Access control delegation for the Cloud

Nikos Fotiou, Apostolis Machas, George C. Polyzos and George Xylomenos
Mobile Multimedia Laboratory
Department of Informatics
Athens University of Economics and Business,
Athens, Greece
Email:{fotiou, polyzos, xgeorge}@aueb.gr, a.machas@dias.aueb.gr

***Abstract*—Cloud computing has become the focus of attention in the industry, from the point of view of both providers and customers, as well as researchers. However, security concerns still impede the widespread adoption of this technology. Most enterprises are particularly worried about the lack of control over their outsourced data since the authentication and authorization systems of Cloud providers are generic and they cannot be easily adapted to the requirements of each individual enterprise. An adaptation process requires the creation of complex protocols, often leading to security problems and "lock-in" conditions. In this paper we present the design of a lightweight solution that overcomes these problems. We have implemented and incorporated this solution in a popular open-source Cloud stack: OpenStack. Our solution eliminates the need for developing complex adaptation protocols, offers data owners the flexibility to switch among Cloud providers, or use multiple, different Cloud providers concurrently, and enhances end-user privacy.**

I. Introduction

Cloud computing is an emerging paradigm that offers a cost-effective way for outsourcing data storage and computation. Nevertheless, despite its intriguing properties, enterprises are reluctant to fully adopt it, since they are concerned–among other things–about *losing the governance* of their outsourced assets, i.e., losing the ability to enforce their own, enterprise-specific, security policies. According to PwC's Global State of Information Security Survey 2012 [1], the largest perceived Cloud security risk is the "uncertain ability to enforce provider security policies", whereas according to the survey of Subashini and Kavitha [2] one of the biggest security challenges for providing Cloud-based services is the "adherence of the Cloud provider to the security policies of its clients", as well as "the administration of user authorization systems". It is therefore observed that, not only the mismatch between provider-enterprise security policies impedes Cloud adoption, but overcoming this problem is a challenging task that requires further research. Indeed, "effective models for managing and enforcing data access policies, regardless of whether the data is stored in the Cloud or cached locally on client devices" was identified back in 2010 as a top research priority, by the European Network and Information Security Agency (ENISA) [3].

One question that may arise is how likely it is for *loss of governance* of the outsourced data to occur, and what is its impact. According to ENISA's Cloud Computing Security Risk Assessment report [4], the loss of governance is a risk with *very high* probability, and with *very high* impact. The same report states that two of the vulnerabilities that may expose an enterprise to that risk are "unclear roles and responsibilities" and "poor enforcement of role definition". This outcome comes as no surprise, since the authentication and authorization systems of Cloud providers cannot capture the organizational structure and the security policies of each individual enterprise. The interoperability between the corresponding systems requires the development of complex API's; this, however, increases the chances of a security breach due to implementation errors, according to the Cloud Security Alliance [5].

In this paper, we propose a novel solution that gives full control of the access control assessment to the data owner, introducing minimal overhead for the Cloud provider. Our solution is based on a similar system that we developed for providing access control enforcement delegation in ICN architectures [6]. In our approach data outsourcing and access control assessment are treated as two clearly separated functionalities, implemented by different system entities: the former is implemented by a Cloud provider, whereas the latter is implemented by an *Access Control Provider* (ACP). The *ACP* is a trusted entity that may as well be provided by the enterprise itself, for example, by leveraging its user management system. This clear separation introduces many advantages, including: Cloud providers are relieved from the burden of implementing the business logic of each individual enterprise, enterprises can easily migrate from one Cloud provider to another, and user privacy is enhanced.

The paper is organized as follows. In Section II we discuss related work in this area. In Section III we detail our scheme. In Section IV we present our prototype that implements a secure private Cloud file storage service using the open source Cloud stack OpenStack. In Section V we evaluate the security properties of our solution and we analyze its performance. Finally, we conclude the paper in Section VI.

II. Related work

Single Sign-On (SSO) systems–such as Kerberos and, more recently, OpenID 2.0 [7] and OAuth 2.0 [8]–have similar goals with our scheme. In these systems, user identity management is performed by a separate trusted entity. Kerberos has been widely used for providing access control to network resources. In a Kerberos system a *Ticket Granting Service* (TGS) provides a "ticket" to an authenticated user that enables her to use a resource. The TGS and the resource, however, have to belong

to the same administration realm, or they should be pre-configured with a shared secret. In our system there is no restriction on the administrative domains in which the various entities should belong to. Moreover there is no secret with which an entity has to be pre-configured.

OpenID is an identity management system that allows third parties to delegate identity management to an *Identity Provider* (IdP) trusted by the user. In an OpenID system, the IdP is responsible for authenticating the user and for providing a token that proves that a user is authenticated. This token is unique per user, therefore it enables the third party to track user activity. Nunez et al. [9] used OpenID in conjunction with proxy re-encryption in order to provide Cloud based identity management services, whereas Khan et al. [10] have implemented OpenID based authentication mechanisms for the OpenStack platform. OpenID provides only user authentication; in an OpenID-based access control system, the Cloud provider is responsible for evaluating the access control policies. In our system tokens are ephemeral, therefore they can not be used to track the long term activity of a specific user. In addition, in our system the access control policy is evaluated by a third trusted party and not by the Cloud provider.

OAuth 2.0 is an IETF standard for authorizing access to resources over HTTP. OAuth 2.0 requires the resource owner to be online during the third party authorization procedure (Section 1.2 of [8]), and requires implicitly the development of a communication protocol between the resource server and the authorization server in order to be able to exchange an *access token* whose form–as mentioned in Section 1.4 of [8]–is not specified. The latter limitation raises obstacles to implementations in which the resource server and the authorization server belong to different administrative domains. An approach for providing access control using OAuth 2.0 is the following: the data owner defines an access control policy using *attributes* that can be provided by an authorization server (e.g., user age, as provided by a social network), these attributes are regarded as resources and they are accessed by the Cloud provider using OAuth 2.0; the Cloud provider uses these attributes and evaluates the access control policy. In this scenario, user credentials are protected. However, the Cloud provider learns some information about the user (in this example his age), and has to understand the authorization server specific attributes in order to evaluate the access control policy. In our system the Cloud provider learns nothing about the user and does not have to understand any authorization server-specific semantics.

Security Assertion Markup Language (SAML) [11] is an XML-based security assertion language, used for exchanging authentication and authorization statements about *subjects*. Being a *language* and not a system, SAML is orthogonal to our work. As a matter of fact, messages in our schemes can be exchanged using SAML, using the Authentication Request Protocol (Section 3.4 of [11]). However, our implementation follows OpenStack's API, which is incompatible with SAML.

III. System design

In this section we present our system design. We begin with a high level overview of our scheme and present our goals. Then we detail the functionality of our system.

A. Scheme overview

In our scheme we consider four basic roles: the *data owner (owner)*, the *data consumer (consumer)*, the *Cloud provider (CP)*, and the *access control provider (ACP)*. The goal of an *owner* is to store some data in a *CP* and allow *authorized consumers* to perform operations over this data. The data is protected using an *access control policy*. An access control policy is regarded as a function executed in an *ACP*. This function accepts as input a consumer's identification data and outputs either an error message if the user cannot be *authorized*, or an integer number that denotes the access *level* of the consumer. The access level of a *consumer* indicates which operations she can perform over the data that is protected by the corresponding access control policy.

In our scheme, the following trust relationships are considered: the *owner* trusts the *ACP* to authorize a *consumer*, and the *owner* and the *consumer* trust the *CP* to respect the decision of the *ACP*. The first trust relationship type can be trivially established if the *ACP* belongs to the *owner* (e.g., a leveraged enterprise user management system). The second trust relationship is a relaxed form of the currently existing trust relationship between an owner and a Cloud provider: currently, in the best case, an owner trusts a Cloud provider to securely store the owner's business logic, to execute it correctly and to enforce its outcome.

Our goal is to design a system in which the following properties hold:

- *The system is secure*: Provided that all system entities respect the trust relationships described above, it should not be possible for an attacker to perform an operation over some protected data, without being properly authorized.
- *Data consumer privacy is preserved*: In our system a *CP* should gain minimal information about the identity of a *consumer*. Ideally it should only learn that a *consumer* can be authorized by a specific *ACP* and the consumer's *level*. Moreover an *ACP* should not be able to tell the exact data that a *consumer* wants to access.
- *Data can be easily migrated among different Cloud providers*: In our system the only entities that should be aware of the access control policy and its implementation details are the *ACP* and the *owner*. *CPs* are oblivious about the access control policy implementation details. Therefore, providing two *CPs* implement our solution, moving data from one *CP* to another is as trivial as copy-pasting it.
- *An access control policy does not reveal anything about the data and the operations it protects*: In our system an access control policy is decoupled from the data and the operations it protects and it should be defined taking into account solely consumer attributes.
- *An access control policy is re-usable*: In our system it should be possible to use the same access control policy in order to protect many and diverse data items, stored in multiple *CPs*.
- *An access control policy can be easily modified*: In our system the modification of an access control policy

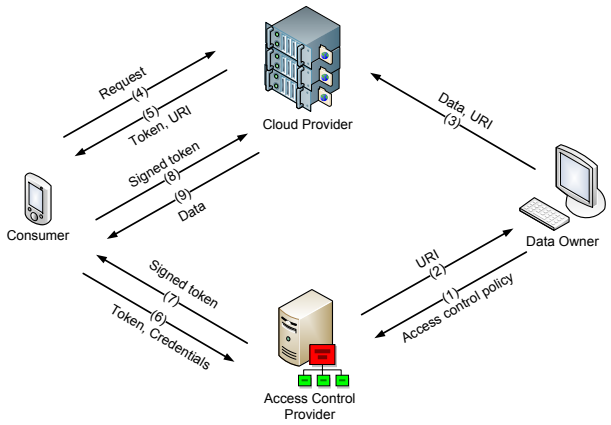

Fig. 1. Scheme overview.

should not involve any *CP*; the only entity that should be involved in the modification of an access control policy is the *ACP* where the policy is stored.

A high-level view of the interactions between the system entities is illustrated in Figure 1. An execution round of our scheme includes the following steps. Initially an *owner* stores an access control policy in an *ACP* and obtains a *URI* for that policy. As a next step she communicates the obtained URI, as well as the data it protects, to a *CP*, specifying at the same time the required access level(s) for each operation. When a *consumer* tries to perform an operation over some protected data for the first time, she receives as a response from the *CP* a *token* and the URI of the access control policy that protects the data item requested, and she is being redirected to the appropriate *ACP*. Then, the *consumer* authenticates herself to the *ACP*, by providing some form of identification data, and requests authorization, based on the access control policy that corresponds to the obtained URI. The *ACP* checks if the consumer satisfies the stored access control policy; if this is true, the *ACP signs* the token, including in the signature the consumer's access level. The signed token can now be used by the *consumer* in order to perform the desired operation.

B. Detailed system description

In this section we provide details about our system design. In our system it is assumed that *ACPs* and *CPs* have a pair of public/private keys, and the public keys are known to the *owners*, as well as to the *consumers*. Moreover, it is assumed that all messages are exchanged over a secure channel. Throughout this section the notation of Table I is used. Our system consists of the following functions :

1) Access control policy creation and data storage: This function is executed using out-of-band mechanisms. With this function an *owner* initially creates and stores an *access control policy* in an *ACP*. The *ACP* in return provides a URI_{acp}. In order to protect a data item stored in a *CP*, using an access control policy represented by URI_{acp}, the *owner* has to communicate to the *CP* the URI_{acp}, the Pub_{ACP}, as well as the *levels* of *consumers* that are allowed to perform each operation. A *CP* maintains for each data item a URI_{acp}, a Pub_{ACP} and an *Access Table* that contains tuples of the form $< operation, levels >$. A URI_{acp} is re-usable, i.e., it can be used to protect multiple items stored in many *CPs*.

TABLE I
NOTATION

Pub_{CP}	The public key of a CP
Pub_{ACP}	The public key of an ACP
URI_{data}	The URI of a data item stored in a Cloud provider
URI_{acp}	The URI of an access control policy
$Sign_{ACP}(Y)$	The digital signature of plaintext Y using the private key of an ACP

2) Data operation, unauthorized request: This function is executed by a *consumer* in order to perform an operation over some protected data, stored in a *CP*. The *consumer* sends a data operation request message to the *CP*. This message contains the operation and a URI_{data}. Upon receiving such a request the *CP* creates a unique *token* and sends it back to the consumer, along with the corresponding URI_{acp}. Therefore, the following exchange of messages takes place:

$$MSG\ \#1:\ Consumer \rightarrow CP : Operation, URI_{data}$$

$$MSG\ \#2:\ CP \rightarrow Consumer : URI_{acp}, Token$$

It should not be possible for a third party to guess a token. In order to keep track of the generated tokens *Cloud providers* maintain a *Token Table* that contains entries of the following form: $< Token, authenticated, expires, URI_{acp}, Level >$. When a new token is generated, a new entry is added to this table, with $authenticated$ being set to $false$ and $expires$ being set to the generation time plus a very small amount of time, sufficient to obtain an authorization.

3) Consumer authentication and authorization request: This function is executed by a *consumer* upon receiving the response of the data operation request. Initially the *consumer* sends her identification data, along with the Pub_{CP} and the URI_{acp} and $Token$ she received with message $MSG\ \#2$, to the *ACP* responsible for evaluating the access control policy stored in URI_{acp}. The *ACP* verifies the *consumer's* identification data against URI_{acp}. If the *consumer* satisfies URI_{acp}, the *ACP* creates a new message that contains the token, the authorization level of the consumer, the amount of time that the token should be valid (i.e., its lifetime), the URI_{acp}, and the Pub_{CP}. Then it signs this message and sends it back to the *consumer*. Therefore during this function the following messages are exchanged:

$$MSG\ \#3:\ Consumer \rightarrow ACP : IDdata, Pub_{CP}, URI_{acp}, Token$$

$$MSG\ \#4:\ ACP \rightarrow Consumer : M2, Sign_{ACP}(M2)$$

where:

$$M2 = Token, Level, Lifetime, URI_{acp}, Pub_{CP}$$

4) Data operation, authorized request: With this function a *consumer*, claiming to be authorized, requests to perform an operation over some protected data. The request includes the operation, the URI_{data}, the token, the token's lifetime and the signature of the $M2$ part of the $MSG\ \#4$ message. Therefore the following message is sent:

$$MSG\#5:\ Consumer \rightarrow CP : operation, URI_{data}, Token, Level, Lifetime, Sign_{ACP}(M2)$$

Upon receiving this message a *CP* performs the following actions:

1) Find the token in the Token Table and check if it has expired. If it has expired, return an error
2) If the $authenticated$ field of the corresponding record in the Token Table is $false$ then
   a) Retrieve Pub_{ACP} that corresponds to URI_{data}
   b) Retrieve the URI_{acp} that corresponds to the token
   c) Reconstruct the $M2$ part of the $MSG\ \#4$ message
   d) Verify $Sign_{ACP}(M2)$, using Pub_{ACP}
   e) If the signature verification succeeds adjust the expiration time of the token according to the $LifeTime$ field, set $authenticated$ equal to $true$, set the appropriate value in the $Level$ field, and proceed to Step 3a.
   f) If the signature verification fails, return an error and exit
3) if the $authenticated$ field of the corresponding record in the Token Table is $true$ then
   a) Find the URI_{acp} and the level that corresponds to the token, from the Token Table
   b) Find the URI_{acp} and the level for the requested operation that corresponds to the URI_{data}, from the Access Table
   c) Check if the retrieved values match. If they match perform the operation, else return an error

Once the *CP* adjusts the Token Table and marks a token as authenticated, then the consumer does not have to include the $Level, Lifetime, Sign_{ACP}(M2)$ fields in her subsequent requests; the $Token$ is sufficient.

IV. Implementation

As a proof of concept we implemented a secure file storage service[1] using a popular open source Cloud stack, the OpenStack[2]. In particular we leveraged the functionality of the OpenStack component *Swift*, which is used for building object storage systems. The implemented system allows file storage and retrieval, as well as the following operations over the stored files: organizing files in containers, listing the files of a container, copying a file, moving a file and deleting a file. We implemented our communication channels using HTTPS and we pre-configured the *consumer* software with the public keys of the *CP* and the *ACP* components.

[1]http://pages.cs.aueb.gr/~fotiou/software/access/index.html
[2]http://www.openstack.org/

A. Swift-based architectures

A Swift-based object storage architecture is composed by two networks: the internal (private) network that consists of *storage nodes*, and the external (public) network that consists of a *proxy* server and (optionally) an *authentication* server. The proxy server accepts HTTP requests and processes them using a Web Server Gateway Interface. The parameters used in each request are encoded as HTTP headers. Each request is pipelined through a number of add-ons, each of which may transform it, forward it, or respond on behalf of the system to the user.

Objects stored in a Swift-based architecture are organized in a three level hierarchy. The topmost level of this hierarchy is the *accounts* level, followed by the *containers* level (second level) and the *objects* level (third level). The accounts level contains user accounts. Each user account is associated with many containers from the containers level. A container is used for organizing objects, therefore a container is associated with many objects from the objects level. An object may be a file or a folder (that contains other objects). Every object within a container is identified by a container-unique name. Each request for an operation over an object contains a URI that denotes the account, the container and the name of the object in question.

B. Add-on implementation

The *CP* part of our scheme has been implemented as a Swift add-on added in the pipeline of the add-ons processing incoming requests. For each supported operation a user may specify an account-wide URI_{ACP}, a container-wide URI_{ACP}, or an object-wide URI_{ACP}. For each URI_{ACP} the corresponding Pub_{ACP} is provided. When a request is pipelined for the first time through our add-on, the add-on checks if a URI_{ACP} has been set for the object URI specified in the request (or its container, or its account); if this condition is true, the add-on generates a new *token*, using the token generation mechanism provided by Swift, and creates a $MSG\ \#2$ message as described in Section III-B. The add-on creates a new entry in the *Token Table* that contains the *token*, as well as the corresponding URI_{acp}. The authenticated field of this entry is set to $false$ and the expiration time is set equal to the current time plus 10 sec. Finally the add-on responds with $MSG\ \#2$ to the *consumer*.

Upon receiving $MSG\ \#2$, the *consumer* initiates the authentication and the authorization process, which involves the exchange of messages $MSG\ \#3$ and $MSG\ \#4$ with the appropriate *ACP*. In our system we implemented a simple *ACP* that authenticates users using a username and a password, and authorizes them using an access control list stored in an SQLite database. With the reception of $MSG\ \#4$ the *consumer* is ready to perform an authorized request. The first time an authorized request is made, all parameters of message $MSG\ \#5$ have to be set. In all subsequent requests only the token is sent to the CP.

V. EVALUATION

A. Security evaluation

It can be easily observed that our system enhances *consumer* privacy. The only information that a *CP* learns about a *consumer* is that he has a trust relationship with a particular ACP, as well as his level. Of course, the latter can be encoded in such a way that it will not reveal any meaningful information. Any other sensitive information is stored in the (trusted) *ACP*. Moreover, regardless of the lifetime of a token, a *consumer* may drop it and request a new one in order to avoid being profiled by a *CP*. Finally an *ACP* does not gain any information about the actual data item that a consumer wants to access: the only information that the *ACP* learns is the public key of the entity that hosts the desired item.

Another security feature of our system is that access control policies can be easily modified. Access control policies are stored in a single point (in the *ACP*) and all protected assets have a *pointer* to that policy; therefore, the modification of an access control policy does not involve communication with the *CP(s)* in which protected data is stored. When an access control policy is changed, all new *consumers* will be authorized using the new policy, whereas all already authorized *consumers* will be re-authorized with the new policy when their token expires.

We now proceed to the security analysis of our system using the threat model proposed by Wang at al. [12], adapted to the context of our system. For our analysis we consider three different attack scenarios: (A) a malicious entity that can be authorized under an access control policy P_{mal}, acting as a consumer trying to perform an operation over a data item protected by an access control policy P_{leg}, with P_{mal} and P_{leg} stored in the same *ACP*, (B) a malicious entity that acts as a *CP* pretending to host an item protected by an access control policy P_{leg}, and trying to access a data item protected by P_{leg} stored in a different CP, and (C) a malicious entity trying to impersonate a consumer from the same system. In all cases we assume that messages are exchanged through a secure channel and communication endpoints cannot lie about their identity. Finally, we do not consider the case in which a malicious entity acts as an *ACP* and steals the credentials of a consumer, since this attack is out of the scope of our system.

1) Malicious entity acting as a consumer: In this attack scenario a malicious entity, Con_M tries to perform an operation over an item protected by an access control policy P_{leg}, stored in ACP_A. Con_M does not abide by P_{leg}, but he abides by another access control policy, namely P_{mal}, also stored in ACP_A. Con_M's goal is to obtain a $MSG\ \#4$ message in which the $M2$ part would be equal to $(Token, Level, Lifetime, \mathbf{URI_{P_{leg}}}, Pub_{CP})$. Under normal circumstances Con_M will receive a $MSG\ \#4$ message with an $M2$ part of the following form $(Token, Level, Lifetime, \mathbf{URI_{P_{mal}}}, Pub_{CP})$. If Con_M simply replaces $URI_{P_{mal}}$ with $URI_{P_{leg}}$ then $Sign_{ACP}(M2)$ will not be valid anymore, therefore the *CP* will understand the attack. The only way to include $URI_{P_{leg}}$ in message $MSG\ \#4$, with $Sign_{ACP}(M2)$ being valid, is to include $URI_{P_{leg}}$ in message $MSG\ \#3$, i.e., have Con_M send to ACP_A a message $MSG\ \#3$ of the following form: $(IDdata, Pub_{CP}, URI_{P_{leg}}, Token)$. However since Con_M does not abide by $URI_{P_{leg}}$ this message will result in an error.

2) Malicious entity acting as a CP*:* In this attack scenario we assume that the attacker's goal is to perform an operation over a data item $Item_A$ stored in CP_A and protected by an access control policy P_A, stored in ACP_A. The attacker acts as a Cloud provider, CP_B, which hosts a data item, $Item_B$, also protected by P_A. Moreover the attacker is able to lure a consumer Con_L, that abides by P_A, to perform an operation over $Item_B$.

The attacker initially sends a message $MSG\ \#1$ to CP_A and obtains a $Token_A$; in order for this attack to be successful the attacker has to obtain a $MSG\ \#4$ message with an $M2$ part of the following form $(\mathbf{Token_A}, Level, Lifetime, URI_{P_A}, \mathbf{Pub_{CP_A}})$. When Con_L is lured to request to perform an operation over $Item_B$, stored in CP_B[3], the attacker responds with a message $MSG\ \#2$ of the following form: $(URI_{P_A}, Token_A)$. Subsequently Con_L sends a message $MSG\ \#3$ to ACP_A of the following form: $(IDdata, Pub_{CP_B}, URI_{P_A}, Token_A)$, and receives a message $MSG\ \#4$ with an $M2$ part of $(\mathbf{Token_A}, Level, Lifetime, URI_{P_A}, \mathbf{Pub_{CP_B}})$. In order for the attacker to obtain the desired message he has to replace Pub_{CP_B}, with Pub_{CP_A}, but in this case $Sign_{ACP}(M2)$ will not be valid anymore, therefore CP_A will detect the attack.

3) Malicious entity co-located with a consumer: This attack scenario is applicable when a *CP* maintains a user management system and associates operations over protected data with particular users (e.g., for charging reasons). In these cases a *CP* maintains in its Token Table the identifier of the (*CP*) user for whom the token has been generated. The goal of an attacker in this scenario is to make a *CP* believe that a consumer Con_L wants to perform an operation OP_A over an item $Item_A$ protected by access control policy P_A. For this scenario it is assumed that the attacker is also a valid *CP* user and he is eligible to perform OP_A over $Item_A$. Moreover it is assumed that the attacker is able to inject messages on behalf of Con_L.

In order for this attack to take place, the attacker requests to perform OP_A over $Item_A$ and proceeds through all steps until he receives $MSG\ \#4$. At this point, instead of sending $MSG\ \#5$ on behalf of himself, he sends it on behalf of Con_L. It can be easily observed that this attack is trivially mitigated since the *CP* also maintains the identifiers of the users that correspond to each token, therefore $MSG\ \#5$ will be rejected. It should be noted however that this is possible due to our design choice to have the *CP* generate the tokens, which is not always the case in other similar systems. This attack, for example, was successfully exploited by Wang at al. [12] against three popular websites that were using Facebook connect and Twitter OAuth for associating their user accounts with their corresponding Facebook and Twitter accounts.

[3]According to our assumptions, the attacker cannot pretend to be CP_A

B. Overhead

In our implementation, HTTP methods (GET, PUT, DELETE) are used for denoting the desired operation. The size of the RSA keys is 2048 bits and the keys are encoded in JSON format. The size of an encoded key is 400 bytes. Every other field is encoded as a string of hexademical digits: tokens are encoded in a 32 byte string, the digital signatures in a 512 byte string and the token's lifetime in an 8 byte string. Finally, a single byte is used to represent access levels. When a *consumer* wants to perform an operation over some data stored in a CP, protected by an URI_{acp}, a number of messages has to be exchanged. If an *ACP* has already asserted that the *consumer* abides by the URI_{acp}, and the corresponding authenticated *Token* (that has been generated by the *CP*) has not expired, then a single message from the *consumer* to the *CP* has to be sent. In any other case five messages have to be exchanged: three between the *consumer* and the *CP*, and two between the *consumer* and the *ACP*.

C. Interoperability

Many cloud providers offer storage services (e.g., Amazon S3, Google Drive, Microsoft OneDrive), as well as an API for accessing and managing the stored content. This API can be used to build *middleware* providing controlled access to the storage service using our solution. The process for building this middleware is the following: the account that is used for accessing the storage services is kept secret, the middleware is given full permissions over the stored content, and all applications are configured to interact with the middleware, which is now used as an interface to the storage service. The middleware can be built using the infrastructure of the cloud provider (e.g., Amazon EC2, Google App Engine, Microsoft Azureus), or as a standalone service provided by a third party. The middleware should implement the legacy API of the storage provider in order to facilitate the extension of existing applications, as well as a standard network storage API (e.g., CIFS) so as to facilitate the migration from one storage provider to another.

VI. Conclusions

In this paper we proposed a solution that enables data owners to outsource data storage and computation, without losing governance of their assets. Our solution introduces a new role, that of the Access Control Provider (ACP), that relieves Cloud providers from the burden of implementing complex security solutions and enables enterprises to deploy their own access control mechanisms. Data can be easily migrated from one Cloud provider to another, since Cloud providers are oblivious about the access control policy implementation details and business logic behind it. We demonstrated the feasibility of our scheme through a proof of concept implementation, using a real, publicly available, Cloud stack system.

Our solution also creates a new business opportunity. We envision that a new market can arise due to our solution, that of the ACPs. In addition to the enterprise specific ACPs, there can be independent ACPs that offer security services to end-users. Existing security companies can utilize their expertise to offer cutting edge access control services without investing in the Cloud market. Moreover, existing social networks may leverage their role to act as ACPs. To this end, future work for our scheme includes support for ACP federations and support for multiple URI_{ACP} definitions per single data item.

Acknowledgment

This research was supported by a grant from the Greek General Secretariat for Research and Technology, financially managed by the Research Center of AUEB.

References

[1] PwC, "Global state of information security survey," 2012.
[2] S. Subashini and V. Kavitha, "A survey on security issues in service delivery models of cloud computing," *Journal of Network and Computer Applications*, vol. 34, no. 1, pp. 1–11, 2011.
[3] S. Gorniak (ed.), "Priorities for research on current and emerging network trends," *ENISA*, 2010.
[4] D. Catteddu and G. Hogben (eds.), "Cloud Computing Benefits, risks and recommendations for information security," *ENISA*, 2009.
[5] Cloud Security Alliance. (2013) The notorious nine cloud computing top threats in 2013. [Online]. Available: https://cloudsecurityalliance.org/
[6] N. Fotiou, G. F. Marias, and G. C. Polyzos, "Access control enforcement delegation for information-centric networking architectures," *SIGCOMM Comput. Commun. Rev.*, vol. 42, no. 4, pp. 497–502, 2012.
[7] D. Recordon and D. Reed, "OpenID 2.0: a platform for user-centric identity management," in *Proc. of the 2nd ACM workshop on Digital Identity Management*, 2006, pp. 11–16.
[8] D. Hardt (ed.), "The OAuth 2.0 authorization framework," *RFC 6749*, October 2012.
[9] D. Nunez, I. Agudo, and J. Lopez, "Integrating OpenID with proxy re-encryption to enhance privacy in cloud-based identity services," in *Proc of the IEEE 4th International Conference on Cloud Computing Technology and Science*, 2012.
[10] R. Khan, J. Ylitalo, and A. Ahmed, "OpenID authentication as a service in OpenStack," in *Proc. of the 7th International Conference on Information Assurance and Security*, 2011, pp. 372–377.
[11] S. Cantor, J. Kemp, R. Philpott, and E. Maler (eds.), "Assertions and protocols for the OASIS Security Assertion Markup Language (SAML) v2.0," *OASIS*, 2005.
[12] R. Wang, S. Chen, and X. Wang, "Signing me onto your accounts through facebook and google: A traffic-guided security study of commercially deployed single-sign-on web services," in *Proc. of the IEEE Symposium on Security and Privacy*, 2012, pp. 365–379.