Access Control Management for Secure Cloud 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. With the widespread success and adoption of cloud-based solutions, we are witnessing an ever increasing reliance on external providers for storing and managing data. This evolution is greatly facilitated by the availability of solutions - typically based on encryption - ensuring the confidentiality of externally outsourced data against the storing provider itself. Selective application of encryption (i.e., with different keys depending on the authorizations holding on data) provides a convenient approach to access control policy enforcement. Effective realization of such policy-based encryption entails addressing several problems related to key management, access control enforcement, and authorization revocation, while ensuring efficiency of access and deployment with current technology. We present the design and implementation of an approach to realize policy-based encryption for enforcing access control in OpenStack Swift. We also report experimental results evaluating and comparing different implementation choices of our approach.

1 Introduction

Cloud technology is increasingly becoming a central component for storing or processing data. Such growing adoption and success of cloud-based solutions is due to the considerable obvious benefits they provide in terms of reliability, scalability, elasticity, efficiency, and economic cost. This adoption would further accelerate in the presence of robust solutions guaranteeing effective control by data owners over the data they outsource to cloud service providers.

A promising solution for providing data protection and maintaining control in the hand of data owners is encryption, with data encrypted before being outsourced to the external cloud service provider. The first obvious benefit of using encryption when outsourcing data is that data are kept unknown to the provider hosting them. Also, encryption provides the ability to realize an approach where the evaluation of the policy and user authentication are separate from the management of the physical access to the data. This also ensures the protection of data confidentiality against adversaries who may have access to the physical representation or who may be able to subvert an access control service managed by the cloud service provider. Another important benefit provided by data encryption is that it enables effective enforcement of access control. In fact, data can be encrypted with different keys, depending on the authorizations holding on them, and keys shared with users according to authorization (*policy-based encryption* [7]). This policy-based encryption translates the access control policy into an equivalent *encryption policy* which provides self-protection and effective access control enforcement on the outsourced data.

One of the complicating aspects in the management of policy-based encryption relates to the enforcement of possible changes to the access control policy, and in particular revocation of authorizations. When resource maintenance is decoupled from access control thanks to the use of encryption, revocation cannot be simply managed by dropping access to the encryption key (as done in other scenarios). The revoked user can, in fact, have maintained local copies of the keys, and if the layer of protection is not refreshed, the user could still be able to pass the encryption wrap and access objects for which she does not have authorization anymore. On the other hand, changing the key and re-encrypting objects affected by revocation would entail download and re-upload operations by owners, which could become cumbersome and affect the performance of the system. The solution that was proposed to this problem in [7] assumes the introduction of *over-encryption*, based on the application by the server of an additional layer of encryption (operating on the object already encrypted by the data owner) with a key not accessible by the revoked user, thus adapting the encryption on objects to the new state of the access control policy.

Policy-based encryption for providing self-enforcement of the access control policy and over-encryption for supporting policy changes result particularly appealing and promising. However, their integration and deployment in available cloud storage systems requires addressing several problems, including: the support for co-existence of several data owners in a single system, the realization of key management solutions to enable users to access keys used for objects for which they have authorizations, and the implementation of policy-based encryption and over-encryption functionality with services supported by the cloud service providers. In this paper, we investigate all these issues and illustrate the realization of policy-based encryption and over-encryption in the context of OpenStack Swift. OpenStack [16] represents today the reference platform for the cloud [19], and is receiving significant attention by the industrial community, and Swift is the OpenStack's object storage system. Swift exhibits features that are shared by most object storage solutions for the cloud, like Amazon S3. In this paper, we illustrate how policy-based encryption can be realized building on the OpenStack Swift module. We also investigate how policy changes can be enforced implementing over-encryption in Swift. For over-encryption, in particular, we investigate different implementation alternatives, which can be suitable for different scenarios, depending on the frequency of access requests and policy changes. The contribution of the paper is therefore twofold. First, it provides an effective design and implementation of policy-based encryption and over-encryption, which can be adopted by others and see immediate deployment in current cloud storage solutions. Second, our extensive experimental evaluation of different design choices can provide precious observations for such adoption, enabling the tuning of the implementation depending on the characteristics of the considered scenario.

Outline. The remainder of this paper is organized as follows. Section 2 describes some basic concepts as well as the scenario and the problem considered. Section 3 illustrates how policy-based encryption can be realized in Swift. Section 4 shows how policy changes can be enforced and describes different options for the implementation of over-encryption. Section 5 presents experimental results. Section 6 discusses related work. Finally, Section 7 presents our conclusions.

2 Basic Concepts

We consider a scenario where users wish to outsource data to an external cloud service provider (CSP) and selectively share their data with others. Different data (owned by the same user) may be accessible by different sets of users. Every data owner has an access control policy specifying authorizations on her data.

We assume that the CSP is based on the OpenStack framework, which includes the Swift module, an object storage service allowing users to store and access data in the form of *objects* (i.e., each resource, such as a file, uploaded on Swift is an object). Swift organizes objects in *containers*, which are user-defined storage areas containing sets of objects. Containers are organized in *tenants*, which are sets of containers. Each tenant is usually assigned to an organization. Swift enforces discretionary access control restrictions over the objects it stores by associating a read access control list and a write access control list with each container and tenant in the system. These access control lists identify the users who can read and write the container/tenant. To enforce access control restrictions, Swift relies on *Keystone* for users authentication. Keystone is an OpenStack component acting as identity server, which provides a central directory of users mapped to the OpenStack services they can access.

We assume the cloud service provider to be *honest-but-curious*, that is, trusted to correctly manage the data (i.e., trustworthy) but not trusted for accessing the content of objects. Consistently with our focus on data confidentiality, in this paper we are concerned with the representation and enforcement of an access control policy regulating read access to objects. We note however that our approach can be extended to the consideration of write authorizations [5]. In the following, $acl(o)$ denotes the read access control list of object o and $\mathcal{A}_u$ is the set of read access control lists defined by user u for her objects. Figure 1(a) illustrates an example of authorization policy defined by user Alice. In this example, we assume that there are three users, Alice (A), Bob (B), and Dave (D), and four objects (o_1, o_2, o_3, and o_4) owned by Alice. In the matrix in Figure 1(a), entry $[u,o]$ has value 1 if u is authorized to read o (i.e., $u \in acl(o_i)$) and 0 if u is not authorized to read o (i.e., $u \notin acl(o_i)$).

$$\begin{array}{c|c|c|c|c|}
 & \mathbf{o_1} & \mathbf{o_2} & \mathbf{o_3} & \mathbf{o_4} \\
\hline
\mathbf{A} & 1 & 1 & 1 & 1 \\
\hline
\mathbf{B} & 1 & 1 & 1 & 1 \\
\hline
\mathbf{D} & 0 & 1 & 0 & 1 \\
\hline
\end{array}$$

Authorization policy$\mathcal{A}_\mathbf{A}$
(a)

$$m_A \rightsquigarrow \{k_1,k_2,k_3,k_4\}$$
$$m_B \rightsquigarrow \{k_1,k_2,k_3,k_4\}$$
$$m_D \rightsquigarrow \{k_2,k_4\}$$

$$\begin{array}{c|c|c|c|c|}
 & \mathbf{o_1} & \mathbf{o_2} & \mathbf{o_3} & \mathbf{o_4} \\
\hline
\mathbf{key} & k_1 & k_2 & k_3 & k_4 \\
\hline
\end{array}$$

Policy-based encryption $\mathcal{E}_\mathbf{A}$
(b)

Fig. 1: An example of authorization policy defined by user Alice (a) and corresponding policy-based encryption (b)

Our work is based on the policy-based encryption and over-encryption approach proposed in [7, 8], and aims at their representation and enforcement with Swift, which also require some re-definition and adjustment of these concepts. Essentially, each user is associated with a symmetric key, and each object is encrypted using a symmetric key that depends on the access control policy. Keys are organized in such a way that a user u can derive (via public tokens), all and only the keys of the objects o_i she is authorized to access (i.e., $u \in acl(o_i)$). Policy updates, which would require re-encryption of an object, are enforced by super-imposing a second layer of encryption on the encrypted object itself. Every object can then have a first layer of encryption (*BEL, Base Encryption Layer*) imposed by the data owner for protecting the confidentiality of the data from unauthorized users as well as from the CSP, and a second layer of encryption (*SEL, Surface Encryption Layer*) applied by the CSP for protecting the object from users who are not be authorized to access the object but who might know the underlying BEL key. A user will be able to access an object only if she knows both the SEL key and the BEL key with which the object is encrypted. In the following, we use notation $\mathcal{E}_u$ to denote the policy-based encryption equivalent to the authorization policy $\mathcal{A}_u$ defined by user u. Figure 1(b) illustrates the policy-based encryption equivalent to the authorization policy in Figure 1(a). In this figure, keys m_A, m_B, m_D are the symmetric keys of the users and keys k_1, k_2, k_3, k_4 are the symmetric keys used to encrypt the objects. Notation $m_x \rightsquigarrow k_y$ represents the fact that key k_y is derivable from key m_x. In the remaining sections, we first describe how a policy-based encryption can be realized in Swift (Section 3), and then illustrate how to enforce policy updates (Section 4).

3 Access Control Enforcement in Swift

Our approach translates the authorization policy defined by a user into a policy-based encryption that relies on the use of different keys and ad-hoc structures supporting the client-based Swift encryption. In this section, we describe such keys and ad-hoc structures (which are stored as traditional Swift objects), and then illustrate how policy-based encryption can be implemented.

3.1 Keys and User-Based Repositories

Our approach is based on the definition and management of different keys. There are (symmetric) keys associated with objects for objects' encryption (enforcing the self-protection mentioned in the introduction). Also, each user is associated with a (symmetric) key as well as with two pairs of asymmetric keys to support identity management and signature, respectively. Finally, authorizations are realized by encrypting object keys with user keys. This allows users to retrieve the key of objects they are authorized to access, providing the same functionality that public-tokens provided in [7, 8].

We describe the different keys and their characteristics and functionality in the following.

Data Encryption Key (DEK) $\mathbf{k_i}$. Every object o_i is protected by symmetric encryption using a DEK k_i. Each DEK k_i has a given size, is associated with an encryption algorithm, and has an identifier, denoted $id(k_i)$, that identifies the key among all the keys used in the system.

Master Encryption Key (MEK) $\mathbf{m_u}$. Every user u has a personal symmetric master encryption key m_u. The knowledge of this key permits to access, directly or indirectly, all the objects that user u is authorized to see. Given the user identity loss that would derive from a compromise of the MEK, it is assumed that the user keeps the MEK only on the client-side, never exposing it to the server or to other users.

User encryption key pair $\langle \mathbf{p_u}, \mathbf{s_u} \rangle$. Each user u is associated with an asymmetric key pair $\langle p_u, s_u \rangle$ for encryption (our implementation adopts RSA). As we show later on, the availability of asymmetric cryptography supports the realization of a cooperative cloud storage service, where each user may make her objects available to other users. Note that in most application domains, the correspondence between a user identity and a public key is supported by certificates issued by a trusted Certification Authority. Swift can instead benefit from the availability of Keystone, which already centralizes the management of user identities, and the public key is assumed to be available in the user profile managed by Keystone.

User signing key pair $\langle \mathbf{sp_u}, \mathbf{ss_u} \rangle$. Each user u is associated with an asymmetric key pair $\langle sp_u, ss_u \rangle$ for signing messages (our implementation adopts EC-DSA). The reason for having a signing key pair is that it is common in security systems to separate the encrypting and signing identities. This improves security and flexibility, giving the option to use a dedicated cryptographic technique for each function. Signatures are used to guarantee the integrity of objects and of the information that users adopt for deriving the DEKs. Like for asymmetric encryption, the public key for signatures is also stored in the Keystone profile of users.

Key Encryption Key (KEK). A KEK is at the basis of the mechanism that translates the access control policy defined by a user into an equivalent policy-based encryption. Intuitively, a KEK is the encryption of a DEK that a user can extract using a secret (key) that only she knows. For each container that a user is authorized to access, there is therefore a KEK that the user can decrypt to obtain the DEK used for encrypting the objects in the container. As we will see in the following sub-section, there are two variants of KEKs, depending on the cryptographic technique used to protect them: symmetric KEKs, encrypted with the MEKs of users, and asymmetric KEKs, encrypted with the public keys of users. The KEKs that allow a user u to derive the keys of the objects she is authorized to access are stored in a user-based repository, denoted $\mathcal{R}_u$. Each KEK is characterized by the following information: a KEK identifier, the identifier of the protection key, the identifier of the encrypted key, a timestamp, the identifier of the creator (only for asymmetric KEKs), an authentication code, and the encrypted key. The authentication code is used to verify the integrity of a KEK and is generated with the symmetric key of the user who creates the KEK (in case of symmetric KEK) or with the private signing key of the creator (in case of asymmetric encryption). Functions are available that allow the user to extract from her repository the KEK associated with a given protected key identifier.

The identifier of the DEK used to protect an object is maintained in the descriptor of the object itself. Such a piece of information is needed, whenever a user accesses an object, to retrieve the right KEK that allows the user to derive the corresponding DEK. Analogously, the descriptor of a container includes the identifier of the key to be used to encrypt the objects that will be inserted in the container. At initialization time, the key identifier in the descriptor of the objects stored in a container coincides with the key identifier in the container descriptor. As we will discuss in Section 4, due to policy changes, the key associated with a container may change and objects in the container may still be protected with a previous container key.

3.2 Policy-Based Encryption

All users in the system can define an access control policy for the objects they own. We now describe how the authorization policy $\mathcal{A}_u$ defined by user u is translated into an equivalent policy-based encryption $\mathcal{E}_u$ using the keys illustrated in the previous section.

User u creates as many containers $C_1, \ldots, C_m$ as needed and, for each of them, creates a DEK k_i, $i = 1, \ldots, m$, using a robust source of entropy. Consistently with Swift working, we assume that all objects in a container have the same acl. User u then encrypts all objects in a container C_i with the DEK k_i of the container and stores them in C_i, which will have therefore the same acl for all the objects in it. Each DEK k_i is encrypted with the MEK m_u of the user who created the container and the resulting KEK is stored in the user's repository $\mathcal{R}_u$. For each user u_j in the acl corresponding to container C_i, user u

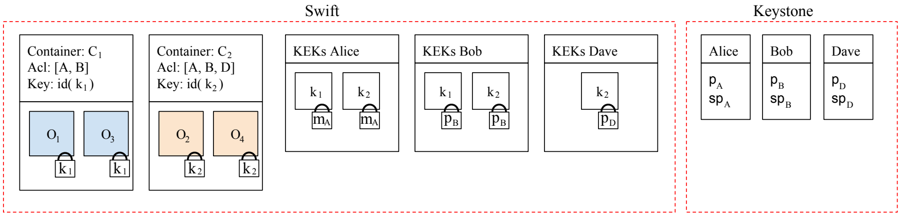

Fig. 2: Policy-based encryption $\mathcal{E}_A$ equivalent to the authorization policy $\mathcal{A}_A$ in Figure 1(a)

encrypts DEK k_i with u_j's public key p_{u_j} and signs it using ss_u, thus producing an asymmetric KEK usable by u_j. This KEK is stored in u_j's repository $\mathcal{R}_{u_j}$.

Example 1. Consider the authorization policy of Alice in Figure 1(a). Figure 2 shows how this policy is translated into an equivalent policy-based encryption. Alice creates two containers C_1 and C_2 and stores objects o_1 and o_3 both encrypted with key k_1 in C_1, objects o_2 and o_4 both encrypted with k_2 in C_2. She then creates her KEKs as well as the KEKs that Bob and Dave can use to access the objects for which they are authorized. In particular, Alice encrypts DEKs k_1 and k_2 with her MEK m_A and stores the resulting KEKs in her repository $\mathcal{R}_A$. Then, she encrypts DEK k_1 with Bob's public key p_B and DEK k_2 with public keys p_B and p_D of Bob and Dave, respectively. The resulting KEKs are stored in repositories $\mathcal{R}_B$ and $\mathcal{R}_D$, respectively. The figure also illustrates the profiles of Alice, Bob, and Dave managed by Keystone. These profiles contain the public keys of the users.

When a user u_j wishes to access an object o_l, the object descriptor is first accessed to retrieve the identifier of the DEK used to encrypt o_l. This identifier is then used to retrieve the corresponding KEK from repository $\mathcal{R}_{u_j}$ and then derive the DEK k_l. Derivation will require user u_j either to use her own MEK m_{u_j} (for symmetric KEK), or to apply the private encryption key s_{u_j} (for asymmetric KEK). 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 in the repository by a symmetric KEK built using the user own MEK. For instance, suppose that Bob requires access to object o_1. Bob first retrieves from the descriptor of object o_1 the identifier $id(k_1)$ of DEK k_1. Then, it retrieves from $\mathcal{R}_B$ the corresponding KEK, decrypts it using his private key s_B and uses the retrieved DEK for decrypting o_1. Furthermore, Bob replaces the original asymmetric KEK with a symmetric KEK obtained by encrypting k_1 with his master key m_B.

When a new object o_l is inserted into a container C_i, user u retrieves the descriptor of the container and looks for the identifier $id(k_i)$ of the corresponding DEK k_i. The user will then look in her repository $\mathcal{R}_u$ for the KEK associated with $id(k_i)$ and will extract the corresponding DEK. The DEK will be used to encrypt object o_l that will be given to Swift and DEK $id(k_i)$ will be inserted into the object descriptor. For instance, suppose that Alice inserts a new object o_5 in C_2. Since the DEK associated with C_2 is k_2, Alice encrypts o_5 with k_2, inserts $id(k_2)$ in the descriptor of o_5, and stores the encrypted version of o_5 in C_2.

4 Policy Updates

Since the authorization policy regulating access to objects in Swift is enforced through a policy-based encryption, every time the authorization policy changes, also the encryption policy needs to be re-arranged accordingly. Updates to the authorization policy include the insertion and deletion of users, objects, and authorizations. The insertion of a user requires the generation of her master key, user encryption key pair, and signing key pair, and the insertion of her public keys in Keystone. The removal of a user requires only the removal from Keystone of her public (encryption and signing) keys. The removal of an object instead requires its deletion from the container including it. We then focus on granting and revoking authorizations, and on the insertion of new objects. For simplicity, but without loss of generality, we consider policy updates that involve a single user u_i and a single container C (the extension to a set of users and of containers is immediate).

In the remainder of this section, we first illustrate how policy updates can be realized, and then discuss different alternatives for the practical implementation in Swift of the over-encryption requested for their enforcement.

4.1 Enforcement of Policy Updates

We now illustrate how granting and revoking authorizations as well as the insertion of a new object with its authorization policy can be enforced. Recall that authorization policies operate at the granularity of container. Then, grant and revoke operations modify the set of users authorized to access a container C, and hence all the objects that it stores. Also, the insertion of an object in a container implies that it inherits the container acl.

Grant authorization. If user u grants u_i access to container C (and hence to the content of all its objects), she simply needs to create an (asymmetric) KEK enabling u_i to derive the DEK k of the container and to store it in the repository $\mathcal{R}_{u_i}$ of user u_i. For instance, with reference to the authorization policy in Figure 1(a), to grant Dave access to container C_1, Alice needs to create a KEK enabling Dave to derive k_1.

Revoke authorization. If user u revokes from u_i access to container C (and hence to all its objects), it is not sufficient to delete the KEK that allows u_i to derive the DEK k of the container, as the revoked user u_i may have accessed the KEK before being revoked and may have locally stored its value. A straightforward approach to revoke user u_i access to container C consists in replacing

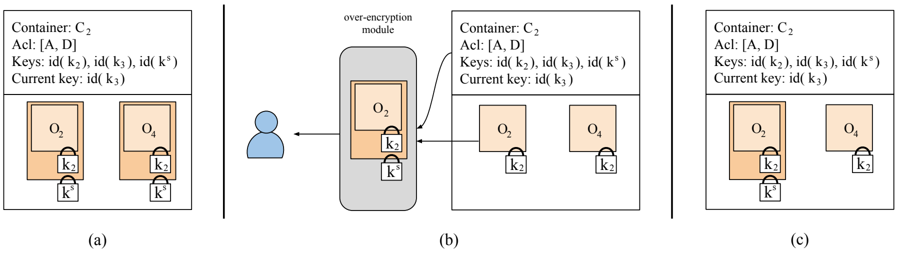

Fig. 3: An example of implementation of a revoke operation using immediate (a), on-the-fly (b), and opportunistic (c) over-encryption

the DEK of the container with a new key k_{new}. However, this would require the owner u of the container to download from the server all the objects in C, decrypt them with the original DEK k, encrypt them with the new DEK k_{new}, and then re-upload the encrypted objects, together with the KEKs necessary to authorized users to derive k_{new}. This would cause a significant performance and economic cost to user u. To limit such an overhead, we adopt over-encryption (Section 2). Hence, when a user u revokes from another user u_i the authorization to access the objects in a container C, u updates C's acl and asks the storing server to over-encrypt the objects in C with a SEL key k^s that only non-revoked users can derive. Each container is then associated with a DEK k at the BEL enforcing the initial authorization policy, and possibly also with a DEK k^s at the SEL enforcing revocations. Also, there is a KEK for each user initially authorized for C enabling her to compute k, and a KEK for each non-revoked user enabling her to compute k^s. For instance, consider the authorization policy in Figure 1(a), and assume that Alice wants to revoke from Bob the access to C_2. As illustrated in Figure 3(a), objects o_2 and o_4 are over-encrypted with a SEL key k^s. Also, the KEK enabling Bob to compute k_2 is dropped from $\mathcal{R}_B$, while the KEKs enabling Alice and Dave to compute k^s are created and inserted into $\mathcal{R}_A$ and $\mathcal{R}_D$, respectively.

Insert object. When a new object o_j is inserted into a container C, the object inherits the acl of the container. To enforce such an authorization policy, the object owner u can simply decide to encrypt o_j in the same way as the objects already in the container. However, if the authorization policy regulating access to the container has already been modified, this would require to encrypt o_j with both the DEK at the BEL k and the DEK at the SEL k^s associated with the container. Since the policy of object o_j has never been updated, the adoption of the SEL might be an overdo. We therefore propose to adopt a new DEK k_{new} at the BEL to protect objects that are inserted into a container on which revoke operations had been applied. As a consequence of the revoke operation (and the new acl associated with the container), a new DEK BEL key (and the corresponding KEKs) corresponding to the new acl is generated for the container, and used for objects that will be inserted into the container after the revoke

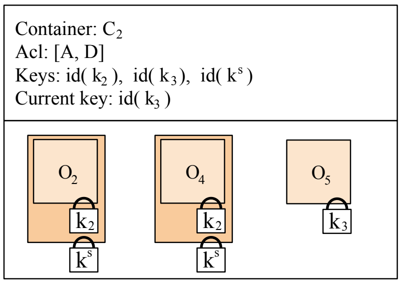

Fig. 4: An example of insertion of an object into an over-encrypted container

operation. While for existing objects over-encryption is needed to guarantee protection from the revoked user, new objects can be encrypted with the new key known only to the users actually authorized for them. To enable non-revoked users to derive the new (current) key of the container, an (asymmetric) KEK enabling them to derive the new key is added to their repositories. Consider, as an example, container C_2 illustrated in Figure 3(a), which is encrypted with k_2 at the BEL and with k^s at the SEL because of the revoke of Bob. Assume now that Alice needs to insert a new object o_5 into C_2. Object o_5 will be encrypted at the BEL with key k_3, generated when Bob has been revoked access to C_2 (together with the KEKs enabling Alice and Dave to compute k_3 from their own private key). Figure 4 illustrates the content of container C_2 after the insertion of o_5.

4.2 Implementation of Over-Encryption

The implementation of over-encryption for the enforcement of revoke operations in Swift can operate in different ways, depending on the time at which SEL encryption is applied, which can be: 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*). In the following, we elaborate on each of these strategies.

Immediate Over-Encryption. The storing server applies over-encryption when a user revokes the authorization over container C to a user u_i. Immediate over-encryption requires the user to define, at policy update time: the SEL DEK k^s necessary to protect the objects in the revoked container C, and the KEKs necessary to authorized users (and to the server) to derive k^s. Also, the objects in container C will be over-encrypted. The server will then immediately read from the storage the objects in C, re-encrypt their content (possibly removing SEL encryption), and write the over-encrypted objects back to the storage. Hence, immediately after the policy update, the objects in C are stored encrypted with two encryption layers. Every time a user needs to access an object in C, the server will simply return the stored version of the requested object. Figure 3(a)

illustrates container C_2 in Figure 2 after Bob has been revoked access to C_2, when adopting immediate over-encryption.
Immediate over-encryption causes a considerable cost at policy update time, which is however significantly lower than the cost that would be paid if over-encryption is not used. The advantage of immediate over-encryption lays in its simplicity in the management of `get` requests by clients, because objects will be returned by the server as they are stored. This approach can be an interesting option in scenarios where policy updates are extremely rare and the overall size of objects is modest.

On-the-fly Over-Encryption. The storing server applies over-encryption on-the-fly, that is, every time a user accesses an object. Then, even if the owner of the container asks the server to over-encrypt the objects in C, the server only keeps track of this request, but it does not re-encrypt stored objects. When a user needs to access an object in C, the server possibly over-encrypts the object before returning it to the user. Figure 3(b) illustrates the adoption of on-the-fly over-encryption when Alice accesses object o_2, after Bob has been revoked access to container C_2 in Figure 2. As it is visible from the figure, the server over-encrypts o_2 with k^s, which can be computed by Alice and Dave but not by Bob, before sending the object to the requesting user.
When adopting on-the-fly over-encryption, keys can be managed according to the following two strategies.

- *Static key generation*: the owner of the container defines, at revoke time, the SEL DEK k^s necessary to protect the objects in the revoked container C, and the KEKs necessary to non-revoked users (and to the server) to derive k^s.
- *Dynamic key generation*: the server generates a fresh SEL DEK k^s for every `get` request involving an object in the revoked container C. Also, it creates and makes available to the requesting user a KEK enabling her to derive k^s. At revoke time, the owner of the container only needs to communicate to the server the container C subject to the revoke operation and the revoked user.

In terms of performance, if the same user makes repeated requests for objects in the same container (i.e., protected with the same DEK), dynamic key generation may require a greater amount of work. On the other hand, if the number of requests for the objects in a container is significantly lower than the number of KEKs produced by the static approach for the same container, the dynamic approach is more efficient. The profile of key management for the two alternatives presents significant differences, but key management operations exhibit negligible computational and I/O costs compared to the management of the objects themselves. This is the reason why in the experiments (Section 5), focusing on the overall object management cost, we do not distinguish between static and dynamic key generation.
The advantage of on-the-fly over-encryption is that over-encryption is applied only when needed. However, if an object is asked multiple times during a period when the policy is stable, the server will incur a higher cost than immediate over-encryption, due to the multiple applications of encryption on the same ob-

ject. On-the-fly over-encryption can then be an interesting option in scenarios where the ratio between accesses and revoke operations is low.

Opportunistic Over-Encryption. This approach aims at combining the advantages of both immediate over-encryption and on-the-fly over-encryption. It presents a similarity with the *Copy-On-Write* approach commonly used by operating systems to improve the efficiency of copying operations. Analogously to the immediate approach, opportunistic over-encryption requires the owner, when a user is revoked access to a container, to define both the SEL DEK k^s necessary to protect the objects in the revoked container C, and the KEKs necessary to authorized users (and to the server) to derive k^s. Similarly to the on-the-fly approach, the server over-encrypts an object o_j in the revoked container C 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 management of opportunistic over-encryption is more complicated than the approaches illustrated above. In fact, after multiple policy updates and object insertions, a container may include objects associated with different BEL and SEL keys. Therefore, the object descriptor must specify also its state (i.e., not over-encrypted, over-encrypted with the most up-to-date SEL key, over-encrypted with an old SEL key). When a user needs to access an object o_j, the server first checks its descriptor. If o_j is protected only at BEL and it has been subject to a revoke operation, the server derives the most recent SEL key and over-encrypts o_j on-the-fly, storing then the result. If o_j is protected also at the SEL with the most up-to-date key (or it is encrypted only at the BEL and no revoke operation affected the container), it is returned to the requesting user. Finally, if o_j is protected at the SEL with an outdated key (e.g., because another revoke operation has been performed after o_j has been last accessed), the server decrypts o_j with the old SEL key, re-encrypts it with the new one, and stores the result. Note that KEKs enabling to derive old SEL keys can be dropped from repositories only when no object is protected with those keys. Figure 3(c) illustrates container C_2 in Figure 2 after Bob has been revoked access to C_2 and Alice has accessed object o_2. As it is visible from the figure, object o_2 is protected at both the BEL and SEL, while o_4 is encrypted only at the BEL as it has not been accessed yet. The critical advantage of opportunistic over-encryption is that it shows good adaptability to a variety of scenarios. In some peculiar combinations of policy update frequency, size of data collection, and access profile by clients, the other solutions may be preferable. However, based on our experimental results, we expect that this solution will be preferred in the majority of scenarios.

5 Experimental Results

We discuss the experimental results performed for evaluating the practical applicability of our proposal. We performed different series of experiments aimed at evaluating the following aspects:

- the benefits of the use of over-encryption compared to a system where policy changes are enforced by the client downloading, re-encrypting, and re-uploading the objects involved (Section 5.1);
- the performance of the immediate, on-the-fly, and opportunistic options (Section 5.2);
- the performance of a batch and a streaming option for the execution of encryption by the server (Section 5.3);
- the performance at the client-side for the removal of the two encryption layers for over-encrypted objects (Section 5.4).

The experiments were executed on two PCs with Linux Ubuntu 16.04, 16 GB RAM, 4-core i7 CPU, 256 GB SSD disk. The client and the server were connected with a 100 Mb/s network channel.

5.1 Comparison between Client Re-Encryption and Over-Encryption

We compare different options of over-encryption with a scenario where a policy update on a container is enforced by the data owner through the download, re-encryption and upload of the whole container. For this set of experiments, we consider a container with 1000 files of size 1 MB. Client side re-encryption does not require server work (except for the download and upload request, which are the same in every scenario) and is necessary only for revocations.

Figure 5 compares the overall time required for the management of a policy update followed by a number of `get` requests. The line on top corresponds to the configuration without over-encryption. In the lower part, we have the lines that describe the time required when using over-encryption, considering the on-the-fly approach and the opportunistic approach with uniform distribution of access requests (corresponding to $\alpha = 1$). We also report the time exhibited by the management of a sequence of direct `get` requests, where no encryption is applied to the objects. The graph shows that the lower lines are all one near to the other, proving that over-encryption has a small overhead.

5.2 Analysis of Over-Encryption Approaches

We compare the performance of immediate, on-the-fly, and opportunistic approaches. For this set of experiments, we consider a container with 100 files of size 1 MB. We focus on the time required for the processing on the server module, without considering the time required for the transfer of data across the network. This permits to focus on the component that is most influenced by these options (the network is typically a bottleneck and it hides the difference between the approaches, as shown in Figure 5). Figure 6 reports the cumulated execution time associated with a sequence of requests, for the three over-encryption approaches.

The immediate option requires, at policy update time, to read all the objects in the container, possibly decrypt them, and encrypt and write them back. This

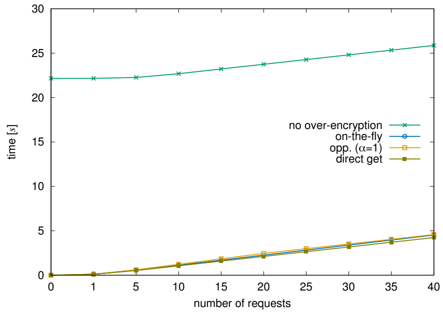

Fig. 5: Overhead of all the solutions

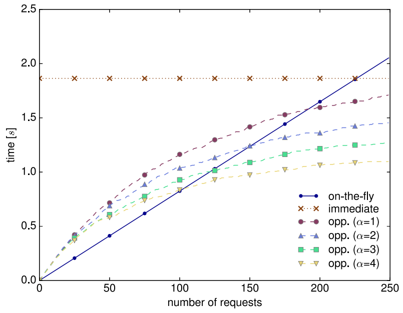

Fig. 6: Cumulative server work with different over-encryption approaches

creates an immediate overhead at policy update, before the first request. Subsequent requests do not require a specific processing by this module, which manages the `get` requests with a direct mapping to the retrieval of the over-encrypted representation of the object. Figure 6 represents the immediate approach with a horizontal line.

The on-the-fly option requires to apply SEL encryption on every returned object. The cost is then identical for all the requests. Figure 6 shows that the on-the-fly option is associated with a constant growth.

For the opportunistic approach, the cost depends on the number of files in the container that are accessed more than once. When an object is accessed for the first time after the policy update, the server will have to encrypt it at the SEL level and then save its new representation. This adds to the encryption cost the cost for the storage of the new version. Subsequent requests for the same object will be managed as a simple `get` of the over-encrypted representation of the object. The frequency of repeated accesses has then an impact on the efficiency of this approach. In our experiments, we therefore consider request profiles associated with power law distributions [11] with varying values for the α parameter, from 1 to 4. A value of α equal to 1 corresponds to a uniform distribution, where all the requests have an equal probability of asking any of the objects in the container; increasing values of α lead to an increasingly skewed distribution of requests. The analysis shows that for the first requests the cost associated with the opportunistic approach is greater than that of the on-the-fly approach. As requests continue to be executed, the opportunistic approach becomes increasingly more efficient compared to the on-the-fly approach. The advantage increases as the profile becomes more unbalanced. The worst case is represented by the uniform distribution, which still becomes more efficient after 180 requests.

From this experimental analysis we conclude that the choice of the over-encryption approach has to consider a few aspects. In terms of pure performance, the opportunistic approach always dominates the immediate approach.

The choice between the on-the-fly and the opportunistic approach has to evaluate the frequency of policy updates, the number of access requests generated between each policy update, and the profile of access requests. For scenarios where policy updates are relatively frequent compared to the frequency of access requests, and the profile is uniform, the on-the-fly approach can be the most efficient solution. In these scenarios, a choice should be made between the static and dynamic key generation. This choice will have to take into account design and configuration aspects, with the static generation requiring a greater upfront processing, but then more efficient computation, and the dynamic generation minimizing setup costs, but requiring a DEK and a KEK creation for every access request. In domains with a profile opposite to that leading to the on-the-fly approach, the opportunistic approach can prove to be the best option.

In addition to performance, there are design and security requirements that may have an impact on the choice. In terms of design, the opportunistic approach requires a more complex procedure, whereas the immediate and on-the-fly approaches both map to a simpler implementation. With respect to security, the immediate approach (for all the objects) and the opportunistic approach (for objects that have already been accessed since the last update) offer greater protection, because a revoked user who may have access to the Swift storage infrastructure would not be able to access the plaintext of the objects, whereas in the on-the-fly approach such an attack would succeed for a revoked user. System administrators will then have to make a choice based on the consideration of a number of parameters. Our expectation is that in most scenarios administrators will select the opportunistic approach.

5.3 Streaming and Batch Encryption

We performed a set of experiments aimed at comparing the execution time of a number of `get` requests when two different kinds of encryptions are used by the server: *Streaming* and *Batch*. They both use the AES-CTR encryption mode. Streaming encryption makes use of the WSGI structure of the Swift servers, and it consists in encrypting every chunk of the file as it is obtained from the proxy server. On the contrary, Batch encryption consists in encrypting the whole file after it is returned from the proxy server and before it is sent to the client. In these experiments, files of the same size are inserted into a container, which has the total size of 1 GB. We studied the benchmark of Streaming and Batch encryption applied to the on-the-fly approach against the direct `get` call that does not apply any encryption.

As it is visible from Figure 7, compared to the direct `get` call, Streaming encryption adds an overhead between 1% and 3%, whereas Batch encryption adds an overhead between 7% and 15%. It is then clear that Streaming encryption is more efficient, both because of shorter response times and because it has a lower memory usage, since it does not have to load the entire object in RAM before encrypting it. Note that the encryption of the chunks could also be parallelized, further reducing the overhead compared to the direct `get` call.

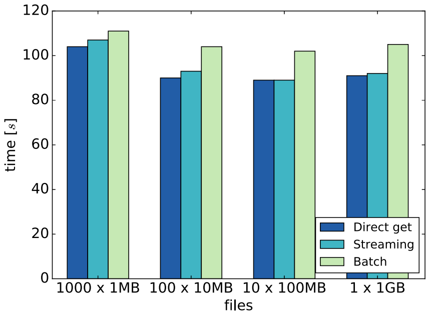

Fig. 7: Comparison of the overhead caused by Streaming and Batch on-the-fly approaches with respect to the direct `get` call

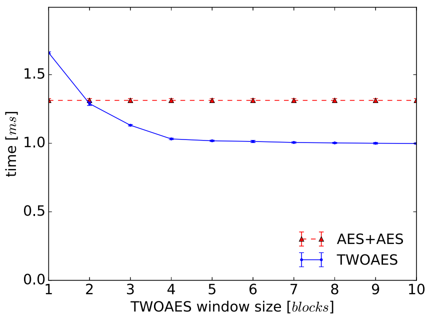

Fig. 8: BEL+SEL encryption performance on a 1MB file using two subsequent AES invocations and TWOAES

	without AES-NI			**with AES-NI**		
	ECB	**CBC**	**CTR**	**ECB**	**CBC**	**CTR**
128 bits	253 MB/s	215 MB/s	154 MB/s	1857 MB/s	408 MB/s	284 MB/s
256 bits	192 MB/s	170 MB/s	133 MB/s	1301 MB/s	336 MB/s	248 MB/s

Fig. 9: AES encryption rate for the modes *ECB*, *CBC*, and *CTR* using the *pycrypto* library without and with AES-NI

5.4 Application of Two Encryption Layers

When over-encryption is used, the client has to decrypt the downloaded objects twice, using the same encryption algorithm with two distinct keys. The simplest approach for the implementation of these two decryptions consists in first removing the SEL layer on the full object and then removing the BEL layer. Such an approach is not the most efficient option, because the portion of the object that has been SEL-decrypted (and still BEL-encrypted) will have to either be temporarily stored in RAM or on mass memory. This is similar to the analysis for Streaming and Batch encryption for the server, where Streaming encryption proves to be more efficient.

We started from these considerations and investigated the joint application of SEL and BEL decryptions. We were also interested in evaluating the performance profile of decryption on the client and in evaluating the impact of the hardware support offered for the execution of cryptographic functions. In particular, we verified the impact of the AES-NI (Intel AES New Instruction set) instructions available on Intel processors. A first set of experiments, reported in Figure 9, showed that the encryption performance of AES-NI compared to an AES software implementation (we used the one available in OpenSSL) is around 7 times faster.

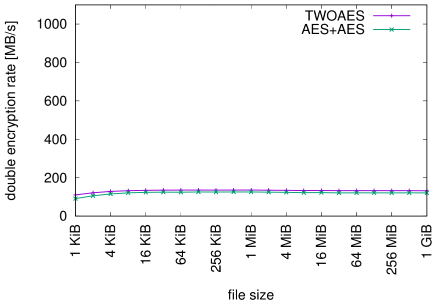

Fig. 10: Re-encryption using AES

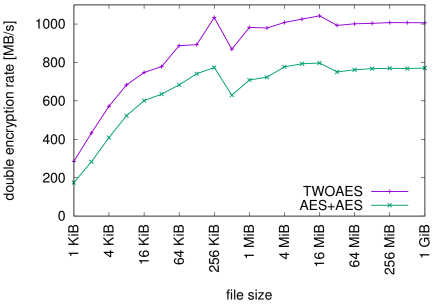

Fig. 11: Re-encryption using AES-NI

We then focused on the application of two decryptions. Our expectation was that the consecutive application of a SEL decryption and BEL decryption on the same block would have produced a benefit, as it would have avoided to pay the penalty of a transfer outside the CPU cache of the data. As shown in Figure 8, where AES-NI instructions were used, we instead observed that the performance of the interleaved decryption depends on the number of consecutive blocks processed with each key. The worst performance is observed when after each block there is a switch of encryption key. Further investigation allowed us to verify that the source of this behavior was an optimization by the C compiler that avoided to execute a write to the registers storing the key value when no changes had occurred to the key since the previous execution. When the switch from the application of the SEL decryption to the BEL decryption occurs after a number of blocks, the cost of the key setup is amortized over a number of blocks, but the blocks remain in the CPU cache after the first decryption and the second decryption becomes more efficient.

We then compared the execution times for the (a) serial application of SEL and BEL decryption (a full SEL decryption, followed by a full BEL decryption) and (b) interleaved SEL and BEL decryption, with the application of the two decryptions 8 blocks at a time. Figures 10 and 11 report the results of these experiments when not using AES-NI and when using AES-NI, respectively. The greater performance of hardware-accelerated AES emphasizes the impact that the CPU/RAM interface has on performance. Figure 10 indeed shows that the difference between the two approaches when hardware acceleration is not used is limited. Figure 11 shows that the 20% benefit observed is persistent across objects with a variety of sizes.

This approach is then the one that has to be applied whenever two layers of decryption have to be removed. It is also important to note that the throughput that can be obtained in the application of two decryptions (a few GB/s) is orders of magnitude greater than the bandwidth available for the network connections between a client and the Swift provider. This confirms the applicability of over-encryption in this scenario.

6 Related Work

The design of encryption techniques for data stored in the cloud is a large research area, with a considerable variety of topics and proposals. A significant amount of work has been dedicated to the design of techniques that support the efficient search and retrieval of encrypted data (e.g., [18]). Techniques have been designed that let the data be available only to users with specific properties (e.g., ABE [4, 12]). Another important line of research focuses on protecting access privacy (e.g., [9, 10, 17]). In this paper, we focus the analysis on over-encryption, on the approaches for existing cloud storage frameworks, and on proposals for the sharing of large client-encrypted objects (instead of structured data).

Over-encryption has been proposed to effectively and efficiently enforce policy updates over encrypted outsourced data [7, 8]. This solution considers the presence of a single data owner, and it has been extended to consider multiple users owning (and willing to share) data [6]. This approach differs from the solution we proposed as it relies on Diffie-Hellman, while our approach is based on the definition of symmetric and asymmetric KEKs. Also, these proposals consider a generic resource management scenario, with no specific connection to existing cloud frameworks. Over-encryption has also been considered in [3] in conjunction with a novel approach called Mix&Slice. In this context, over-encryption does not involve a whole resource but only a fragment of it.

Several proposals have contributed to the design of solutions for the protection of outsourced data with reference to current cloud frameworks. In [2], OpenStack security issues are extensively analyzed. The confidentiality of objects stored in Swift is considered as a significant aspect, but no specific technical solution is presented. A subsequent work by the same authors [1] describes an approach for the encryption of objects in Swift. In [14] another approach for server-side encryption is presented, with the goal of protecting "data at rest" (i.e., an approach for making the object representation on storage devices protected against physical accesses). In these approaches, keys are never seen by clients and they do not consider the support for container acls. Then, they do not have to look at the management of the encryption policy and its evolution.

A number of proposals have considered the application of encryption on the client-side. In [20], a service is presented that maps a file system to an encrypted representation on Amazon S3. The proposal does not support the sharing of files among distinct users and acls are not considered. In [13], an architecture for sharing encrypted objects outsourced to a cloud provider is presented. Revocation is considered as important and difficult and the proposed solution enforces it by limiting access to encryption keys for revoked users. In [21], an extensive architecture for the management of a cloud-based data sharing system is proposed. Resources are protected with keys that are consistent with the policy and significant attention is paid to revocation. The approach used is based on proxy re-encryption and lazy re-encryption. Proxy re-encryption relies on expensive cryptographic techniques that allow a server to convert a representation of a resource encrypted with a key to one associated with a different key, without letting the server executing the transformation be able to access the plaintext of the resource. Proxy re-encryption supports expressive encryption schemes, which allow attribute-based selection. Over-encryption uses standard symmetric encryption, which does not support those features but exhibits better performance. Lazy re-encryption shares some features with our opportunistic over-encryption approach, as it saves on re-encryptions by applying them only after an access request is made to the object, but the motivation is different. The advantage of lazy re-encryption is due to the ability to avoid re-encryptions for resources that are not accessed between a number of policy updates. The same benefit is also valid in our opportunistic approach, but in those scenarios our on-the-fly approach can be preferable.

The OpenStack Swift community is making a significant effort toward the introduction of object encryption in Swift [15]. The support is offered for the server side, aiming at protecting data at rest. We are monitoring this development and are confident that our solution can be easily adapted to leverage their implementation, extending it with our over-encryption techniques.

7 Conclusions

The design of techniques able to enforce confidentiality of outsourced data has the potential to greatly accelerate the rate of adoption of cloud storage, leading it to become the standard approach for the management of any kind of data. Local storage and traditional file systems would then only play the role of a cache that speeds up access to data, but persistence would be guaranteed by cloud providers. The realization of this vision requires to integrate the security techniques developed by the research community with existing cloud solutions.

The work presented in this paper goes in this direction and shows that this integration has to consider several aspects. Our proposal offers then a contribution for the most used open-source cloud storage solution, but the approaches that have been considered for Swift have a clear immediate application also to other domains.

Acknowledgements. 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. Albaroodi, H., Manickam, S., Anbar, M.: A proposed framework for outsourcing and secure encrypted data on OpenStack object storage (Swift). Journal of Computer Science 11(3), 590–597 (2015)
2. Albaroodi, H., Manickam, S., Singh, P.: Critical review of OpenStack security: Issues and weaknesses. Journal of Computer Science 10(1), 23–33 (2014)
3. 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 CCS. Vienna, Austria (October 2016)

4. Chow, S.S.M.: A framework of multi-authority attribute-based encryption with outsourcing and revocation. In: Proc. of SACMAT. Shanghai, China (June 2016)
5. De Capitani di Vimercati, S., Foresti, S., Jajodia, S., Livraga, G., Paraboschi, S., Samarati, P.: Enforcing dynamic write privileges in data outsourcing. Computers and Security 39, 47–63 (November 2013)
6. De Capitani di Vimercati, S., Foresti, S., Jajodia, S., Paraboschi, S., Pelosi, G., Samarati, P.: Encryption-based policy enforcement for cloud storage. In: Proc. of SPCC. Genova, Italy (June 2010)
7. De Capitani di Vimercati, S., Foresti, S., Jajodia, S., Paraboschi, S., Samarati, P.: Over-encryption: Management of access control evolution on outsourced data. In: Proc. of VLDB. Vienna, Austria (September 2007)
8. De Capitani di Vimercati, S., Foresti, S., Jajodia, S., Paraboschi, S., Samarati, P.: Encryption policies for regulating access to outsourced data. ACM TODS 35(2), 12:1–12:46 (April 2010)
9. De Capitani di Vimercati, S., Foresti, S., Paraboschi, S., Pelosi, G., Samarati, P.: Efficient and private access to outsourced data. In: Proc. of ICDCS. Minneapolis, USA (June 2011)
10. De Capitani di Vimercati, S., Foresti, S., Paraboschi, S., Pelosi, G., Samarati, P.: Shuffle index: Efficient and private access to outsourced data. ACM TOS 11(4), 19:1–19:55 (October 2015)
11. Easley, D., Kleinberg, J.: Networks, crowds, and markets: Reasoning about a highly connected world. Cambridge University Press (2010)
12. Goyal, V., Pandey, O., Sahai, A., Waters, B.: Attribute-based encryption for fine-grained access control of encrypted data. In: Proc. of ACM CCS. Alexandria, USA (October–November 2006)
13. Kaaniche, N., Laurent, M., El Barbori, M.: Cloudasec: A novel public-key based framework to handle data sharing security in clouds. In: Proc. of SECRYPT. Vienna, Austria (August 2014)
14. Kang, S., Veeravalli, B., Aung, K.M.M.: ESPRESSO: An encryption as a service for cloud storage systems. In: Proc. of AIMS. Brno, Czech Republic (June-July 2014)
15. Richling, J., Cole, A.: At-rest encryption, `http://specs.openstack.org/openstack/swift-specs/specs/in_progress/at_rest_encryption.html`
16. Sefraoui, O., Aissaoui, M., Eleuldj, M.: OpenStack: Toward an open-source solution for cloud computing. IJCA 55(3), 38–42 (2012)
17. Stefanov, E., van M. Dijk, Shi, E., Fletcher, C., Ren, L., Yu, X., Devadas, S.: Path ORAM: An extremely simple Oblivious RAM protocol. In: Proc of ACM CCS. Berlin, Germany (November 2013)
18. Wang, C., Cao, N., Ren, K., Lou, W.: Enabling secure and efficient ranked keyword search over outsourced cloud data. IEEE TPDS 23(8), 1467–1479 (August 2012)
19. Wen, X., Gu, G., Li, Q., Gao, Y., Zhang, X.: Comparison of open-source cloud management platforms: OpenStack and OpenNebula. In: Proc. of FSKD. Sichuan, China (May 2012)
20. Yao, J., Chen, S., Nepal, S., Levy, D., Zic, J.: Truststore: Making Amazon S3 trustworthy with services composition. In: Proc. of CCGrid. Melbourne, Australia (May 2010)
21. Yu, S., Wang, C., Ren, K., Lou, W.: Achieving secure, scalable, and fine-grained data access control in cloud computing. In: Proc. of INFOCOM. San Diego, USA (March 2010)