Managing Data Sharing in OpenStack Swift with Over-Encryption

Enrico Bacis
Università di Bergamo
24044 Dalmine - Italy
enrico.bacis@unibg.it

Sabrina De Capitani di Vimercati
Università degli Studi di Milano
26013 Crema - Italy
sabrina.decapitani@unimi.it

Sara Foresti
Università degli Studi di Milano
26013 Crema - Italy
sara.foresti@unimi.it

Daniele Guttadoro
Università di Bergamo
24044 Dalmine - Italy
daniele.guttadoro@unibg.it

Stefano Paraboschi
Università di Bergamo
24044 Dalmine - Italy
parabosc@unibg.it

Marco Rosa
Università di Bergamo
24044 Dalmine - Italy
marco.rosa@unibg.it

Pierangela Samarati
Università degli Studi di Milano
26013 Crema - Italy
pierangela.samarati@unimi.it

Alessandro Saullo
Università di Bergamo
24044 Dalmine - Italy
alessandro.saullo@unibg.it

ABSTRACT

The sharing of large amounts of data is greatly facilitated by the adoption of cloud storage solutions. In many scenarios, this adoption could be hampered by possible concerns about data confidentiality, as cloud providers are not trusted to know the content of the data they store. Especially when the data are organized in objects, the application of an encryption layer is an interesting solution to this problem, because it offers strong confidentiality guarantees with a limited performance overhead. In a data sharing scenario, the management of access privileges then requires an adequate support for key derivation and for managing policy evolution.

We present a solution that provides transparent support for the encryption of objects stored on Swift. Our system offers an efficient management of the updates to the access control policy, including revocation of authorizations from some of the sharing users. We explore several alternatives for the architecture, associated with distinct levels of transparency for the applications, and integrate different options for the management of policy updates. Our implementation and experiments demonstrate the easy integration of the approach with existing cloud storage solutions.

Categories and Subject Descriptors

Security and privacy [**Security services**]: Access control, Authorization; Security and privacy [**Database and storage security**]; Computer systems organization [**Distributed architectures**]: Cloud Computing

Keywords

Access control; Policy revocation; Resource encryption; OpenStack; Swift; EncSwift

1. INTRODUCTION

Cloud storage has already a central role in supporting the sharing of data among users and organizations. The approach currently supported by most applications assumes trust in the ability of the cloud provider to support data confidentiality. Such ability can leverage on encryption, providing a protection layer. The use of encryption to protect the confidentiality of the stored data offers clear benefits and can be expected to become commonplace when using cloud storage, as encryption for communication channels has become in the past few years. To support this evolution, it is crucial on one side to advance the support for the sharing of encrypted data, and on the other side to facilitate the integration of these services with existing cloud storage architectures. Since data to be shared are stored in an encrypted format, it is necessary to design and implement key management solutions that allow users authorized to access a given content to get the encryption key that has been used to protect that content. A delicate issue is associated with the management of policy updates, especially for revocation of access privileges. This is a critical aspect and current systems offer limited support for it. Solutions enforcing access revocation have then to be transferred to exiting cloud storage systems. This transfer not only facilitates the adoption of these solutions in real systems, but it also clarifies the issues that have to be faced for their concrete deployment, illustrates alternatives for their implementation that have a significant impact on performance, and in the end demonstrate the applicability of these techniques.

The work presented in this paper aims at the following two goals. First we propose an investigation on how techniques for sharing data using the services of a cloud provider can support both static and dynamic authorization policies. This analysis is done in the domain of the OpenStack Swift system, arguably one of the most successful solutions for

cloud storage. Second, we show that approaches for the efficient management of policy updates can be integrated with current cloud storage services. We propose different options, characterized by different levels of transparency, for the integration of policy enforcement with existing applications. The support for efficient policy updates relies on the use of Over-Encryption, and we consider three different options for its realization, considering distinct security assumptions and performance. The implementation permits to verify the concrete performance exhibited by each option, offering guidelines for the selection of the option that a system may adopt, depending on the specific requirements of the application.

The remainder of this paper is organized as follows. Section 2 provides an overview of the reference platform (i.e., OpenStack Swift). Section 3 describes our EncSwift tool, used to encrypt data at the Cloud Service Provider. Section 4 illustrates two possible alternatives, App-aware and Proxy architectures, for enforcing selective data sharing. Section 5 considers the efficient management of policy evolution in EncSwift, describing the principles for the application of Over-Encryption. Section 6 describes the main design options that characterize the implementation of EncSwift, discussing the motivations that led us to some specific design choices. Section 7 presents the experimental results derived from the application of EncSwift alternatives. Section 8 discusses related work. Finally, Section 9 concludes the paper.

2. OPENSTACK SWIFT

OpenStack [14] is an open-source platform for creating and managing cloud infrastructures, originally developed by NASA and Rackspace. OpenStack operates with large pools of computing, storage, and networking resources, all managed through a dashboard that gives administrators control.

Swift, the object storage system used by OpenStack, plays a central role in the architecture we propose. It allows users to store data in the form of objects (in most cases equivalent to files) using RESTful APIs. Objects have a canonical name containing three parts: *tenant/container/object*. The tenant corresponds to a customer (for public clouds) or to a specific application domain (for private clouds). The container is a loose equivalent of the directory of common file systems. The object is the stored data content and is included in a container.

To store objects, Swift relies on clusters of standardized servers (nodes). Depending on the kind of Swift processes running on a Swift node, we can distinguish two kinds of nodes: *Proxy* nodes and *Storage* nodes.

Proxy nodes are the public face of Swift, as they are the only ones that communicate with external clients. As a result, they are the first and last to handle an API request to upload objects, modify metadata, and create containers. All requests to and responses from Proxy nodes use standard HTTP commands and response codes.

Storage nodes run one or more *tenant/container/object* server processes, as well as a number of other services to maintain data consistency. The *tenant/container/object* server process handles requests regarding metadata for the tenant, container, or object, respectively. This information is stored by the account server process in SQLite databases on disk. Furthermore, the *object* server process provides a blob storage service that can store, retrieve, and delete objects on the Storage node's drives.

When a valid request on a given object is sent to Swift, the Proxy node will verify the request, determine the correct Storage nodes responsible for the object (based on a hash of the object name), and will send the request to these Storage nodes concurrently. The Storage nodes will then return a response to the Proxy node, that will in turn forward the response to the requester.

To define the users that are able to access the stored objects, Swift uses *Access Control Lists* (ACLs) organized in two levels.

Tenant level ACL: set of users that can perform administrative level operations on the tenant (e.g., grant privileges to other users).

Container level ACL: set of users that can perform *read/write/listing* actions on the container. Users who are included in the *read* (*write* and *listing*, resp.) ACL are able to download objects from the container (upload objects to the container, and listing the container content, resp.).

We note that access control is currently not offered at the level of single object, thus objects inherit the ACL of the container they belong to. The evolution of Swift may lead to the introduction of this support and our proposal can be easily adapted to this evolution.

OpenStack also provides an identity (*Keystone*) and a secret storage (*Barbican*) service. Keystone provides a central directory of users mapped to the OpenStack services they can access and acts as a common authentication system across the cloud operating system. Barbican offers a RESTful API designed for the secure storage, provisioning, and management of secrets such as passwords, encryption keys, and X.509 certificates.

3. CLOUD STORAGE ENCRYPTION

We consider a scenario where users outsource their data to an external cloud service provider using OpenStack Swift and would like to selectively share such data with others. Each user then stores her data through Swift objects, which are organized in containers, and can grant other users access to each of her containers (and hence to all the objects in it). We assume the external cloud provider to be *honest-but-curious*: it is trusted to correctly perform the operations required by users, but it is not trusted for accessing the content of the objects. We also assume correct and trusted behaviors only by the client, meaning that we set the *trust-boundary* at the client.

To enforce access control restrictions (i.e., enable only authorized users to read the objects in each container), while preserving confidentiality of sensitive data from the cloud provider, we adopt *policy-based encryption* [7]. According to policy-based encryption, each object is encrypted with a key that all and only the users authorized for it know (or can derive).

In this paper, we illustrate the design of a component, *EncSwift*, that implements access control restrictions over data stored in Swift without the intervention of the data owner as well as of the cloud provider. The core building block of EncSwift is the *Encryption Layer* module, which is in charge of managing encryption and decryption operations of Swift objects to enforce access control.

3.1 Encryption Keys

The adoption of policy-based encryption for access control enforcement requires the definition and management of different keys discussed in the following.

Data-Encryption-Key (DEK). Each object stored in Swift is encrypted using a symmetric encryption algorithm (e.g., AES) and a secret DEK. Since data owners grant/revoke privileges at the container granularity, all the objects in the same container are initially encrypted with the same DEK (different objects can however be encrypted with different keys when enforcing policy updates, as discussed in Section 5).

RSA Key Pair. Each user in the system is associated with an asymmetric key pair used for encryption. This key pair is necessary to users for sharing objects with others, by properly encrypting and exchanging the DEKs used to encrypt the objects to be shared. Through her RSA key pair, a user can directly or indirectly derive all the DEKs used to encrypt the objects she is authorized to access.

Signature Key Pair. Each user is also associated with an asymmetric key pair for signing messages. Signatures are used to guarantee the integrity of both objects and of the keys (or information necessary for their derivation) exchanged between users.

Master Key (MK). To facilitate key management at the user side, each user can also have a personal symmetric encryption key (see Section 4.3). The master key enables the user to retrieve her RSA and signature key pairs.

Key-Encryption-Key (KEK). KEKs enable users to retrieve, starting from their own secret keys, the DEKs used to encrypt the objects they are authorized to access. Intuitively, a KEK is an encrypted DEK that only an authorized user can decrypt (and hence gain access to the objects encrypted with such a DEK). KEKs can be generated in two different ways, as illustrated in the following.

- *Symmetric KEK* is obtained by encrypting a DEK with the master key of a user. Since the master key of a user is known only to the user herself, only the user can create and use her own symmetric KEKs. To provide users with the ability to verify the integrity of symmetric KEKs, when generating a KEK the user also applies a MAC function to its content, and stores the result.

- *Asymmetric KEK* is obtained by encrypting a DEK with the public RSA key of a user. An asymmetric KEK can be created by any user knowing the corresponding DEK, but it can be used only by one user for key derivation: the one for which it has been produced. To verify the identity of the user who created the KEK and to assess the integrity of the key itself, asymmetric KEKs must be signed. Then, the user generating the KEK computes a hash of the KEK and signs it with her own private signature key.

The availability of a collection of KEKs allows a user to derive a large number of DEKs and access a wide set of objects, starting from the knowledge of a single secret key (i.e., the master key).

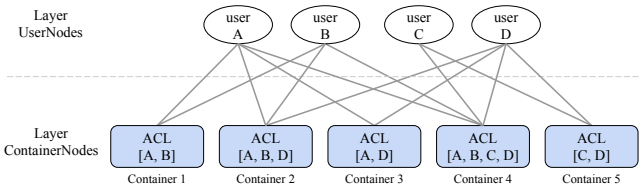


Figure 1: An example of a configuration with 4 users and 5 containers with different ACLs

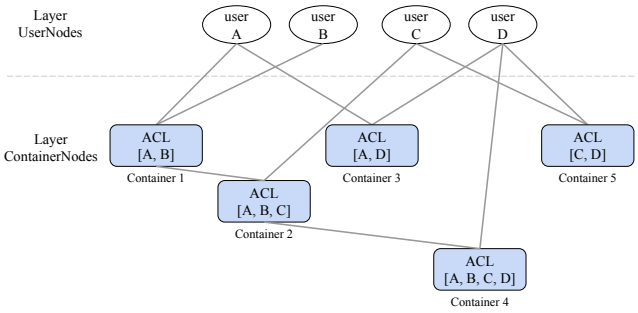


Figure 2: Graph with DEK derivation from other DEKs

3.2 Key Derivation Structure

The enforcement of sharing restrictions is driven by the definition of symmetric and asymmetric KEKs. To share access to a container with others, the data owner only needs to generate an asymmetric KEK for each of the users in the container ACL, and a symmetric KEK for herself. These KEKs will permit all the authorized users (and the container owner) to retrieve the DEK of the container. Figure 1 graphically illustrates a scenario characterized by four users (A, B, C, D), represented by white nodes, and five containers (Container 1,...,Container 5), represented by colored nodes. Edges in the graph represent KEKs: an edge between a user and a container is a KEK corresponding to the encryption of the DEK of the container with the user public key. For instance, the two edges between users A and B correspond to two KEKs that allow users A and B, respectively, to obtain the DEK used for encrypting the objects in Container 1.

To reduce the number of KEKs in the system (i.e., of edges in Figure 1), we introduce KEKs that encrypt a DEK with another DEK. A DEK can be used to protect another DEK only if the latter can be accessed by a superset of users than the former. For instance, Figure 2 represents a set of KEKs that enforce the same authorization policy as Figure 1. Here, user A should first obtain the DEK of Container 1 to be able to retrieve the DEK of Container 2, which in turn will enable her to obtain the DEK of Container 4.

3.3 Catalog Structure

Since the number of KEKs can be considerably high, they cannot be stored at the user side. We then define a *catalog* for each user, storing all and only the KEKs that the user needs to know for accessing the objects she is authorized to read. The catalog is stored in a Swift container, specifically created for this purpose. The containers storing users catalogs are stored in a specific tenant, used only for this purpose. In the following, we will refer to containers storing

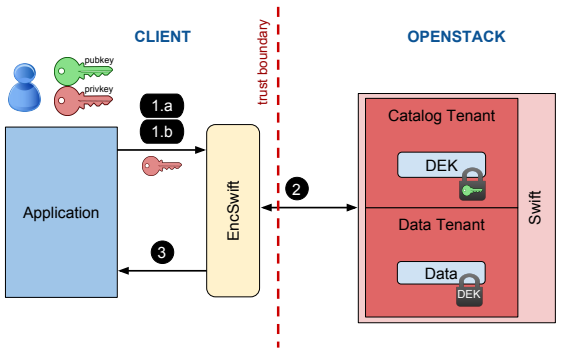


Figure 3: App-aware architecture

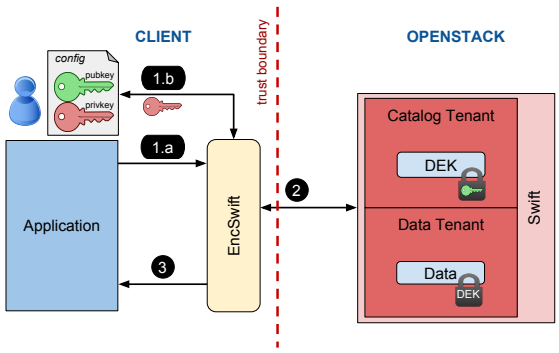


Figure 4: Proxy architecture

users catalogs as *meta-container* and to the corresponding tenant as *meta-tenant*. Note that, while we have a meta-container for each user, the meta-tenant is unique for the system.

When creating a new user, the Encryption Layer will initialize a meta-container for the user catalog. When creating a container, the data owner will create a symmetric KEK for herself, and an asymmetric KEK for each of the users in the ACL of the container. These KEKs are inserted into the catalog of the corresponding user.

Note that, to improve efficiency in key derivation, a user can replace an asymmetric KEK with the corresponding symmetric counterpart in her catalog after the first access. This practice reduces the size of the KEK and makes future derivations more efficient.

4. CLIENT-ONLY ENCRYPTION

This section describes two alternative architectures to perform policy-based encryption that do not require any modification to server components. In both architectures the encryption and decryption are performed at the client side. The *App-aware architecture* (Section 4.1) requires adjustments to the application that uses Swift, while the *Proxy architecture* (Section 4.2) is totally transparent to the application but requires an additional trusted proxy server, which may cause delays. Section 4.3 illustrates an approach, which can be integrated with both the App-aware and Proxy architectures, enabling users to store RSA and signature key pairs in the cloud.

4.1 App-aware Architecture

Many applications that use Swift as storage service, also adopt the *python-swiftclient* library[1] that provides several high-level APIs for the communication between applications and Swift server. In the App-aware architecture, the EncSwift component that is responsible for encrypting and decrypting the resources is obtained by modifying of the *python-swiftclient* library (see Section 6).

Figure 3 illustrates the App-aware architecture for policy-based encryption as well as the steps followed by the client to access a remote object. The process to access an (encrypted) object operates as follows.

- *Step 1*: EncSwift Encryption Layer receives the user's request (*step 1.a*) and her private key (*step 1.b*).
- *Step 2*: EncSwift retrieves, from the user's catalog stored in Swift, the KEK corresponding to the DEK used to protect the object of interest, and decrypts it to obtain such a DEK. Also, EncSwift retrieves the encrypted object and decrypts it using the retrieved DEK.
- *Step 3*: When the object has been decrypted by EncSwift, it is forwarded to the application client.

Note that the process necessary to upload an object operates in a similar way. Since the APIs now need also the user's keys to encrypt/decrypt the data, the API parameters have been enriched. Hence, the application needs to be modified to invoke the enriched APIs.

To prove the applicability of EncSwift in existing applications, we implemented a library that has been directly integrated into *SwiftBrowser*[2] (a simple web-app based on the Django web framework that permits to access Swift resources). The *python-swiftclient* library used by this application has been replaced with our modified version, to enrich its functionality with the Encryption Layer protection. In this way the user can interact with EncSwift using a browser.

During the login phase, the username, the tenant name, and the password must be provided. Then a session is created and managed by *SwiftBrowser* to store relevant data, including the authorization token issued by Keystone (the OpenStack identity service) that is used in place of the username and password.

Once the user has logged into the system, the GUI shows the list of containers that she is authorized to access. Using the web interface, the user can upload and download objects, access the object list of a specific container, and manage the container ACLs (i.e., grant/revoke authorizations).

The integration of EncSwift in SwiftBrowser preserves full compatibility with SwiftBrowser, which does not use any encryption layer.

4.2 Proxy Architecture

The App-aware architecture expects a modification of the application client. An adaptation to this architecture that provides enhanced transparency is the *Proxy architecture*, illustrated in Figure 4. To avoid changing the API parameters, in the Proxy architecture the EncSwift Encryption Layer reads a configuration file that defines the path of the user's keys. Then, the user does not need to provide to the

[1]https://github.com/openstack/python-swiftclient

[2]https://github.com/cschwede/django-swiftbrowser

Encryption Layer her own private key every time she wants to retrieve an object from Swift. The steps followed by the client to access an object are the same as the ones illustrated for the App-aware architecture (Figure 3), with the difference that in the Proxy architecture the private key is provided by the configuration file rather than by the user (*step 1.b*).

With this approach the application can be fully unaware of the presence of the Encryption Layer. This permits to plug the Encryption Layer in every application without any modification, since the application can continue to use the regular APIs.

Note that the Encryption Layer can also be deployed as a trusted proxy server, which receives all the requests by clients and forwards them (after encryption of the objects) to the Swift storage service.

This architecture presents several advantages compared to the App-aware architecture. First, the proxy server providing EncSwift functionality offers the same APIs provided by Swift. This guarantees that all the existing libraries (e.g., JOSS Java OpenStack Storage [10]) and applications (e.g., Cyberduck [9]) can benefit from EncSwift functionality without any modification to their source code. Second, the trust boundary can be moved by placing the proxy server (Encryption Layer) in different locations. We then have a centralized proxy for an entire organization (i.e., all the users that belong to the same trust boundary) by running it on a trusted server (i.e., inside the organization network) or on a trusted cloud provider, while the data can be outsourced to a lower grade (and generally cheaper) cloud provider. In this case the entire organization is the EncSwift user. The proxy server can also be run directly by the client to keep the trust boundary as close as possible to the user.

The disadvantage of this architecture is that it relies on the presence of an additional trusted server running the Encryption Layer, which may introduce overheads and delays into the system.

4.3 Key Storage in the Cloud

One of the clearest trends of the past few years has been the adoption by users of mobile devices, replacing personal computers as the reference platform for carrying out their daily activities. The management of keys on these kind of devices seems to be unfeasible. Hence, the user has to outsource her own private and public keys, to store securely her own data on the cloud.

OpenStack already offers a component, *Barbican*[3], for the secure storage of private information. Instead of locally managing her RSA and signature encryption keys, the user only has to store a symmetric *Master Key* (MK) used to retrieve the private RSA and signature keys from Barbican. The Master Key is then used to encrypt the RSA and signature keys of the user. Figure 5 illustrates the architecture for key management, which can be integrated with both the App-aware and the Proxy architectures changing the working of *step 1* in the object download process. To download an object, the user then first provides to EncSwift her own Master Key (*step 1.a*). EncSwift uses such a key to decrypt the user's private RSA key retrieved from Barbican (*step 1.b*). Steps 2 and 3 then operate as illustrated for the App-aware and for the Proxy architectures discussed above.

[3]https://wiki.openstack.org/wiki/Barbican

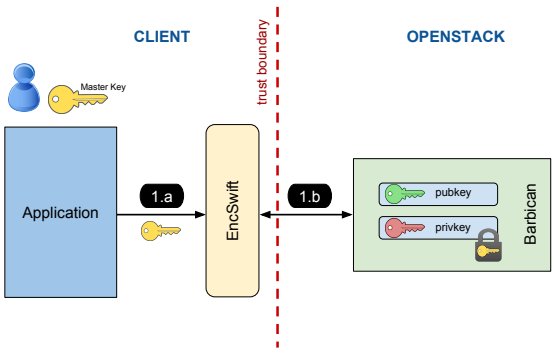


Figure 5: Key storage in the cloud

5. POLICY EVOLUTION

Since we adopt policy-based encryption for access control enforcement, a change in the ACL of a container requires a modification in the encryption key used to protect the objects it contains. Intuitively, the owner of the container would need to download all the objects in the container, re-encrypt them with a new key, and then re-upload the objects to the Swift Storage Service. *Over-Encryption* [7] prevents object re-encryption by requiring the server to add a further layer of encryption. Intuitively, every object has a first layer of encryption (*Base Encryption Layer*, BEL) imposed by the data owner to enforce the initial access control policy and to protect the confidentiality of the objects content from the server. A second layer of encryption (*Surface Encryption Layer*, SEL) is imposed by the server to enforce policy updates. Only users knowing both the BEL and the SEL encryption keys of an object can read its plaintext content. When relying on Over-Encryption, grant and revoke operate as follows.

- To grant a user access to the objects in a container, the data owner needs to share with the user the (BEL and possibly SEL) DEK used to encrypt the objects in the container, by properly creating and storing a new KEK in the user's catalog.

- To revoke a user access to a container, the objects in the container must be encrypted at the SEL level with a new DEK that the revoked user does not know. The data owner then generates a new SEL DEK and the KEKs necessary to authorized users and to the server to retrieve the new SEL DEK. The server re-encrypts all the objects in the container with the new SEL DEK.

We note that, for efficiency reasons, objects inserted into a container after a policy change will be encrypted with a BEL key that reflects the new ACL of the container. This limits the adoption of SEL to only the objects directly involved in a revoke operation. To this purpose, after a revoke operation, the data owner generates a new (BEL) DEK for the container, and inserts into the catalog of non-revoked users the KEKs necessary to retrieve the new DEK.

Since grant operations are easy to manage and do not require Over-Encryption, in the following we will focus on revoke operations.

The implementation of Over-Encryption on Swift objects requires to modify the Swift Storage Service and can oper-

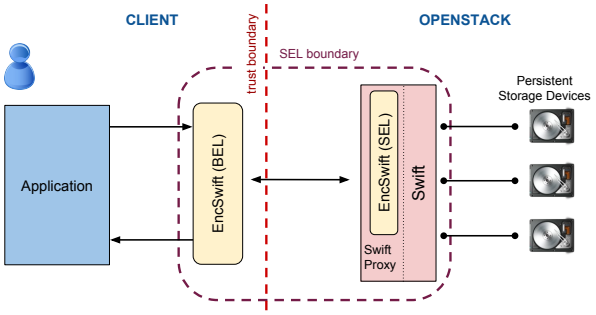


Figure 6: On-the-fly Over-Encryption mode

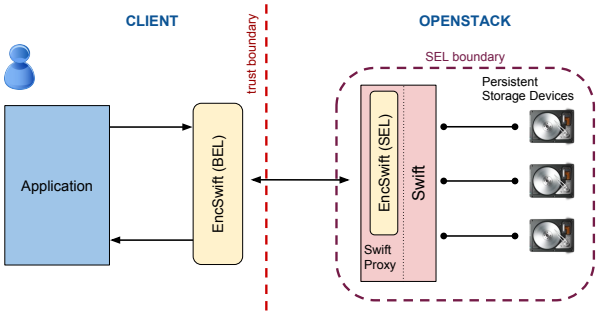


Figure 7: On-resource Over-Encryption mode

ate in three different ways, which provide a different trade-off between download/upload efficiency and data protection. The *on-the-fly* mode applies Over-Encryption only on data in transit between the client and the server. The *on-resource* mode applies Over-Encryption only on data at rest, that is, while objects are stored on server disks. The *end-to-end* mode applies Over-Encryption on data at rest and in transit, combining the protection of the other two modes. In the following, we describe these three Over-Encryption modes.

On-the-fly mode. Over-Encryption is applied on requested objects only when they are downloaded by a user. Objects stored on the server disks are then not over-encrypted, but they are just protected with the BEL key applied by the data owner at upload time. When a user wants to download an object protected by an old BEL DEK (i.e., by a BEL DEK different from the BEL DEK generated by the most recent update of the policy), it is over-encrypted to protect it while in transit from the server to the client. Figure 6 illustrates the on-the-fly Over-Encryption mode. In this figure, the SEL boundary represents when SEL encryption protects objects. If Over-Encryption is needed during a download operation, the server retrieves the KEK necessary to determine the proper SEL DEK to protect the requested object against revoked users. In this mode, the client has to manage a double decryption (at both BEL and SEL).

On-resource mode. If the connection between the users and the server is assumed to be secure while the storage devices may be at risk, Over-Encryption can be directly applied on the objects stored on the server disks. This Over-Encryption mode prevents security breaches, such as the dump of data directly from the server disks and the theft of the server disks themselves, performed by revoked users who gain physical access to the storage server.

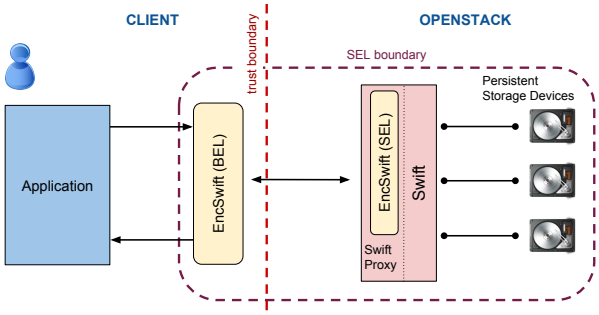


Figure 8: End-to-end Over-Encryption mode

When the ACL of a container is changed and a user is revoked, the server retrieves all the objects in the container, retrieves the SEL DEK (if any) protecting them, and possibly decrypts the objects. It then uses the new SEL DEK received from the owner to encrypt the objects in the container and over-writes the old version of the encrypted objects on disks with the new one. When adopting this mode, Over-Encryption guarantees that the objects are encrypted at SEL only when stored at the server, as illustrated by the dashed box in Figure 7. When a user accesses an object, the server retrieves the SEL DEK protecting the object, decrypts it, and sends the object encrypted only at BEL to the requesting user.

End-to-end mode. The end-to-end Over-Encryption mode applies Over-Encryption on the objects stored on the server disks and keeps the SEL encryption when objects are transmitted to the client. It then combines the protection guarantees offered by the on-the-fly and on-resource modes. In fact, SEL encryption protects objects against revoked users on the server disks and in transit, until they reach the user premises, as illustrated by the SEL boundary in Figure 8.

Like for the on-resource mode, when a user is revoked access to a container, the server re-encrypts at the SEL level all the objects of the container, updating their encrypted representation stored on disks. On the contrary, when a user accesses an object, the server retrieves the object of interest and returns it to the requesting user as it is stored on disk (i.e., without removing SEL encryption layer). Like for the on-the-fly mode, the client has to manage a double decryption (at both BEL and SEL) when downloading objects from the server.

On-resource and end-to-end Over-Encryption modes have the advantage, over the on-the-fly mode, of enjoying a faster response time when objects are downloaded, since they apply SEL encryption at revoke time. On the contrary, the on-the-fly mode introduces two additional operations at download time: an encryption at the server side and a decryption at the client side. The on-resource and end-to-end modes however imply a slower policy update, since revoke operations require to over-encrypt the objects in the revoked container. This could cause a large overhead when the size of the revoked container is relevant. Instead, the overhead of Over-Encryption is distributed among the download requests in the on-the-fly mode, since it is applied on single requested objects. Therefore, as shown by the experimental evaluation in Section 7, depending on ratio between the fre-

bel_key_id	Identifier of the current BEL DEK of the container, used to encrypt new objects uploaded in the container
sel_key_id	Identifier of the SEL DEK of the container; it is empty if Over-Encryption is not necessary for the container

Figure 9: Metadata added to the container header

bel_key_id	Identifier of the BEL DEK used to encrypt the object
sel_key_id	Identifier of the SEL DEK used to encrypt the object in storage; it is empty if the object does not need Over-Encryption or when using on-the-fly Over-Encryption mode

Figure 10: Metadata added to the object header

quency of object downloads and of policy changes, one mode may be more convenient than the other.

6. IMPLEMENTATION

This section describes the major design choices in the implementation of EncSwift.

6.1 Container and Object Headers

To enforce the access control policy, each object and container is associated with BEL and SEL DEKs. Each container is associated with at most one SEL DEK. In fact, every time a revoke operation implies Over-Encryption, a SEL DEK is generated and is associated with the container. SEL KEKs are updated according to policy changes and are always available in the catalog of each authorized user. Differently from SEL, each container may include objects encrypted with different BEL DEKs as they may have been inserted into the container at different times (and hence with different ACLs, as discussed in Section 5). A BEL DEK therefore must be associated also with each object. However, each container is associated with only one current BEL DEK, reflecting the current ACL of the container.

Figures 9 and 10 show the metadata added to the container and object headers, to allow users to retrieve the DEK necessary to encrypt/decrypt each object in the system.

6.2 Encryption Layer APIs

Our application offers a new set of routines that should be used in substitution of the official ones provided by *python-swiftclient* to take advantage of our policy-based encryption functionality. When the final user invokes one of these routines, the Encryption Layer in EncSwift, which is in charge of the dialog with the different modules of the OpenStack environment, manages it. In the following, we refer our discussion to the Encryption Layer APIs of the on-the-fly mode, with the note that the on-resource and end-to-end modes operate in a similar way.

Create User Method

To create a new user, the Encryption Layer invokes the Keystone standard *create_user* method and generates a new RSA key pair and a new signature key pair, and stores them both on the local storage and on Barbican (encrypting the RSA and signature private keys with the user's Master Key). Then, the Encryption Layer communicates with Swift to create the meta-container with the standard *put_container* method. It then sets the ACL of the meta-container to include the user only, using the traditional *post_container* method. Finally, the catalog is generated and stored in the meta-container.

Put Container Method

To create a new container, the Encryption Layer generates a new BEL DEK and produces the corresponding set of KEKs according to the ACL of the container, following the approach illustrated in Section 3. Clearly, when a container is created, no SEL is required since the container (and its objects) are protected only with BEL encryption. The container is then inserted into Swift using the traditional *put_container* method, properly initializing the metadata in the header of the container (Figure 9).

Put Object Method

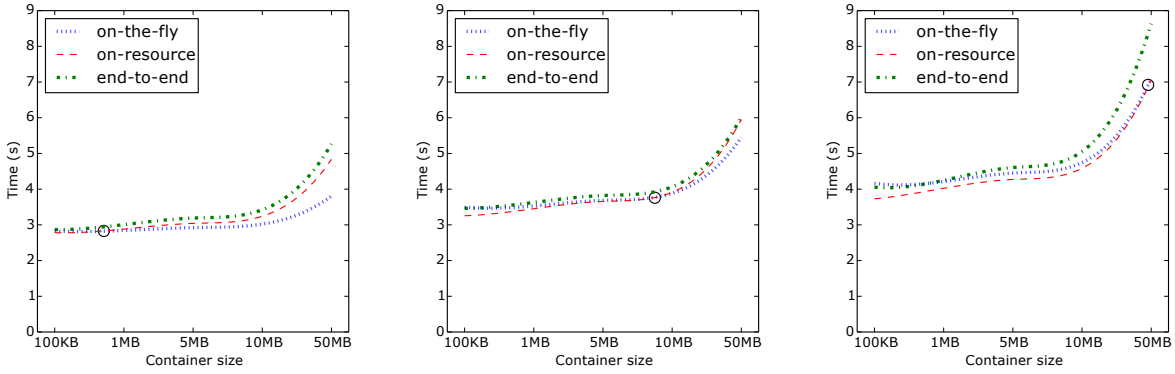
To insert a new object into a container, the traditional *put_object* method is modified to guarantee that the object is uploaded in encrypted (in contrast to plaintext) form, using a key that enforces the container ACL. To this aim, the Encryption Layer retrieves, from the header of the container, the identifier of the BEL DEK. If the user who invokes the put object method is authorized for the container (i.e., she appears in the ACL), the Encryption Layer asks Swift for the user's catalog, to retrieve the KEKs necessary to obtain the BEL DEK associated with the container. The Encryption Layer then uses such a BEL DEK to encrypt the object (Section 3) and puts it in the container, invoking the traditional *put_object* method.

Get Object Method

To retrieve an object from a container, it is first necessary to retrieve the identifier of the BEL DEK and SEL DEK used for the object. The Encryption Layer then asks Swift to retrieve the stored object, using the traditional *get_object* method. The Swift server then verifies whether the object has to be protected with SEL encryption. If the **sel_key_id** of the container is empty or if the **bel_key_id** of the container is the same as the **bel_key_id** of the object, Over-Encryption is not necessary. In this case, either the policy of the container has never been updated, or the container has not been subject to revoke operations after the insertion of the object. The object is then returned to the client. At this point, two (or four) decryptions occur at the client side: one to obtain the DEK (at BEL and, if necessary, also at SEL) from the KEKs, and one to decrypt the object (at BEL and, if necessary, also at SEL).

Post Container Method

To grant or revoke users access to a container, we use the *post_container* method, which is used to change the meta-information associated with a container and then also its header.



(a) $\gamma = 0.5$ (b) $\gamma = 1.0$ (c) $\gamma = 1.5$

Figure 11: Comparison of the performance in serving a single request in the three Over-Encryption implementation modes

When a user invokes the *post_container* method, the Encryption Layer first checks if the post operation requests to change the ACL (and whether it is a grant or a revoke operation). If this is the case, the Encryption Layer retrieves the container header, to obtain the ACL and the identifier of the current BEL DEK and of the SEL DEK. When a user is added to a container ACL, the Encryption Layer computes the BEL KEKs and SEL KEKs necessary to the granted user to access the BEL DEK and SEL DEK, respectively. These KEKs are then stored in the catalog of this user. When a user is removed from a container ACL, Over-Encryption is required. Hence, the Encryption Layer creates a new SEL DEK (possibly substituting the old one) and updates the catalogs of the users in the current ACL with the new KEKs (possibly removing the KEKs used to obtain the old SEL DEK). In the on-resource and end-to-end modes, the server also re-encrypts the objects with the new SEL DEK.

6.3 Catalog Management Service

To guarantee the security and the consistency of users catalogs, they can be updated only by users with administrative privileges. Since also users who do not have these privileges may need to update catalogs (i.e., when granting other users access to their containers), we introduce an always-listening *Catalog Management Service*. This service provides an API that users can invoke to update catalogs when a container ACL is changed. The Catalog Management Service checks if the request is valid and guarantees a correct update of all the catalogs of the users involved in the grant/revoke operation. In this way, there is only one authorized entity that can change the users' catalogs.

When a user creates a container, she also invokes the Catalog Management Service API, providing the KEKs that must be inserted into the catalogs of the users in the container ACL.

Since the version of Keystone that we used for our implementation requires administrative privileges to obtain the user ID that corresponds to a username and vice versa, and Swift ACLs include user IDs, our Catalog Management Service also converts user IDs into usernames and vice versa.

6.4 Swift Middleware Pipeline

To apply Over-Encryption it is also necessary to modify the server-side Swift service, which is based on *WSGI*, Web Server Gateway Interface (a modular interface between the web server and the web application) and on Paste Python Framework. WSGI permits to define a pipeline where several components (the middleware) process and modify the request before it reaches the main web server component (e.g., the Proxy node), and the response they get from it. To add Over-Encryption functionality to Swift, we introduced two new components, the *key_master* and the *encrypt* components, in the Proxy node pipeline. The key_master component is in charge of retrieving the correct DEKs, while the encrypt component applies SEL encryption before returning the object to the client. Including these components in the Proxy node pipeline has the advantage of leveraging the work done by the components preceding it in the pipeline (e.g., authentication and request validation).

7. EXPERIMENTAL EVALUATION

A set of experiments were executed to compare the performance of the Over-Encryption modes. The experiments have been run for the client on a machine with Linux Ubuntu 16.04 LTS, Intel i7-4770K, 3.50 GHz, 4 cores. For the server, we used an Amazon EC2 m4.large instance, with 4 CPUs and 8 GB of RAM. The client was connected to the Internet with a symmetric 100 Mbps connection.

To evaluate the profile of the modes we considered a variety of scenarios with a different ratio between the amount of data accessed in the period between two policy updates and the overall size of resources, represented by parameter γ. We can then assess the performance representing the overall time required for the joint execution of a policy update and the access to the over-encrypted data. The time experimentally observed is well approximated by a function $f(\gamma) = a + \gamma d$, where a is the time spent by a *post_container* request that introduces an ACL change and d is the total time necessary to download all the objects in the container.

Figure 11 illustrates the time required for the policy update and object access for the three Over-Encryption modes varying the container size, and considering three different values for parameter γ (0.5, 1, and 1.5). In our experiments

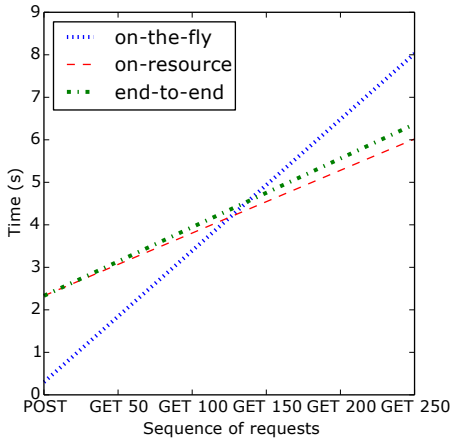


Figure 12: Comparison on the performance for serving multiple requests in the three Over-Encryption implementation modes

we considered 100 objects of uniform size in the container. In all the considered configurations, the time naturally increases with the size of containers in all the three Over-Encryption modes. The component that dominates the execution time is the time required for the interaction between client and server for small containers, and the data transfer for large containers. Encryption and decryption operations have a negligible impact.

As visible from the graphs in Figure 11, the end-to-end mode presents the same trend as the on-resource mode, but shifted up. This is due to the fact that the two modes require the evaluation of exactly the same operations, with the only difference that SEL encryption and decryption operate at the client side in the end-to-end mode and at the server side in the on-resource mode. Since the server has faster access to storage, the on-resource mode is more efficient.

The on-the-fly and the on-resource modes show instead a different behavior and the relationship between their costs changes varying γ. When γ is small (as in Figure 11(a) where $\gamma = 0.5$, meaning that half of the objects in the container are accessed by the client after a policy update), the intersection between the on-the-fly and on-resource curves is on the left (this break-even point is circled in the graphs). This means that the two modes offer the same performance only for small containers. This is due to the fact that ACL updates dominate the total cost and the on-resource mode has to manage more operations. When each object in the container is dowloaded once after the policy update (as in Figure 11(b) where $\gamma = 1$), the intersection between the two lines corresponds to a container size of about 9MB. The on-the-fly mode shows an advantage with larger containers since, for each ACL change, it requires one key exchange only, whereas the on-resource mode must change the encryption keys of all the objects in the container. When γ is larger (as in Figure 11(c) where $\gamma = 1.5$) the break-even point moves to the right, since *get_object* operations cause a higher overhead in the on-the-fly mode.

Figure 12 illustrates the total time necessary to each of the three Over-Encryption modes to process a *post_container* request that updates the policy over a 100MB container, followed by a sequence of 250 *get_object* requests, each for an object of size 1MB. The end-to-end and on-resource modes show an immediate overhead on the initial *post_container* due to the re-encryption of the stored objects, while the on-the-fly mode spends a negligible amount of time to exchange keys with other users. We note that *get_object* requests are slower in the on-the-fly mode. The break-even point in this scenario among all the considered Over-Encryption modes is reached after about 130 *get_object* requests. After such a number of *get_object* requests, the end-to-end and the on-resource modes become more efficient than the on-the-fly mode. The experiments confirm that the on-the-fly mode is preferable when the frequency of policy updates is great compared to the frequency of access requests, whereas the end-to-end and on-resource modes have an advantage when policy updates are relatively rare compared to object access requests. The on-resource mode shows a small performance advantage compared to the end-to-end mode, which can be preferred for the weaker trust assumptions it imposes on the system.

8. RELATED WORK

Due to the rapid growth of cloud storage services, a significant amount of research has investigated the development of encryption techniques to protect confidentiality when moving data to the cloud [8, 13, 16]. The proposal developed in this paper focuses on the Over-Encryption technique [3, 5, 6, 7], which has been adapted to operate in the OpenStack Swift scenario. In fact, the original work [7] targets a generic provider (in contrast to the realization with a specific technology like in this paper), with a generic interface for the management of resources and keys. Also, the original proposal assumes the presence of a single data owner. An extension of the original proposal to a multi-owner scenario has been presented in [6] and is based on a Diffie-Hellman scheme. Again, it considers a generic provider. The implementation of Over-Encryption in OpenStack Swift has also been considered in [3], where approaches for the enforcement of revoke operations have been investigated. The work in [3] is then complementary to the investigation in this paper, which explores different alternatives for the architecture, associated with distinct levels of transparency for the application, and integrate different options for the management of policy updates.

A recent solution relying on policy-based encryption for access control enforcement, while efficiently supporting revoke operations has been proposed in [4]. This approach enforces revoke operations by applying a sequence of encryption steps, which guarantee that even if a small portion of a resource is not accessible, the entire resource cannot be decrypted. Hence, re-encryption of a small portion of a resource is sufficient to revoke a user access to the whole resource. This paper, aiming at deployability with current cloud technology and open source efforts, investigates instead the realization of Over-Encryption as a protection layer on a resource (object in OpenStack) as a whole.

Being a central piece of modern cloud infrastructures, OpenStack Swift has recently been widely investigated. In [2] some of the most critical Swift security aspects are analyzed in detail. The framework in [1] proposes to encrypt objects in Swift, but do not address the problem of efficiently managing revoke operations.

Other proposals focused on the use of client-side encryption to protect data in the cloud (e.g., [11, 17]). *Trustore* [17] defines a file system that is backed by Amazon S3 and encrypts files before outsourcing them. This proposal, however, does not consider the use of ACLs for sharing resources among users. *CloudaSec* [11] is a framework that handles secure data sharing in the cloud. This approach enforces revoke operations by limiting access to keys from revoked users. A number of other proposals, such as ESPRESSO [12], use instead server-side encryption to protect "data at rest" (i.e., from attacks that are able to gain access to the physical storage devices). These proposals do not consider the use of ACLs to share access to resources and encrypt data with keys that are never exposed to users.

9. CONCLUSIONS

We presented the realization of an encryption solution, and of Over-Encryption for the management of policy updates, in OpenStack Swift. Our investigation offers an analysis of different implementation strategies for key management, policy enforcement, and policy updates. Leveraging the openness and modular architecture of OpenStack Swift, our EncSwift tool provides a convenient approach for flexible data sharing with current cloud technology. The tool has also been designed with extensibility in mind to enable its adaptation and extension to possible evolutions of OpenStack, such as the ongoing efforts on Swift at-rest encryption [15]. Our proposal contributes then to the support of effective and practical data protection solutions in real-world cloud scenarios. Also, we believe that our analysis can be of benefit to other researchers and practitioners interested in developing efficient data protection and sharing solutions for the cloud.

10. ACKNOWLEDGMENTS

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

11. REFERENCES

[1] H. Albaroodi, S. Manickam, and M. Anbar. A proposed framework for outsourcing and secure encrypted data on OpenStack object storage (Swift). *Journal of Computer Science*, 11(3):590–597, 2015.

[2] H. Albaroodi, S. Manickam, and P. Singh. Critical review of OpenStack security: Issues and weaknesses. *Journal of Computer Science*, 10(1):23–33, 2014.

[3] E. Bacis, S. De Capitani di Vimercati, S. Foresti, S. Paraboschi, M. Rosa, and P. Samarati. Access control management for secure cloud storage. In *Proc. of the 12th International Conference on Security and Privacy in Communication Networks (SecureComm 2016)*, Guangzhou, China, October 2016.

[4] E. Bacis, S. De Capitani di Vimercati, S. Foresti, S. Paraboschi, M. Rosa, and P. Samarati. Mix&Slice: Efficient access revocation in the cloud. In *Proc. of the 23rd ACM Conference on Computer and Communications Security (CCS 2016)*, Vienna, Austria, October 2016.

[5] S. De Capitani di Vimercati, S. Foresti, S. Jajodia, G. Livraga, S. Paraboschi, and P. Samarati. Enforcing dynamic write privileges in data outsourcing. *Computers & Security*, 39:47–63, November 2013.

[6] S. De Capitani di Vimercati, S. Foresti, S. Jajodia, S. Paraboschi, G. Pelosi, and P. Samarati. Encryption-based policy enforcement for cloud storage. In *Proc. of the 1st ICDCS Workshop on Security and Privacy in Cloud Computing (SPCC 2010)*, Genova, Italy, June 2010.

[7] S. De Capitani di Vimercati, S. Foresti, S. Jajodia, S. Paraboschi, and P. Samarati. Encryption policies for regulating access to outsourced data. *ACM Transactions on Database Systems (TODS)*, 35(2):1–46, Article 12, April 2010.

[8] S. De Capitani di Vimercati, S. Foresti, G. Livraga, and P. Samarati. Practical techniques building on encryption for protecting and managing data in the cloud. In P. Ryan, D. Naccache, and J.-J. Quisquater, editors, *Festschrift for David Kahn*. Springer, 2016.

[9] Z. Y. Duan and Y. Z. Cao. The implementation of cloud storage system based on OpenStack Swift. *Applied Mechanics and Materials*, 644:2981–2984, September 2014.

[10] Javaswift JOSS. http://joss.javaswift.org.

[11] N. Kaaniche, M. Laurent, and M. El Barbori. Cloudasec: A novel public-key based framework to handle data sharing security in clouds. In *Proc. of the 11th International Conference on Security and Cryptography (SECRYPT 2014)*, Vienna, Austria, August 2014.

[12] S. Kang, B. Veeravalli, and K. M. M. Aung. ESPRESSO: An encryption as a service for cloud storage systems. In *Proc. of the 8th IFIP WG 6.6 International Conference on Autonomous Infrastructure, Management, and Security (AIMS 2014)*, Brno, Czech Republic, June 2014.

[13] F. Kerschbaum. Client-controlled cloud encryption. In *Proc. of 21st the ACM Conference on Computer and Communications Security (CCS 2014)*, Scottsdale, Arizona, USA, November 2014.

[14] OpenStack Project. http://www.openstack.org.

[15] J. Richling and A. Cole. At-Rest Encryption. http://specs.openstack.org/openstack/swift-specs/specs/in_progress/at_rest_encryption.html.

[16] P. Samarati and S. De Capitani di Vimercati. Cloud security: Issues and concerns. In S. Murugesan and I. Bojanova, editors, *Encyclopedia on Cloud Computing*. Wiley, 2016.

[17] J. Yao, S. Chen, S. Nepal, D. Levy, and J. Zic. Truststore: Making Amazon S3 trustworthy with services composition. In *Proc. of the 10th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid 2010)*, Melbourne, Australia, May 2010.