Protecting Resources and Regulating Access in Cloud-based Object Storage

Enrico Bacis[1], Sabrina De Capitani di Vimercati[2], Sara Foresti[2], Stefano Paraboschi[1], Marco Rosa[1], and Pierangela Samarati[2]

[1] Università degli Studi di Bergamo, Italy
firstname.lastname@unibg.it
[2] Università degli Studi di Milano, Italy
firstname.lastname@unimi.it

Abstract. Cloud storage services offer a variety of benefits that make them extremely attractive for the management of large amounts of data. These services, however, raise some concerns related to the proper protection of data that, being stored on servers of third party cloud providers, are no more under the data owner control. The research and development community has addressed these concerns by proposing solutions where encryption is adopted not only for protecting data but also for regulating accesses. Depending on the trust assumption on the cloud provider offering the storage service, encryption can be applied at the server side, client side, or through an hybrid approach. The goal of this chapter is to survey these encryption-based solutions and to provide a description of some representative systems that adopt such solutions.

1 Introduction

The ever increasing availability of off-the-shelf cloud storage platforms has contributed to the growing of the *Storage-as-a-Service* (*SaaS*) market, with an increasing trend for users and companies to offload their (possibly sensitive or confidential) data and resources. There are several reasons for using cloud storage services such as the benefits in terms of availability, scalability, performance, and costs as well as the ability to easily share data with other users. However, this trend also introduces several security and privacy risks that can slow down the widespread adoption of storage services (e.g., [13, 18, 20]). In fact, by relying on third parties for the storage of their data and resources, users and companies lose the control over them: how can users and companies trust that their data are properly protected when stored on a third-party server? The research and development communities have dedicated many efforts in designing solutions for addressing this concern (e.g., [13]). Encryption is at the basis of many of these techniques: when data are encrypted they are visible only to the users who know the encryption key. Encryption has then been adopted not only as a valid solution for protecting data confidentiality (even against adversaries with access to the physical representation of the data, including the cloud providers themselves), but also for supporting selective sharing of such data [12]. In this case,

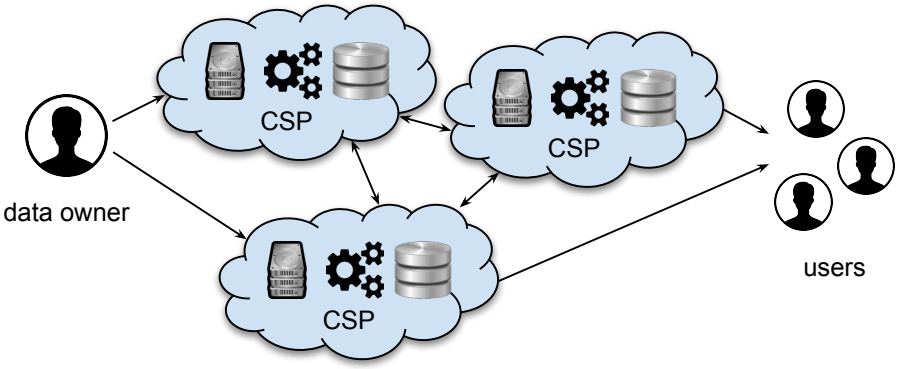

Fig. 1. Reference scenario

the idea consists in encrypting different portions of the data with different keys and then sharing the encryption keys with only the users that have the authorization for accessing the corresponding encrypted data. Figure 1 illustrates the typical reference scenario when considering cloud storage infrastructures. As it is visible from the figure, there are three main entities involved in this scenario: the *data owner* who wishes to outsource the management of her data to a third party, the *cloud providers* (CSPs) offering storage services, and other *users* who may need to access the data stored on cloud providers.

A fundamental aspect that needs to be considered when applying encryption to protect data is the *trust assumption* on the cloud providers in charge of storing and managing the data. Cloud providers can be *trusted*, *honest-but-curious*, or *lazy/malicious*. A trusted provider is fully trusted to access and manage the data that it stores. A honest-but-curious provider is trustworthy for providing services but should not be allowed to know the actual data content. A lazy or malicious provider is neither trusted nor trustworthy and therefore its behavior should be controlled. Depending on the trust assumption, encryption can be applied following three different strategies: *server-side*, *client-side*, *hybrid*. Server-side encryption means that the encryption of the data is managed directly by the cloud provider, which stores and manages also the encryption keys. In this case, the cloud provider guarantees that the data are stored in an encrypted format. However, whenever the cloud provider's services require direct visibility of the plaintext data for access execution, the provider can decrypt the data. Since the cloud provider has full visibility on the data, it can also enforce access restrictions. Server-side encryption can be applied only when the cloud provider is fully trusted. Client-side encryption means that users encrypt their data before storing them on external cloud providers. In this case, the encryption keys are stored and managed by the owner of the data and cloud providers cannot access the data in plaintext form, which limits the functionality that they can offer. Also, access control restrictions need to be enforced by

Trust Assumption \ Encryption type	*server-side*	*client-side*	*hybrid*
trusted	✓	✓	✓
honest-but-curious	×	✓	✓
lazy/malicious	×	✓	×

✓: applicable; ×: not applicable

Fig. 2. Encryption scenario depending on the trust assumption on cloud providers

the data owner who has to mediate all access requests to the data. This clearly reduces the advantages of outsourcing the management of data to a third party. Client-side encryption can be applied under any trust assumption on the cloud provider. However, it is usually adopted when the cloud providers are honest-but-curious or lazy/malicious. In the hybrid approach, the encryption of the data is performed both at the client-side and at the server-side with the consequence that there are two sets of encryption keys: one managed by the data owner and another one managed by the cloud provider. The rationale behind the hybrid scenario is that client-side encryption protects the data from cloud providers while server-side encryption efficiently enforces changes in the access control policy without the involvement of the data owner. Clearly, this approach can be applied only when cloud providers are honest-but-curious (or trusted) but cannot be applied when the cloud provider is lazy/malicious since there is no guarantee that the provider applies the required encryption operations. Figure 2 summarizes the applicability of the three encryption strategies according to the trust assumptions that characterize the cloud providers.

The goal of this chapter is to provide an overview of the current encryption-based solutions for protecting and enforcing selective access over data stored in the cloud. In particular, for each of the three encryption strategies discussed above, we first describe its salient aspects along with the main advantages and disadvantages. We then describe a representative system that applies the considered strategy. The remainder of this chapter is organized as follows. Section 2 focuses on server-side encryption and presents OpenStack as a representative system. Section 3 illustrates client-side encryption and describes the MEGA service. Section 4 shows the hybrid approach and describes a prototype system (ESCUDO-CLOUD EncSwift) protecting data confidentiality in OpenStack Swift. Finally, Section 5 presents future research directions and provides our conclusions.

2 Server-side Encryption

With server-side encryption, the cloud provider protects data in storage with an encryption layer that it can remove when needed to perform access and query execution (i.e., the cloud provider manages both the data and the encryption

keys). In this case, users placing data in the cloud have complete trust that the cloud provider will correctly manage the outsourced information.

2.1 Discussion

Being fully trusted, the management of data is completely delegated to the cloud provider itself. From the point of view of the users, the main advantage of this solution is that they can use all the functionality offered by the server for querying the outsourced data. Furthermore, the data owner can delegate the cloud provider to enforce access control policies for regulating access to data. From the point of view of the cloud provider, server-side encryption allows it to use *deduplication techniques* to avoid the storage of multiple copies of the same data, thus saving storage space. Basically, a cloud provider keeps the hash of every resource it is storing. When a user uploads a resource, the cloud provider computes the hash of the resources and checks whether the computed hash corresponds to the hash of a resource it already stores. If this is the case, the cloud provider discards the storage request and provides a link to the resource already stored.

Although many of the most well-know public cloud storage providers use server-side encryption (e.g., Dropbox, Amazon, and Google), this solution is not always feasible and introduces security risks. In fact, since the encryption keys are stored with the data, an adversary can exploit possible vulnerabilities of the cloud provider to obtain both the encrypted data and the encryption keys, thus obtaining the access to the plaintext version of the data themselves. Furthermore, the cloud provider might be forced by authorities to provide the stored data in their plaintext form. With respect to the data deduplication techniques commonly adopted by cloud providers, they can be exploited for violating data confidentiality. As an example, suppose that an adversary knows that a certain resource is stored on the cloud provider but does not know the value of some specific bytes (e.g., one value of a csv file). The adversary might try to generate as many resources as the possible combinations for the missing bytes and to upload each of them, one at a time. When the upload operation is not performed, the adversary knows that the uploaded file corresponds to the one already stored and therefore knows the value of the missing bytes. We note that these considerations apply to both public clouds and private clouds (i.e., cloud solutions built internally by a company).

Examples of public storage services based on server-side encryption are *Dropbox* [14], *Amazon Simple Storage Service (S3)*, and *Google Cloud Storage (GCS)*. All these services typically store the encryption keys in their proprietary key management system and mainly differ in the pricing schema. Although the companies ensures that no access is performed on users' data, they could potentially access all the data they store. In the following, we present OpenStack Swift as an example of cloud solution offering server-side encryption.

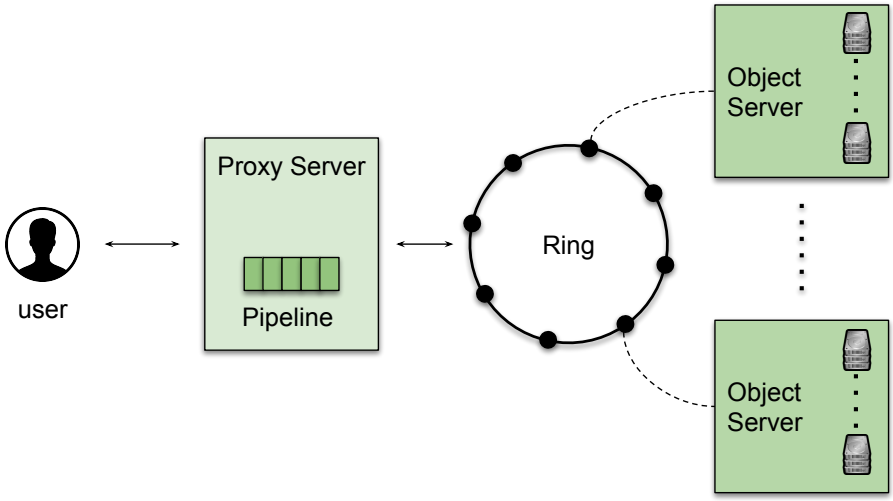

Fig. 3. OpenStack Swift architecture

2.2 Case Study: OpenStack Swift IBM Key Rotation

A well-known open source cloud computing platform that adopts server-side encryption is OpenStack (`http://www.openstack.org`). OpenStack manages large pools of computing, storage, and networking resources, all controlled by administrators through a dashboard. OpenStack consists of several components including an object storage system, called *Swift*. The architecture of Swift is composed of a *Proxy Server*, a *Ring*, and an *Object Server* (Figure 3). The Proxy Server is the key component of Swift and is responsible for processing requests coming from users and interacts with all other components. The Ring determines the physical device where a file should be located. In other words, it is responsible for mapping names and physical location of data. The Object Server is a blob storage (i.e., a storage that can manipulate unstructured data) in charge of storing, retrieving, and deleting objects on disks. Each object is stored as a binary file, and its metadata are stored as extended attributes of the file. Objects stored in Swift are organized in *containers*, which loosely corresponds to directories of common file systems. Containers are organized in *tenants* (or accounts). For interacting with Swift, a user sends a valid request to the Proxy Server. The request is then processed by a pipeline of middlewares, and each of them can enrich, filter, or drop metadata. In case the request reaches the end of the pipeline, it is dispatched to the relevant Object Server based on the information contained in the Ring. Once the request is processed by the Object Server, a response is sent to the user, processed again by the middlewares of the Proxy Server but in reverse order.

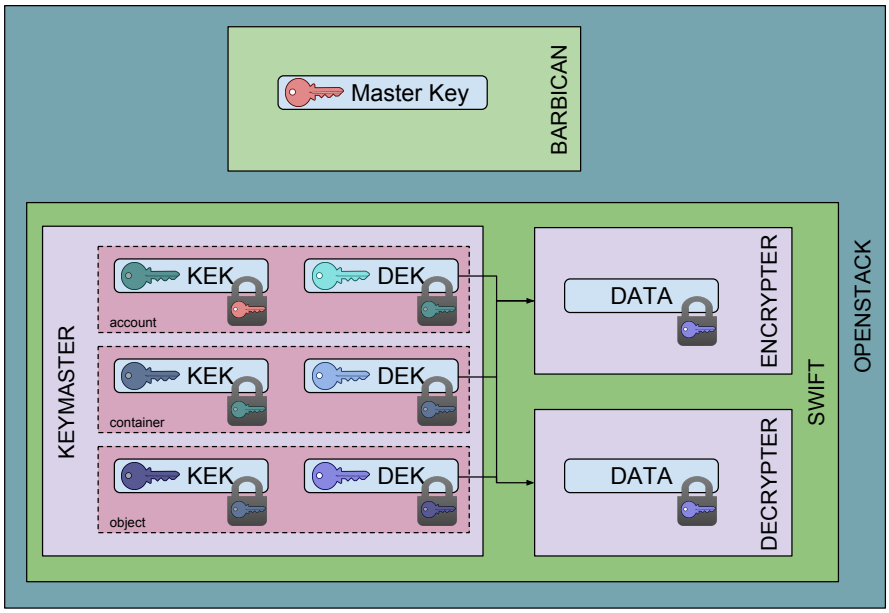

Fig. 4. Swift-KeyRotate: Key organization

One of the latest release of OpenStack Swift (Ocata[1]) supports server-side encryption to protect data at-rest (both objects content and metadata). To this purpose, three new middlewares have been added: *encrypter*, *decrypter*, and *keymaster*. Encrypter and decrypter are middlewares in charge of performing encryption and decryption operations on data and metadata. Keymaster is responsible for deciding whether a resource should (or should not) be encrypted and which encryption key should be used[2]. Swift supports a variety of keymaster implementations, including *Swift-KeyRotate*[3] proposed by IBM. The Swift-KeyRotate is a hierarchical key management system that manages three types of keys: a top-level *Master Key*; *Data Encryption Keys* (DEKs), used to decrypt and encrypt user/system metadata and user data; and *Key Encryption Keys* (KEKs), used internally in the keymaster middleware to protect other KEKs and DEKs. As data are hierarchically organized in accounts, containers, and objects, also KEKs and DEKs are hierarchically organized according to the account/container/object hierarchy (Figure 4). More precisely, a KEK and a DEK are generated for each account, container, and object. DEKs associated with accounts and containers are used to encrypt the metadata of the accounts and containers, respectively. DEKs associated with objects are used to encrypt both objects and their metadata. The Master Key (which is stored in the Barbican

[1] https://github.com/openstack/swift/blob/master/CHANGELOG
[2] http://specs.openstack.org/openstack/swift-specs/specs/in_progress/at_rest_encryption.html
[3] https://github.com/ibm-research/swift-keyrotate

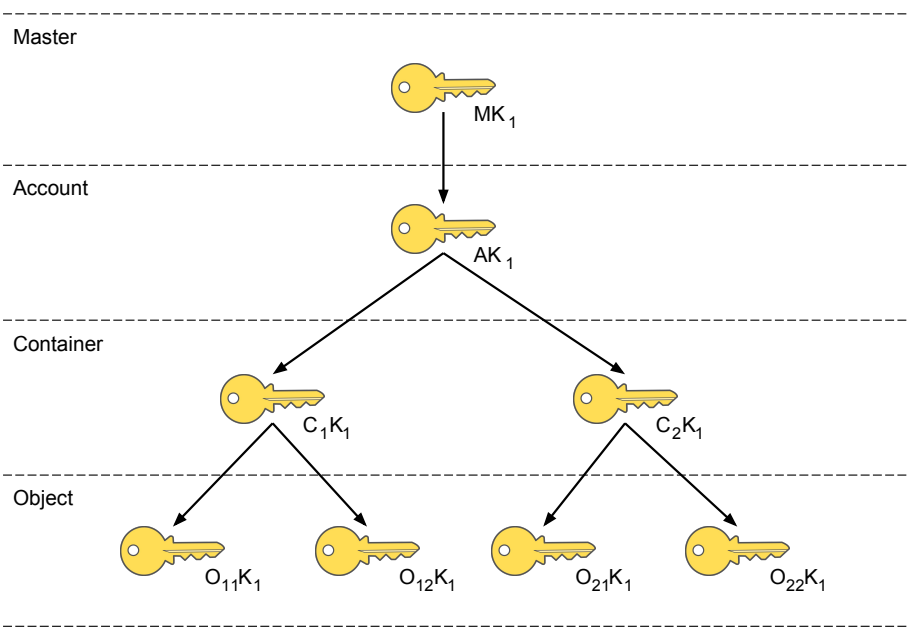

Fig. 5. Swift-KeyRotate: An example of KEK hierarchy with two containers and four objects

system, the secret storage of OpenStack) is used to encrypt the KEK associated with an account. Then, the KEK associated with an entity (i.e., an account, a container, or an object) is used to encrypt the DEK associated with the same entity and the KEKs associated with the entities of the level below (if any). Figure 4 illustrates the hierarchical organization of KEKs and DEKs. When a user authenticates to OpenStack via Keystone (the identity server of OpenStack), the user is associated with an account and therefore she can access a Master Key that is retrieved from Barbican through the user's authentication token.

Good key management practice requires a periodic key rotation, meaning that encryption keys must be periodically changed. The rotation of the Master Key stored in Barbican is similar to the approach adopted by systems for industrial key-lifecycle management [7, 15]. However, in Swift-KeyRotate, it is not sufficient to rotate the Master Key since an adversary could have stored the key of a lower level and then could be still able to obtain access to all the underlying data. Key rotation is then performed on all levels and is also needed to securely delete objects. We note that key rotation involves only the KEKs while the DEKs are generated when the corresponding entity is created and are never changed. As an example, consider two containers, C_1 and C_2, each of which includes two objects, $\{o_{11}, o_{12}\}$ and $\{o_{21}, o_{22}\}$, respectively. Figure 5 illustrates the corresponding KEK hierarchy: nodes of the hierarchy represent keys and an arc from a key k to key k' means that k' is encrypted using k (e.g., in the figure an arc from MK_1 to AK_1 means that the account KEK is encrypted via

the Master Key). Suppose that a user wishes to delete object o_{11}. In this case, new KEKs have to be generated for all entities in the key hierarchy that are on the path to object o_{11} (i.e., container C_1, account A, and the master). Furthermore, the KEKs of all entities whose parent KEKs have been changed are re-encrypted with the new parent key. In our example, the KEK $O_{12}K_1$ of object o_{12} is encrypted with the new KEK associated with container C_1, say C_1K_2, the KEK C_2K_1 of container C_2 is encrypted with the new KEK of account key A, say AK_2, and the account key AK_2 is encrypted with the new Master Key, say MK_2.

3 Client-side Encryption

With client-side encryption, the data owner encrypts her data before outsourcing them to a cloud provider. The encryption keys are therefore stored at the client-side and are never exposed to the cloud provider, which cannot decrypt the outsourced data. This solution is typically applied when the cloud provider is honest-but-curious or lazy/malicious.

3.1 Discussion

Like for the server-side encryption, this solution has some advantages and disadvantages for users and the cloud provider. From the point of view of the users, the main advantage is an increase of the spectrum of cloud providers to which a data owner can outsource her data. In fact, since the data are encrypted at the client-side, the data owner can also leverage the services of less reputable cloud providers, which are typically cheaper than well-known cloud providers. The main disadvantages are that the data owner has to directly manage the encryption keys and has to enforce access control restrictions as well as changes in the access control policy. In this scenario, access control can be enforced using an approach based on *selective encryption* [12]. Intuitively, selective encryption means that the data owner encrypts different portions of her data using different keys and discloses to each user only the encryption keys used to protect the data they can access. Whenever the access control policy changes, the data owner must download the involved data, decrypt and re-encrypt them with a new encryption key, re-upload the new encrypted data, and share the new encryption key with authorized users. Clearly, such an approach puts much of the work at the data owner side, introducing a bottleneck for computation and communication. Another disadvantage is that both the client and the server storing the data may be the subject to attacks from an adversary. Common client-side attacks include, for example, the *man-in-the-browser* attack, in which an adversary takes control over a part of the browser (e.g., browser extension hijacking) to replace the cryptography algorithms used by the cloud provider with algorithms controlled by the adversary. This attack can also compromise the key-generation and the client-side integrity checks without the client being aware of it. The

adversary might also try to compromise the server to use it as a vehicle to send malicious code to the client. For services that provide access via browser, in fact, the server still plays a central role by providing the JavaScript code that encrypts the data before upload. If an adversary is able to replace this code with a malicious one, the adversary can compromise the confidentiality of the outsourced data collection.

From the point of view of the cloud provider, the main advantage is that the cloud provider should not be worried about the protection of data, which is guaranteed by client-side encryption. The main disadvantage is that deduplication techniques cannot be used since the same plaintext data are encrypted by different data owners using different keys, thus generating different ciphertexts. A possible approach for addressing this issue consists in using *convergent encryption,* a cryptosystem that can generate identical ciphertexts from identical plaintext data. While interesting, this techniques is still vulnerable to the brute force attack described in Section 2.1.

Examples of cloud storage services supporting client-side encryption are SpiderOak and MEGA [9]. In the following, we describe the MEGA system.

3.2 Case Study: Mega

MEGA system supports browser-based User Controlled Encryption (UCE), meaning that resources are automatically encrypted on the user's device before they are stored on MEGA cloud service [17]. Client-side encryption uses different encryption keys managed by the data owner: a *Master Key* is a user's key used to protect the symmetric *file key* adopted for encrypting a file that is stored on MEGA; a *user password* is then used to encrypt the Master Key. File keys encrypted with the Master Key as well as the Master Key encrypted with the user password are stored on MEGA. Different files are encrypted with different file keys and therefore the knowledge of a file key allows a user to decrypt only the file encrypted with such a key. This mechanism enforces selective encryption, as illustrated in the previous section. Note that an adversary compromising a store server of MEGA cannot decrypt the encrypted files stored on the node since the encryption key is managed at the client-side. Furthermore, MEGA uses HMAC to provide integrity guarantee to the file stored in MEGA store node. In this way, an adversary with access to a MEGA store node and the file key of a file cannot replace the file without the original user who has uploaded the file noticing that it has been changed.

Figure 6 illustrates the MEGA encryption and decryption processes. When a user wishes to store in the MEGA system a resource, a new file key is generated with the support of a cryptographically strong random number with entropy coming from both HTML5 APIs and mouse/keyboard entropy pool. The file is then encrypted with the file key and AES-12 and the resulting ciphertext is uploaded on MEGA. The encryption operation is performed either by the MEGA client or directly in the browser, using JavaScript. The file key is encrypted with the Master Key that in turn is protected with the user password. The resulting encrypted keys are then uploaded on MEGA (Figure 6(a)). When a user wishes

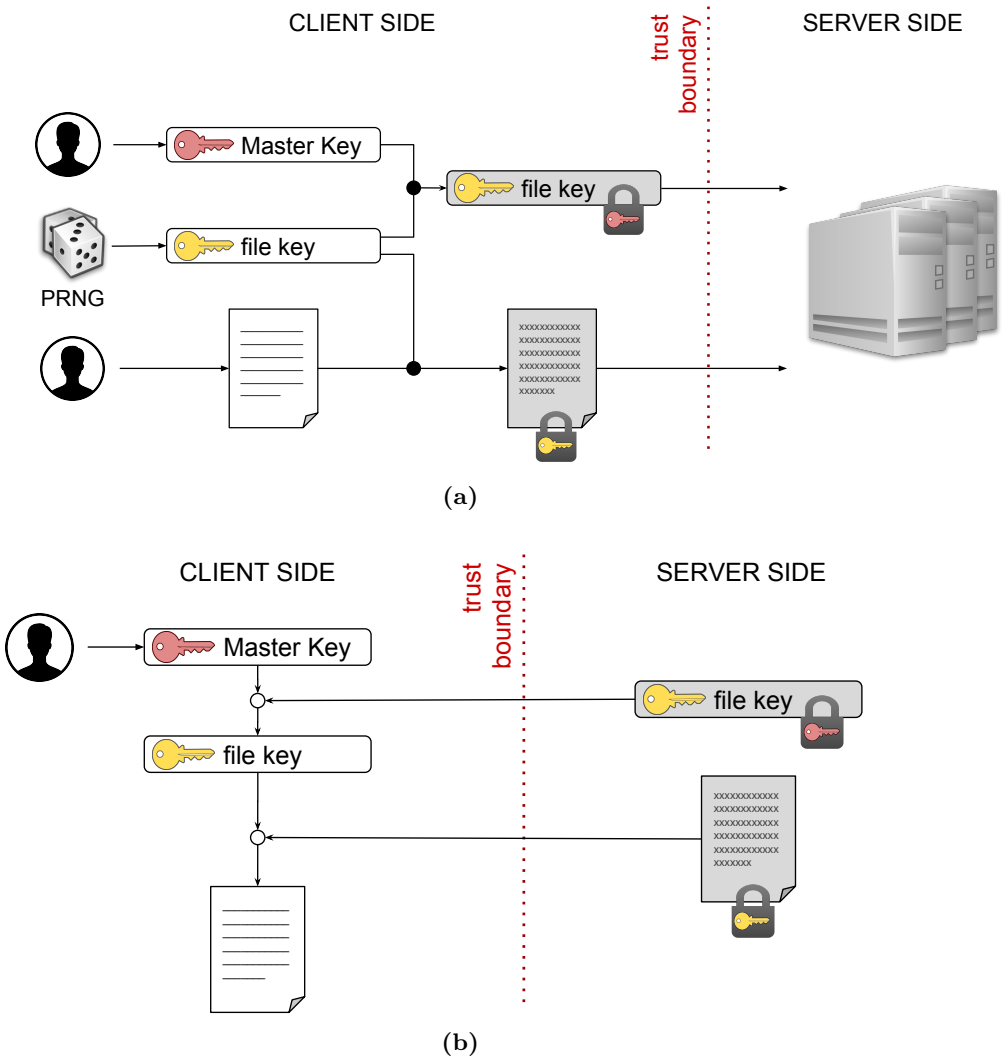

Fig. 6. MEGA upload (a) and download (b) process

to access a given file, she first provides her password, which is used to decrypt the Master Key. The file key of the file of interest is then decrypted, using the Master Key, and it is used to decrypt the file. Post-download integrity checks are performed via a chunked variant of the Counter with CBC-MAC (CCM) mode, which is an encryption mode only defined for block ciphers with a block length of 128 bits. Note that MEGA supports *end-to-end* encryption, meaning that encryption and decryption operations are performed at the client side.

With respect to the ability of supporting deduplication, MEGA can apply a deduplication process only when a user copies/pastes a file within her cloud drive or when the file is shared with another user who imports it. In fact, even if two (or more) users upload the same encrypted file, it will appear different since the file is encrypted using different keys.

Resource sharing is supported using two different strategies. The first strategy consists in sharing a public link that will allow a user receiving it to decrypt the corresponding resource, as the file key used to encrypt the file is included in the link (it is important to note that the link is generated at the client side and not at the server side). With this strategy, the public link can be shared with anyone who may not necessarily have a MEGA account. The second strategy is only applicable between MEGA users and is based on asymmetric encryption (RSA-2048). Each user is associated with a public key and a private key both stored on MEGA: the public key is stored in plaintext and the private key is stored in encrypted form, using the Master Key of the user as encryption key. When a user, say A, wishes to share a resource with another user, say B, A encrypts the corresponding file key with the public key of B and the resulting ciphertext is stored on MEGA. When B wishes to access the resource, she first retrieves from MEGA her encrypted private key, decrypts it using her Master Key and the resulting plaintext private key is used to decrypt the file key that B can use for decrypting the file of interest. To provide access revocation to users who were previously given access to the file key, MEGA applies a classical access control policy defined by the data owner. A revoked user is therefore prevented access to the encrypted files. Note that MEGA is trusted to correctly enforce the access control policy defined by the data owner.

4 Hybrid encryption

The hybrid approach combines client-side encryption with server-side encryption to improve efficiency in data management. Hybrid approaches are usually based on different layers of encryption with some encryption keys managed at the client side and other encryption keys managed at the server side. The latter keys are needed by the cloud provider to correctly enforce changes in the access control policy.

4.1 Discussion

The main advantage of the hybrid approach is the efficient enforcement of changes in the access control policy without impacting the confidentiality of the resources. In fact, while with client-side encryption changes in the access control policy must be enforced by the data owner (Section 3), with a hybrid approach such changes can be enforced directly by the cloud provider. This approach can therefore be applied only when the cloud provider is honest-but-curious, since the provider has to correctly enforce the changes as dictated by

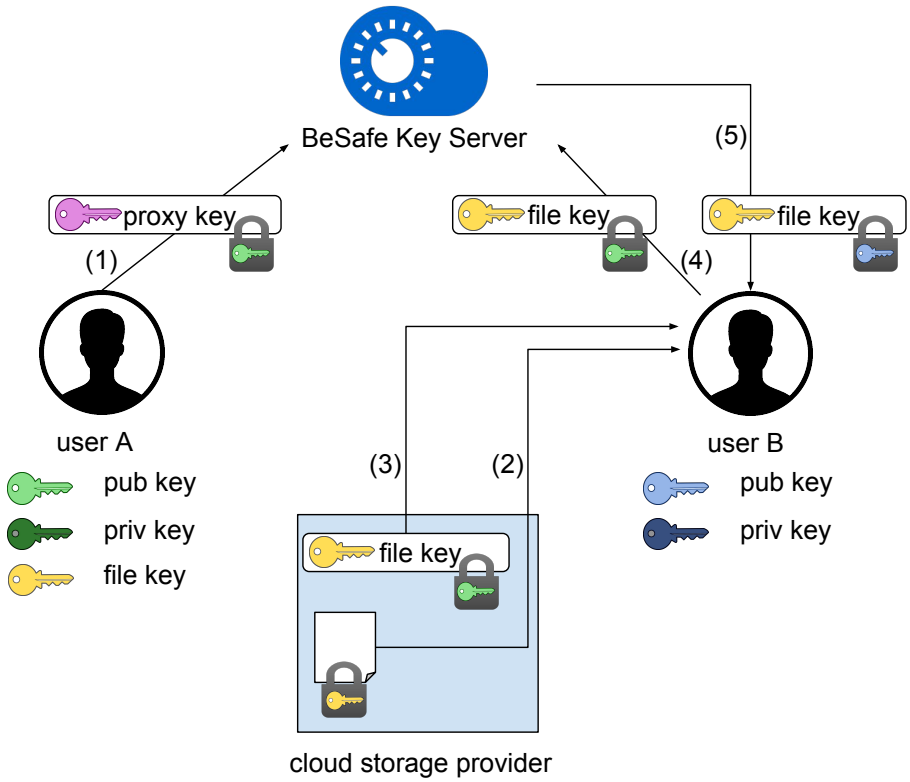

Fig. 7. BeSafe proxy re-encryption architecture

the data owner. An example of commercial solution adopting this approach is BeSafe SkyCryptor[4], a commercial platform providing end-to-end encryption. BeSafe is based on a honest-but-curious proxy (BeSafe Key Server) performing a proxy re-encryption [2, 8] on encryption keys, and on a public cloud storage provider storing the encrypted data. Proxy re-encryption is a cryptographic technique that transforms a ciphertext generated with a key k into a ciphertext that can be decrypted using a different key k', without the need for decryption the original ciphertext. Hence, it can be performed also by a party not trusted for the plaintext content of the data. Each user of the BeSafe SkyCryptor has a pair of public and private keys. Whenever a user wants to store a resource at the public cloud provider, the resource is first encrypted at the client side using a symmetric encryption key, called *file key*. The encrypted resource and the file key, encrypted with the public key of the user, are then stored on the cloud provider. Resources can be shared only among users with a BeSafe account. Figure 7 shows an example of sharing between user A and user B. User A first generates a new *proxy key*, encrypts such a key with her public key, and

[4] https://besafe.io/

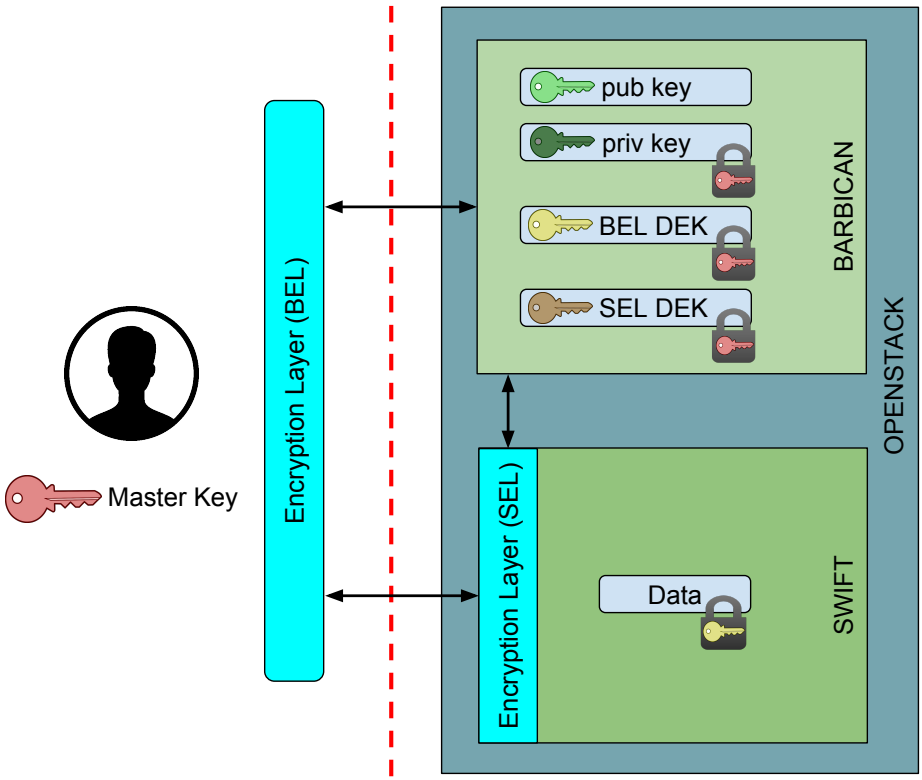

Fig. 8. EncSwift architecture

sends the resulting ciphertext to the BeSafe Key Server (1). *B* downloads the encrypted resource (2) from the public cloud storage provider, along with the corresponding encrypted file key (3). The encrypted file key is then sent to the BeSafe Key Server (4) that proxy-re-encrypts it using the proxy key generated by *A*. The result of the proxy re-encryption is sent to *B* (5) who can decrypt it through her private key for retrieving the file key and then can use the retrieved file key to decrypt the resource [19].

4.2 Case Study: EncSwift

EncSwift is a tool for providing data-at-rest encryption and enforcing access control when relying on a honest-but-curious cloud provider [3, 4]. This tool is based on OpenStack Swift where, as already discussed in Section 2.2, data are hierarchically organized in accounts, containers, and objects. The access control enforcement mechanism implemented by EncSwift is based on selective encryption (Section 3.1) and over-encryption approaches [11, 10]. According to the over-encryption approach, each user has a symmetric encryption key and each resource is encrypted with a symmetric key that depends on the access

control policy regulating access to the resource. This first client-side encryption, called Base Encryption Layer (BEL), is needed to protect the confidentiality of the resources from the cloud provider. Resource encryption keys are organized in a key derivation hierarchy so that each user can use her symmetric key for deriving the encryption keys of all and only the resources she is entitled to access. Policy updates are enforced by applying a second layer of encryption at the server side, called Surface Encryption Layer (SEL). SEL encryption is applied whenever there are users who are not authorized to access an object, but they know the underlying BEL key. This happens, for example, when access to a resource is revoked to a user: the revoked user could have maintained a copy of the BEL key and therefore she could still be able to pass the BEL layer and access the object for which she does not have the access authorization anymore. A user will then be able to access an object only if she knows both the SEL key and the BEL key with which the object is encrypted. We now describe the keys needed to implement the over-encryption approach in Swift, how the access control policy defined by the data owner can be enforced through selective encryption, and how to enforce policy updates.

Keys. The core component of EncSwift is the Encryption Layer (Figure 8), which is in charge of encrypting objects before outsourcing them to the cloud provider, and of decrypting them when they are retrieved from the cloud provider. The implementation of over-encryption in OpenStack Swift is then based on the definition and management of different keys: Master Keys (MKs), RSA key pairs, RSA signature key pairs, Data Encryption Keys (DEKs), and Key Encryption Keys (KEKs). Each user is associated with a symmetric Master Key and two pairs of public and private keys: one pair is used for encryption (RSA key pair) and one pair is used for signing messages (RSA signature pair). A DEK is a symmetric key that the Encryption Layer uses to encrypt (decrypt) an object stored on the cloud provider. All objects in the same container are initially encrypted with the same DEK, then a new DEK is generated whenever a policy update occurs. The Master Key is kept on the client side while all the other keys are stored in Barbican, the OpenStack Secret Storage, or can be stored and managed through other key management services [6]. The user's public keys are stored in plaintext while the corresponding private keys are encrypted with the Master Key. DEKs are encrypted and stored in the form of Key Encryption Keys (KEKs), which should not be confused with the KEK used in the Swift-KeyRotate approach (Section 2.2). The encryption of the DEK can be performed in two different ways. A first way consists in encrypting a DEK with the user's Master Key (*symmetric KEK*). In this case, only the user who knows the Master Key can decrypt the KEK. A second way consists in encrypting the DEK with the RSA public key of a user and signing it with the RSA signature private key of the user who owns the resource protected with the DEK (*asymmetric KEK*). This second strategy allows the user who own the resource to share a DEK (and therefore the access to the corresponding resource) with other users.

Selective encryption. All users in the system can define an access control policy for the objects they own, which can then be translated into an equivalent policy-based encryption as follows. First, a data owner creates as many containers as needed, and, for each of them, defines a DEK. The data owner then encrypts all the objects in the same container with the DEK of the container. This means that all objects in a container are characterized by the same access control list (i.e., they can be accessed by the same set of users). The DEK is then encrypted with the Master Key of the data owner and, for each user in the access control list of the objects in the container, the DEK is encrypted with her RSA public key and signed by the data owner with her signature private key. When a user wishes to access a specific object, the object descriptor is first accessed to retrieve the identifier of the DEK used to encrypt the object. This identifier is then used to retrieve the corresponding KEK and derive the DEK. Derivation will require the user to use either her own Master Key (for symmetric KEK), or her RSA private encryption key (for asymmetric KEK). Note that, to improve the efficiency of the subsequent accesses to the key and simplify the procedure, once a DEK provided by another user is extracted from an asymmetric KEK, the KEK is replaced by a symmetric KEK built using the Master Key of the user.

Policy updates. Policy changes refer to the insertion and deletion of users, objects, and authorizations. The insertion of a user requires the generation of her Master Key, RSA key pair, and RSA signature key pair and the storage of the public keys in Barbican. The removal of a user requires only the removal of her public keys from Barbican. The removal of an object requires its deletion from the container including it. The insertion (grant) and removal (revoke) of authorizations as well as the insertion of new objects require the involvement of the cloud provider for the application of a second layer of encryption (SEL layer). The SEL layer is developed as a new middleware and inserted into the pipeline, using the same approach adopted by IBM and explained in Section 2.2. We now describe how grant/revoke operations and the insertion of a new object in a container are implemented.

In case of a grant operation, it is sufficient to generate a new (asymmetric) KEK for the granted user and store it in Barbican. This new KEK is generated by the owner of the container. In case of a revoke operation, it is not sufficient to remove the KEK that allows the revoked user to derive the DEK of the container since the user could have locally stored the KEK and therefore could still have access to the objects stored in the container. To avoid this problem, the owner of the container asks the cloud provider to over-encrypt all the objects in the container with a SEL key that only non-revoked users can derive. Therefore, each container is associated with two keys: a key at the BEL level that can be derived by all users originally authorized for the container, and a key at the SEL level that can be derived only by non-revoked users. In case of insertion of a new object into a container, the new object inherits the access control list of the container. To correctly enforce such an authorization policy, the new object is encrypted with the BEL DEK key associated with the container and, if the contained was involved in a revoke operation, with the SEL DEK key associated

with the container. Since, however, the authorization policy of the new object has never been updated, the adoption of SEL encryption over it might be an overdo. A new BEL DEK key is the adopted to protect objects that are inserted into a container on which revoke operations had been applied. As a consequence of the revoke operation, a new DEK BEL key (and the corresponding KEKs for the users in the new access control list) is generated for the container, and used for objects that will be inserted into the container after the revoke operation. While for existing objects over-encryption is needed to guarantee protection from the revoked user, new objects can be encrypted with the new DEK known only to the users actually authorized for them.

The implementation of over-encryption for the enforcement of revoke operations can operate in different ways, depending on the time at which SEL encryption is applied [4]: materialized at policy update time (*immediate*), performed at access time (*on-the-fly*), or performed at the first access and then materialized for subsequent accesses (*opportunistic*).

- *Immediate.* The cloud provider applies over-encryption when the owner revokes the authorization over a container to a user. Immediate over-encryption requires the owner to generate, at policy update time: the SEL DEK necessary to protect the objects in the revoked container, and the KEKs necessary to authorized users (and to the server) to derive such a SEL DEK. Also, the objects in the container will be over-encrypted. The cloud provider will then immediately read from the storage the objects in the container, re-encrypt their content with the new SEL DEK (possibly removing SEL encryption), and write the over-encrypted objects back to the storage. Hence, immediately after the policy update, the objects in the container are stored encrypted with two encryption layers. Every time a user needs to access an object in the container, the server will simply return the stored version of the requested object. This approach can be applied when policy updates are rare and the container size is moderated, because no overhead is applied when objects are downloaded, except for the supplementary decryption step with the SEL DEK at the client side. The main drawback is that encryption cost must be paid for the whole container, even for objects that are not accessed before next policy update.
- *On-the-fly.* The cloud provider applies over-encryption every time a user accesses an object. Then, even if the owner of the container asks the cloud provider to over-encrypt the objects in the container, the provider only keeps track of this request, but it does not re-encrypt the objects. When a user needs to access an object in the container, the cloud provider possibly over-encrypts the object before returning it to the user. The advantage of this approach is that over-encryption is applied only if needed. However, if an object is accessed multiple times, the object is encrypted all the times.
- *Opportunistic.* This approach aims to combine the advantages of both immediate over-encryption and on-the-fly over-encryption. Opportunistic over-encryption requires the owner, when a user is revoked access to a container, to define both the SEL DEK necessary to protect the objects in the revoked

container, and the KEKs necessary to authorized users (and to the server) to derive the SEL DEK. Similarly to the on-the-fly approach, the provider over-encrypts an object in the revoked container only when it is first accessed. However, instead of discarding it, the result of over-encryption is written back to storage for future accesses. The main disadvantage of this approach is that the SEL protection must be removed when the object is downloaded after a policy update that generated a new SEL DEK because the object should be protected with the new SEL key.

5 Discussion and conclusions

The design of efficient techniques for protecting the confidentiality and regulating access to data stored at external cloud providers has been the subject of several efforts in the research as well as industrial community. In this chapter, we have presented an overview of recent approaches that protect the confidentiality of the data through encryption as well as enforce access control restrictions. These techniques mainly differ in how encryption is enforced, which depends on the trust assumption on the cloud provider. Interesting evolution of these encryption-based data protection techniques are related to the use of All-or-Nothing Transform (AONT) for enforcing changes in the access control policy without requiring the support of the cloud provider [5], and the consideration of novel distributed cloud storage systems (e.g., Storj [1, 22], Sia [21] and FileCoin [16]) characterized by the availability of multiple (untrusted) nodes that can be used to store resources in a distributed manner.

Acknowledgments

This work was supported in part by the EC within the H2020 under grant agreement 644579 (ESCUDO-CLOUD) and within the FP7 under grant agreement 312797 (ABC4EU).

References

1. A peer-to-peer cloud storage network, Storj Labs Inc. (2016), `https://storj.io/storj.pdf`
2. Ateniese, G., Fu, K., Green, M., Hohenberger, S.: Improved proxy re-encryption schemes with applications to secure distributed storage. ACM Transactions on Information System Security 9(1), 1–30 (February 2006)
3. Bacis, E., De Capitani di Vimercati, S., Foresti, S., Guttadoro, D., Paraboschi, S., Rosa, M., Samarati, P., Saullo, A.: Managing data sharing in openstack swift with over-encryption. In: Proc. of the 3rd ACM Workshop on Information Sharing and Collaborative Security. Vienna, Austria (October 2016)

4. Bacis, E., De Capitani di Vimercati, S., Foresti, S., Paraboschi, S., Rosa, M., Samarati, P.: Access control management for secure cloud storage. In: Proc. of the 12th EAI International Conference on Security and Privacy in Communication Networks. Guangzhou, China (October 2016)
5. Bacis, E., De Capitani di Vimercati, S., Foresti, S., Paraboschi, S., Rosa, M., Samarati, P.: Mix&slice: Efficient access revocation in the cloud. In: Proc. of the 23rd ACM Conference on Computer and Communication Security. Vienna, Austria (October 2016)
6. Bacis, E., Rosa, M., Sajjad, A.: EncSwift and key management: An integrated approach in an industrial setting. In: Proc. of the 3rd IEEE Workshop on Security and Privacy in the Cloud. Las Vegas, Nevada (October 2017)
7. Björkqvist, M., Cachin, C., Haas, R., Hu, X.Y., Kurmus, A., Pawlitzek, R., Vukolić, M.: Design and implementation of a key-lifecycle management system. In: Proc. of the 14th International Conference on Financial Cryptography and Data Security. Tenerife, Canary Islands (January 2010)
8. Blaze, M., Bleumer, G., Strauss, M.: Divertible protocols and atomic proxy cryptography. In: Proc. of the 29th Annual International Conference on the Theory and Application of Cryptographic Techniques. Espoo, Finland (May-June 1998)
9. Daryabar, F., Dehghantanha, A., Choo, K.K.R.: Cloud storage forensics: Mega as a case study. Australian Journal of Forensic Sciences 49(3), 344–357 (2017)
10. De Capitani and Foresti, S., Foresti, S., Jajodia, S., Paraboschi, S., Samarati, P.: Over-encryption: Management of access control evolution on outsourced data. In: Proc. of the 33rd International Conference on Very Large Data Bases. Vienna, Austria (September 2007)
11. De Capitani di Vimercati, S., Foresti, S., Jajodia, S., Paraboschi, S., Samarati, P.: Encryption policies for regulating access to outsourced data. ACM Transactions on Database Systems 35(2), 12:1–12:46 (April 2010)
12. De Capitani di Vimercati, S., Foresti, S., Livraga, G., Samarati, P.: Selective and private access to outsourced data centers. In: Khan, S., Zomaya, A. (eds.) Handbook on Data Centers. Springer (2015)
13. De Capitani di Vimercati, S., Foresti, S., Livraga, G., Samarati, P.: Practical techniques building on encryption for protecting and managing data in the cloud. In: Ryan, P., Naccache, D., Quisquater, J.J. (eds.) Festschrift for David Kahn. Springer (2016)
14. Dropbox business security. a dropbox whitepaper. `https://cfl.dropboxstatic.com/static/business/resources/dfb_security_whitepaper-vfllunodj.pdf`
15. Ducatel, G., Daniel, J., Dimitrakos, T., El-Moussa, F.A., Rowlingson, R., Sajjad, A.: Managed security service distribution model. In: Proc. of the 4th International Conference on Cloud Computing and Intelligence Systems. Beijing, China (August 2016)
16. Filecoin: A decentralized storage network, protocol labs (2017), `https://filecoin.io/filecoin.pdf`
17. Information regarding security and privacy by design at MEGA. `https://mega.nz/help/client/webclient/security-and-privacy`
18. Jhawar, R., Piuri, V., Samarati, P.: Supporting security requirements for resource management in cloud computing. In: Proc. of the 15th IEEE International Conference on Computational Science and Engineering. Paphos, Cyprus (December 2012)
19. Jivanyan, A., Yeghiazaryan, R., Darbinyan, A., Manukyan, A.: Secure collaboration in public cloud storages. In: Proc. of the 21st CYTED-RITOS International Workshop on Groupware. Yerevan, Armenia (September 2015)

20. Samarati, P., De Capitani di Vimercati, S.: Cloud security: Issues and concerns. In: Murugesan, S., Bojanova, I. (eds.) Encyclopedia on Cloud Computing. Wiley (2016)
21. Sia: Simple decentralized storage (2014), https://www.sia.tech/whitepaper.pdf
22. Wilkinson, S., Boshevski, T., Brandoff, J., Prestwich, J., Hall, G., Gerbes, P., Hutchins, P., Pollard, C.: Storj - A peer-to-peer cloud storage network (2014), https://storj.io/storj.pdf