduced. Below, three threat models will be analyzed.

1. The first model applies the user-side encryption approach. Users deploy the encryption software on their local machine and flexibly operate the service without needing to trust any third party. However, users are generally not expert in the security domain. The user's machine therefore suffers the security risks such as key exposure attacks or attacks from malicious programs. Moreover, it is not an easy task for non-expert users to take full responsibility of encryption key management such as key generation, key storage and keeping those keys always safe. Yet, if users are using scarce resource devices such as mobile devices, performing the encryption on such devices may not be possible since the encryption is considered as a compute-intensive task.
2. Users can rely on a third party who offers the encryption service. The third party takes full responsibility for managing the data encryption, protecting the encryption server and preventing the exposure of the users' encryption keys. Assuming that the encryption service is resistant to the security risks, users still have the sole concern on the adversarial behavior of the third party. With the curiosity and economic purpose, the third party might collude with malicious users to harvest data contents when it is highly beneficial [14]. Moreover, this model requires further effort from users to retrieve the encrypted data to their local machine before uploading to the cloud.
3. CSPs play the role of the third party presented in the second model. CSPs deploy the encryption software on a server in its trusted domain as one of its components. Users therefore benefit all advantages but also suffer the security risks as mentioned above. The operation overhead might be lesser since users do not need to manage the encrypted data. Instead, users upload the plaintext data to the CSP who will forward the data to the encryption server to encrypt before storing the encrypted data in storage servers.

As described, each model has advantages and disadvantages. Assuming that users trust the third party in the second model and the CSP in the third model at the same level, we believe that the third model brings users the most advantages. Depending on the model, the encryption service may be designed and implemented differently to assure that it efficiently operates at high performance. We present in the next section the design goals of ESPRESSO, the encryption service for CSPs as we advocate the third model presented above.

2.2 Design Goals

Several design requirements should be carefully considered since the design directly affects the overall performance of the system.

Architectural requirements. The encryption service should include two main components. The first component is the encryption key management. To increase isolation among users, a CSP may use different keys to encrypt different users' data and a user may have multiple keys for different data. To prevent leaking one's key to another, the encryption key must also be encrypted. Additionally, since the data availability is an important requirement of a cloud storage system, keys need to be replicated to be available when requested.

The second component is the data encryption management. ESPRESSO needs to provide the flexibility for both CSPs and users. Since ESPRESSO can be used by different CSPs, it should support multiple encryption algorithms. A CSP may choose its preferred algorithms to process users' data, e.g., Swift may use AES while Eucalyptus may use Blowfish. For users, the service should allow them to specify a desired critical level for their data. Currently, the CSPs, which offer server-side encryption, support only a single key length option, e.g., Google Cloud uses 128-bit keys. However, users may have different levels of security. Financial or medical records need to be more securely protected (using 256-bit keys) than entertainment data like musics or movies (using 128-bit keys).

Choosing supported encryption algorithms and critical levels of data. Given that CSPs offer different encryption algorithms and key lengths, choosing the supported encryption algorithms and critical levels of data is also important to achieve the flexibility. There exists many encryption algorithms in the literature including symmetric and asymmetric algorithms with their own advantages and disadvantages. A symmetric algorithm eases the implementation, however, it may not provide high level of security while an asymmetric algorithm is more complex to manage its key pair. Additionally, an asymmetric algorithm may take longer time for data encryption and decryption.

On the critical level of data, the longer key length is, the higher security level is guaranteed, however, it also takes longer time for encryption and decryption of data. Therefore, choosing the key length for each security level should take into account the tradeoff between the security level and the processing time.

APIs for integration to cloud storage platforms. As a last requirement of ESPRESSO, a well-designed integration API is also important since this allows CSPs to integrate and to use ESPRESSO easily without heavy modification of the architecture and implementation of their infrastructure. For instance, to provide an enhanced encryption service with a flexible critical level, the critical level should be one of API parameters along with data to be stored and user identification. Depending on the design, other parameters could be added. However, they should be carefully chosen since it may be difficult for CSPs to integrate ESPRESSO with redundant parameters.

3 System Architecture of ESPRESSO

In this section, we first present the detailed architecture and then describe the method to handle the flexibility and support multi-user scheme in ESPRESSO.

3.1 Architecture of ESPRESSO

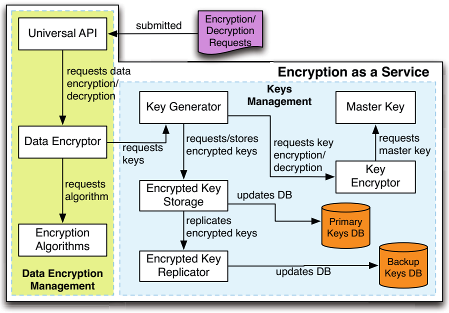


Fig. 1. ESPRESSO overall architecture

The overall architecture of ESPRESSO is depicted in Fig. 1 with two components: Data Encryption Management and Keys Management. The request flow is as follows. Universal API is the gate of ESPRESSO which can provide a wide range of interaction protocols allowing multiple CSPs to integrate ESPRESSO into their infrastructures. After receiving a request, Universal API delivers the request to Data Encryptor which is responsible for processing users' data using algorithms implemented in Encryption Algorithms. Data Encryptor requests encryption key from Keys Management through Key Generator which is the starting point of the Keys Management component. Key Generator retrieves the key stored in Encrypted Key Storage if it already exists, and sends it to Key Encryptor to decrypt using a master key. If the requested key does not exist in the database, that means the user is new on the system or the key for that specific critical level is not yet generated, Key Generator creates a new key and sends a key encryption request to Key Encryptor. The new key is then encrypted by the master key and sent back to Key Generator to store in Encrypted Key Storage. To assure the availability of encryption keys, encrypted keys are replicated and stored in Backup Keys DB.

3.2 Handling the Flexibility and Multi-user Scheme

To provide CSPs the flexibility in choosing a preferred encryption algorithm, ESPRESSO currently supports two algorithms: AES and Blowfish which are symmetric. By choosing symmetric algorithms, we eliminate the complexity of managing encryption key pairs which are supposed to be stored on different servers. Moreover, they are less intensive than asymmetric algorithms in terms of processing time. Additional algorithms can also be integrated into the system without breaking the architecture of ESPRESSO thanks to its agile design.

On the critical level of data, ESPRESSO provides three different critical levels by using three key lengths: 128, 192 and 256 bits for all supported encryption algorithms. The longer key length guarantees the higher critical level of data. A less than 128-bit key may be broken by the modern machine while a more than 256-bit key increases the latency of the service. Thus, each user can have up to three keys corresponding to three critical levels: the highest level uses 256-bit keys and the lowest level uses 128-bit keys, respectively. For a certain user, all the data with the same critical level are encrypted by the same key. Since a CSP

Table 1. Structure of the encryption keys table in MySQL

Field	**Type**	**Description**
key_id	Integer	Key identification: auto increment field
user_id	String	User identification
critical_level	Character	Critical level of user's data
key_string	String	Key string for encryption and decryption.

serves multiple users, each user is therefore identified by a user identification. We tie the user identification to the critical level and the encryption key by a tuple of $<$ user_id, critical_level, key_string $>$ in the encryption key database. Additionally, keys are generated on request of the CSP for a specific user and critical level.

4 Implementation of ESPRESSO

We use Python to implement ESPRESSO based on its broad adoption and efficiency. We implement in Universal API the Web Server Gateway Interface (WSGI) which allows CSPs to deploy ESPRESSO as a WSGI service. The implementation of Universal API handles the WSGI requests, i.e., extracting user_id, critical_level and data, and converts them to internal requests which are then forwarded to Data Encryptor. There are two functions in Data Encryptor: encrypt_data and decrypt_data. The encrypt_data function, which has three parameters: user_id, critical_level and data, prepares the encryption. It includes instructions for requesting the encryption key from Key Generator, initializing the encryption algorithm instance and finally invoking the execution of the encrypt_data function implemented in Encryption Algorithms. The algorithm selected by the CSP is saved in an INI configuration file with simple format, for example, [algorithm]name = AES. Instead of implementing all encryption algorithms by ourselves, we use a library namely PyCrypto [15] which provides the implementation of various algorithms such as AES, DES, RSA and ElGamal.

Since the critical_level parameter is needed to retrieve the encryption key for data decryption in the future; however, users may not remember which level was set for the data in the past, we include this parameter in the encrypted data. For a data retrieval request, the CSP gets the encrypted data from the storage server and passes it to ESPRESSO with the user_id parameter in a decryption request. Data Encryptor first extracts the critical_level parameter from the encrypted data and then invokes the decryption by calling the decrypt_data function.

To provide users a friendly manner to specify the data critical level, we decode three proposed critical levels by three letters: **A** stands for the high level, **B** stands for the medium level and **C** stands for the low level. Theses three symbolic letters hide the complex technical details of critical levels from users who are not expert in the security domain. The CSPs integrating ESPRESSO should provide a usage guideline to make their users aware of the trade-off between the strength and required processing time of each level, i.e., **A** is the strongest level but it requires longer time to complete the encryption.

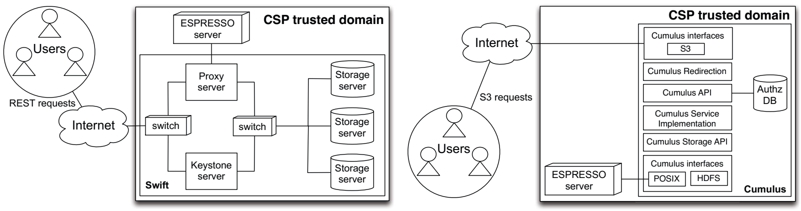
Algorithm 1 Encryption and Decryption calls

```
Input: data, user_id and critical_level for an encryption; encrypted_data and user_id for
    a decryption request; the ESPRESSO server address: server for both requests.
Output: Encrypted data for an encryption; plaintext data for a decryption request.
 1: connection = HTTPConnection(server); /*Create an HTTP/HTTPS connection*/
 2: connection.putrequest('EN', ''); /*'EN' for encryption and 'DE' for decryption*/
 3: for header in headers do /*Send all HTTP headers: user_id, critical_level*/
 4:     connection.putheader(header_name, header_value);
 5: end for
 6: for chunk in data do /*Send data by chunks*/
 7:     connection.send(chunk);
 8: end for
 9: response = connection.getresponse(); /*Waiting for response*/
10: Extract encrypted data or plaintext data from the response;
11: return
```

Encryption keys are generated by the Random library supported in Python. Each is a string including alphabet and numbers with length depending on the critical level. All keys are stored in a MySQL database whose the structure of the key table is shown in Table 1. Key Encryptor uses the same algorithm, i.e., AES or Blowfish, to encrypt the users' keys with a master key retrieved from Master Key. The implementation of Encrypted Key Storage and Encrypted Key Replicator handles the interaction with MySQL database, i.e., formulating the SQL query statements and executing the query.

5 Integration of ESPRESSO

We choose Swift [16] and Cumulus [17] to integrate ESPRESSO. These systems are open-source cloud platforms and they are widely used in both research community for experimental purpose and industry for commercial purpose. The integration involves determining a proper place in the source code of the storage systems where ESPRESSO is connected by using provided APIs and adding code instructions to realize that connection. The abstract pseudocode for encryption and decryption invocations from the storage systems is presented in Algorithm 1. Its detailed implementation depends on the target systems, programming language and supported library, e.g., Swift and Cumulus use Python while Eucalyptus uses Java. Generally, since ESPRESSO is implemented as a WSGI service, when a storage server requests for an encryption, a WSGI connection will be established (line 1). User's information and the data critical level are then passed by the connection header (lines 3 – 5). The data file is divided into chunks and sent to ESPRESSO (lines 6 – 8). When the data transmission is completed, ESPRESSO processes the data on its side while the storage server waits for the result (line 9) and continues the process after receiving data.



(a) With OpenStack/Swift. (b) With Nimbus/Cumulus.

Fig. 2. Integration of ESPRESSO into cloud storage platforms

5.1 Integration of ESPRESSO into Swift

The integration of ESPRESSO into Swift is presented in Fig. 2a where we add the ESPRESSO server as a novel component of the Swift platform. ESPRESSO is deployed on a separate server rather than becoming an internal component of Swift. This avoids breaking down the Swift's code structure. Since the encryption and decryption happen only when users have downloading, uploading or updating requests which correspond to GET and PUT methods in the RESTful API supported by Swift, all of modifications were made to the swift/proxy/controllers/obj.py module in the proxy server at two functions: GET(self, req) and PUT(self, req).

On the user's side, this integration does not complicate the data management operation. Only the uploading and updating requests require one more parameter to be added: the data critical level. For instance, if users use cURL [18] to interact with Swift for data management, the data critical level will be added as a novel header: `-H 'x-critical-level:A'`.

5.2 Integration of ESPRESSO into Cumulus

ESPRESSO has also been similarly integrated into the Cumulus storage system. The encryption and decryption invocations, presented in Algorithm 1 are added in the cumulus/cb/pycb/cbRequest.py module at two classes: cbGetObject(cbRequest) and cbPutObject(cbRequest). Like Swift, the total number of code lines added is less than 50 for both methods. This assesses the easy and light adoption of ESPRESSO in any cloud storage platform.

Since Cumulus supports the Amazon's S3 REST protocol, many client libraries and tools, including s3cmd [19], boto [20] and jets3t [21] can be leveraged by Cumulus users. For instance, if user uses s3cmd, a novel header will be added to specify the data critical level: `--add-header "critical-level: A"`.

With the integrated system, if users do not specify the critical level, ESPRESSO will automatically use the highest level, i.e., **A**, to encrypt the user's data.

6 Experiments and Performance Evaluation

6.1 Experiment Setup

The integrated storage systems were deployed using two dedicated physical servers on the same rack of the Communications & Networks Lab (CNL) at the National University of Singapore. Swift and Cumulus were installed on the server `xx.xx.xx.64` and ESPRESSO was installed on the server `xx.xx.xx.65`. The servers are PowerEdge C6220 with Intel(R) Xeon(R) Processor E5-2640 2.50GHz, 24GB RAM. We used real data files which are downloaded from the Wikipedia archive [22]. The file size varies from several MB to 4GB that allows us to evaluate the efficiency of the encryption algorithm with different loads. Three following performance metrics were considered for evaluation:

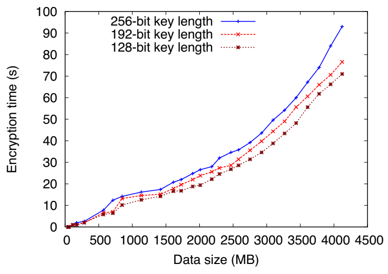
– Latency of encryption algorithms: To show the efficiency of encryption algorithms, we measured the encryption time with different key lengths for the same algorithm. In addition, we compared the encryption time of two different algorithms with the same key length.
– Latency of the integrated system with and without ESPRESSO: To show the transparency of ESPRESSO, the total operation time (i.e., sum of the data uploading time from the client to the storage server and the data encryption time) of Swift with and without ESPRESSO were compared.
– Impact of network bandwidth: In this experiment, a remote client which uses the Internet backbone for transferring data was deployed. Two different network connections: WiFi and wired connection were applied.

For each experiment, we performed 5 times to measure the average and standard deviation values of performance metrics. The second and third experiments were performed on both systems. However, due to the space limit and to avoid the redundancy, only results on Swift are shown. A comparison of total operation time between Swift and Cumulus is given in the analysis of the third experiment.

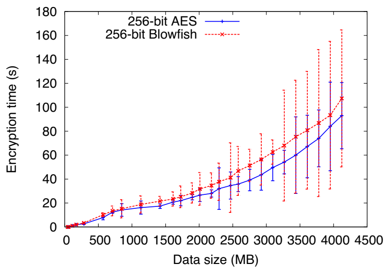
6.2 Performance Analysis

Evaluation of Encryption Algorithms. Fig. 3a presents the encryption latency of the AES algorithm with respect to the data size. We executed AES with three different key lengths: 128, 192 and 256 bits. It is expected that with the same key length, the larger data volume is, the longer time is needed to complete the encryption. With the largest file at 4GB, the encryption time with 256-bit key is 93 seconds. Comparing the latency of AES with three key lengths, it is trivial that the longer key needs longer time to complete, however, it generates a more robust encryption, i.e., the data is more securely protected.

We also measured the encryption time of Blowfish and observed that there is the same behavior as AES. In Fig. 3b, we present the encryption time of AES and Blowfish with respect to the data size and with the same key length, 256 bits. The results show that Blowfish needs a longer time to complete the encryption for the same data compared to that of AES. Indeed, since Blowfish uses a 64-bit

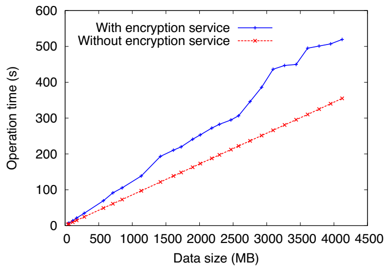


(a) Encryption time of AES

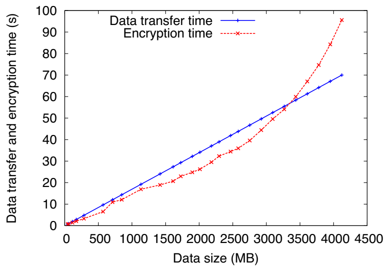


(b) AES vs. Blowfish.

Fig. 3. Performance of encryption algorithms



(a) With and without ESPRESSO.

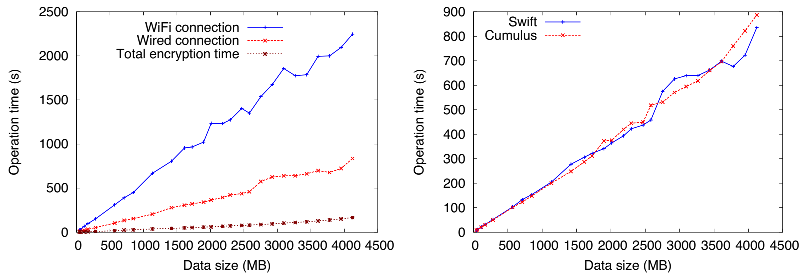


(b) Details of the encryption time.

Fig. 4. Uploading time with/without ESPRESSO and details of encryption time

block size while AES uses a 128-bit block size, the number of blocks processed by Blowfish is doubled compared to that of AES. The processing transition between blocks leads to the overhead of Blowfish. The results also show the nature of the encryption algorithms that the decryption time is almost the same as the encryption time as expected. Thus, to avoid the redundancy, we do not present the results on decryption time here.

Integrated System Validation. To validate the integrated system, we run the Swift client on a machine located in the same LAN to reduce the data transfer time between the client and Swift. The encryption algorithm is AES and the critical level is **A**. Fig. 4a depicts the total operation time for uploading requests of Swift with and without ESPRESSO. In the case without ESPRESSO, the total operation time can be considered as the data transfer time from the client to the Swift server. It is expected that the total operation time of Swift with ESPRESSO is longer than that without ESPRESSO since an additional time is needed for data encryption. This overhead includes data transfer time from Swift to ESPRESSO, the encryption time and the transfer time from ESPRESSO back to Swift for resulted data. In the worst case, the total operation time increases 63.95%. The details of the encryption time overhead are presented in Fig. 4b. While the data transfer time between the Swift and ESPRESSO servers is small and not affected by other users since the servers are installed on the same rack,



(a) Upload time from a distant client (b) Swift vs. Cumulus performance.

Fig. 5. Total uploading time from a distant client to Swift and Cumulus

the encryption time dominates when the file size is large, i.e., larger than 3.5GB. Even though we assume 4GB files as the worst case scenario, which roughly corresponds to the total content of a single-sided DVD, one may have larger files to store. However, the results show that it is strongly discouraged to store large files to not significantly degrade the performance of the system.

Impact of Network Bandwidth In practice, users are not always located nearby the cloud. Therefore, the data transfer time from the user's location to the cloud is much larger than that presented in previous experiment. Indeed, we did the third experiment by running the client machine locating 3 kms from the Swift/Cumulus servers, using the Internet backbone for transferring data. In Fig. 5a, we present the total operation time of Swift for uploading requests when the client uses the WiFi and wired connection. The average uploading speed is 1.54 Mbps and 6.72 Mbps, respectively. The results show that the data transfer time from the client to the Swift server dominates in both connections. With the largest file with the WiFi connection, the total operation time is 37.45 mins while the encryption time overhead is only 2.75 mins, corresponding to 7.34% of the total operation time. From the point of view of a user who is sensitive with the latency, he may still not accept such overhead. However, considering the security aspect that the user's data is securely protected by CSPs, we believe that the cost represented by the time overhead is worth for such a security service. Fig. 5b presents the comparison of operation time between Swift and Cumulus when the remote client uses wired connection. The operation times of both systems are almost the same. While Swift needs longer time for replicating the data with 3 copies, Cumulus does not provide the replication service. However, this overhead on Swift is compromised by the fluctuation of the data transfer time.

7 Related Work

Most of literatures on data encryption have focused on providing a user-side encryption tool which allows the owner to share his data with different consumers. [23] proposed an Identity-Based Authentication scheme by which the owner can share his encrypted data stored in the cloud. In [5], YI Cloud, a framework for protecting the data privacy in the cloud, is presented. The framework includes two components: a client component which is deployed on the user's machine for encryption and key management, and a server component installed on Sector [24] for management of users and storage nodes. Both [5] and [23] did not provide the flexibility for providers and users as ESPRESSO did.

In [25], a progressive encryption system has been proposed based on Elliptic Curve Cryptography. The system allows the owner to share his encrypted data with other consumers without revealing the plaintext data to untrusted entities. The work did not present any real experiment but we believe that this approach involves an intensive computation, thus introduces high latency. Furthermore, [25] focused on the encryption algorithm and the sharing mechanism while we aim at providing an entire encryption service which can be adopted by any existing CSP. Similarly, [14] proposed a secure and scalable fine-grained data access control scheme for cloud computing by combining attribute-based encryption with techniques of proxy re-encryption and lazy re-encryption. Both [14] and [25] considered a different threat model where users do not trust any third party such as cloud providers. Hence, users must take full responsibility for the data encryption and key management on their local machines.

In [7], the authors presented PasS (Privacy as a Service), a set of security protocols for ensuring the privacy of data stored in the cloud. Although presenting a server-side encryption service, this work assumed that the encryption service is maintained by a third party that is trusted by users as well as CSPs. Instead of trusting the third party, CSPs can integrate ESPRESSO as a security component in their infrastructures. Thus, they can increase the reputation and help cloud users alleviate the security concerns with the third party.

8 Conclusion and Future Work

We proposed ESPRESSO, a standalone and transparent encryption service for cloud storage systems. It provides CSPs the flexibility of choosing their preferred encryption algorithm by supporting two algorithms: AES and Blowfish. With the flexible design, CSPs can easily integrate ESPRESSO without heavy modification and implementation of their infrastructures. ESPRESSO provides users three data critical levels which allow users to specify an appropriate level of their data. The integrated system does not require much effort from users to make their data protected. The experiments on the Swift and Cumulus storage systems show that the introduced encryption latency is negligible compared to the total operation time of a data management request. All these advantages assess the effectiveness of ESPRESSO to be integrated into any CSP on the production level. The work can be extended to support data-owner and data-consumer schemes. The access control will be used to handle access permission and an encryption key for each consumer. While we focused only on protecting the backup data in this paper, a secured computation could be considered since the data stored in the cloud can be also used for further computation. However, tackling secured computation properly would require a paper by its self to do justice to the issues involved. Thus, we address this issue in a future paper.

Acknowledgements. This work is supported by A*STAR SERC from the project entitled "Towards Designing Flexible, Cost Effective, Secured Service Provisioning Strategies for Heterogeneous Data Centres in a Cloud-of-Clouds infrastructure" (Grant No. 112 280 4009), and A*STAR TSRP from the project entitled "Secured Large Scale Shared Storage System" (DSI/11-200006).

References

1. IMEX: The Promise & Challenges of Cloud Storage. Technical report, IMEX Research (August 2010)
2. Tian, L.Q., Lin, C., Ni, Y.: Evaluation of User Behavior Trust in Cloud Computing. In: ICCASM 2010, Taiyuan, pp. 567–572 (October 2010)
3. Kamara, S., Lauter, K.: Cryptographic Cloud Storage. In: Sion, R., Curtmola, R., Dietrich, S., Kiayias, A., Miret, J.M., Sako, K., Sebé, F. (eds.) FC 2010 Workshops. LNCS, vol. 6054, pp. 136–149. Springer, Heidelberg (2010)
4. Factor, M., Hadas, D., Hamama, A., Har'el, N., Kolodner, E., Kurmus, A., Shulman-Peleg, A., Sorniotti, A.: Secure logical isolation for multi-tenancy in cloud storage. In: IEEE MSST 2013, Long Beach, CA, pp. 1–5 (May 2013)
5. Huang, Z., Li, Q., Zheng, D., Chen, K., Li, X.: YI Cloud: Improving User Privacy with Secret Key Recovery in Cloud Storage. In: IEEE SOSE 2011, Irvine, CA, pp. 268–272 (December 2011)
6. Hao, L., Han, D.: The study and design on secure-cloud storage system. In: ICECE 2011, Yichang, China, pp. 5126–5129 (September 2011)
7. Itani, W., Kayssi, A., Chehab, A.: Privacy as a Service: Privacy-aware Data Storage and Processing in Cloud Computing Architectures. In: IEEE DASC 2009, Chengdu, China, pp. 711–716 (December 2009)
8. Harrin, E.: Cloud Storage Vendors Offering Encryption as a Service. Technical report, Enterprise Networking Planet (February 2012)
9. Google, `http://googlecloudplatform.blogspot.sg/2013/08/google-cloud-storage-now-provides.html`
10. Amazon S3, `http://docs.aws.amazon.com/AmazonS3/latest/dev/Welcome.html`
11. Microsoft Azure (April 2014), `http://www.windowsazure.com/en-us/`
12. GoGrid, `http://www.gogrid.com/`
13. RackSpace, `http://www.rackspace.com/`
14. Yu, S., Wang, C., Ren, K., Lou, W.: Achieving secure, scalable, and fine-grained data access control in cloud computing. In: IEEE INFOCOM 2010, San Diego, CA, pp. 1–9 (March 2010)
15. Litzenberger, D.C.: PyCrypto (April 2014), `https://www.dlitz.net/software/pycrypto`
16. OpenStack (April 2014), `http://swift.openstack.org/`
17. Bresnahan, J., Keahey, K., LaBissoniere, D., Freeman, T.: Cumulus: An Open Source Storage Cloud for Science. In: ScienceCloud 2011, CA, pp. 25–32 (June 2011)
18. cURL, `http://curl.haxx.se`
19. s3cmd (April 2014), `http://s3tools.org/s3cmd`
20. boto (April 2014), `http://code.google.com/p/boto`

21. jets3t (April 2014), `http://jets3t.s3.amazonaws.com`
22. Wikipedia: Wikipedia archive (February 2014), `http://dumps.wikipedia.org`
23. Kang, L., Zhang, X.: Identity-based Authentication in Cloud Storage Sharing. In: MINES 2010, Nanjing, China, pp. 851–855 (November 2010)
24. Gu, Y., Grossman, R.L.: Sector and Sphere: The Design and Implementation of a High Performance Data Cloud. Philosophical Transactions of The Royal Society A: Mathematical Physical and Engineering Sciences 367 (1897), 2429–2445 (2009)
25. Zhao, G., Rong, C., Li, J., Zhang, F., Tang, Y.: Trusted Data Sharing over Untrusted Cloud Storage Providers. In: IEEE CloudCom 2010, Indianapolis, IN, pp. 97–103 (November 2010)