Access control as a service for the Cloud

Nikos Fotiou, Apostolis Machas, George Xylomenos, George C. Polyzos
Mobile Multimedia Laboratory, Department of Informatics
School of Information Sciences and Technology
Athens University of Economics and Business
Athens 10434, Greece
fotiou@aueb.gr, a.machas@dias.aueb.gr, xgeorge@aueb.gr, polyzos@aueb.gr

***Abstract*—Cloud computing has become the focus of attention in the computing industry. 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 access control solution that overcomes these problems. With our solution access control is offered as a service by a third trusted party, the Access Control Provider. Access control as a service enhances end-user privacy, eliminates the need for developing complex adaptation protocols, and offers data owners flexibility to switch among Cloud providers, or to use multiple, different Cloud providers concurrently. As a proof of concept, we have implemented and incorporated our solution in the popular open-source Cloud stack OpenStack. Moreover, we have designed and implemented a Web application that enables the incorporation of our solution into Google Drive.**

***Index Terms*—Aauthorization; authentication; delegation; security; policies; IEEEkeywords**

I. Introduction

Cloud computing is a technology 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". This mismatch between provider-enterprise security policies severely impedes Cloud adoption and further research on effective solutions for this problem is required. 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 *loss of governance* of the outsourced data is, 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 *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 organizational structure and the security policies of an individual enterprise cannot be easily captured by a Cloud provider. Moreover, the interoperability between an enterprise and a Cloud provider requires the development of complex communication protocols; this, however, increases the chances of a security breach due to implementation errors, according to the Cloud Security Alliance [5]. Armando et al. [6] exploited such implementation errors in order to bypass the SAML-based[1] single sign-on system of Google apps. Similarly, Somorovsky et al. [7] gained access to multiple SAML-based systems by exploiting implementation bugs. Nevertheless, even if the developed protocol is implemented correctly, it will be Cloud provider specific, thus hindering the migration of an enterprise to another Cloud provider; this condition is known as *lock-in*, and has been identified as a *high* probability risk by ENISA [4].

In this paper, we propose a novel solution that enables a trusted entity to store enterprise-specific security policies and take access control decisions on behalf of a Cloud provider: the Cloud provider then has only to respect the access control decision. This trusted entity, which is referred to as the *Access Control Provider* (ACP), may as well be provided by the enterprise itself, for example, by leveraging its user management system, or by a third party. Compared to existing systems, our solution offers better end-user privacy and requires a much simpler communication protocol.

This paper extends our previous work presented in [8], with a more detailed system description, an additional proof of concept implementation, more extensive overhead evaluation, and further comparison with existing systems. 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 OpenStack, an open source

[1]SAML is a generic XML language used for security assessments between different entities.

Cloud stack, as well as a Web application that enables the incorporation of our solution in Google Drive. In Section V we evaluate the security properties of our solution and analyze its performance. Finally, in Section VI we discuss further extensions to our solution and we conclude in Section VII.

II. Related work

Single Sign-On (SSO) systems–such as Kerberos and, more recently, OpenID 2.0 [9] and OAuth 2.0 [10]–have similar goals with our scheme. Kerberos has been widely used for controlling access 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 domain or they should be pre-configured with a shared secret. Our system requires neither common administrative domains nor pre-shared secrets.

OpenID is an identity management system that allows identity management delegation to a third trusted party, known as the *Identity Provider* (IdP). IdPs authenticate users and provide them with an "authentication token", which they can use to access a resource. OpenID has been studied in the context of Cloud computing. Nunez et al. [11] used OpenID in conjunction with proxy re-encryption in order to provide Cloud based identity management services. Smilarly, Khan et al. [12] have implemented OpenID based authentication mechanisms for the OpenStack platform. OpenID provides only user authentication, therefore, in an OpenID-based access control system, the Cloud provider is responsible for evaluating the access control policies. Moreover, the authentication token is unique per user, therefore user activity can be tracked. In our system access control policies are evaluated by ACPs and not by the Cloud providers. In addition, in our system tokens are ephemeral, therefore they can not be used to track the long term activity of a specific user.

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 user authorization procedure (Section 1.2 of [10]), 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 [10]–is not specified. This vagueness impedes implementations of systems where the resource server and the authorization server belong to different administrative domains. An approach for implementing access control using OAuth 2.0 is the following: an access control policy based on *attributes* that can be provided by an authorization server (e.g., user age, as provided by a social network) is defined and stored in the Cloud, the Cloud provider accesses the required attributes using OAuth 2.0 and uses them to evaluate the access control policy. In this scenario, the Cloud provider not only learns some information about the user (in this example his age), but it is also able to interpret them. In our system, Cloud providers neither learn anything about users nor do they have to understand any enterprise-specific semantics.

Policy Based Admission Control [13] is a framework that allows a *Policy Enforcement Point* (PEP) to delegate access control policy decisions to a *Policy Decision Point* (PDP). Each Cloud provider can operate a PEP, whereas PDPs can be implemented by third trusted parties, or even the enterprises themselves. A PEP is responsible for collecting all the information required by a PDP, which includes information about the user that requests access. Moreover, a PEP and a PDP should agree on a, usually complex, communication protocol (e.g., COPS [14]). With our solution, Cloud providers are completely oblivious about access control policies. Moreover, Cloud providers neither collect nor learn any information about users. Finally, our communication protocol is much simpler, therefore less prone to implementation errors.

The Security Assertion Markup Language (SAML) is an XML-based security assertion language [15], 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 scheme can be exchanged via SAML, using the Authentication Request Protocol (Section 3.4 of [15]).

III. System design

A. Overview

Our scheme is composed of the following entities: 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. Each operation is protected by an *access control policy*. An access control policy is stored in an ACP and maps the *identity* of a consumer to a boolean output (true, false). When the output of an access control policy is true, the consumer that provided the identification data is considered authorized.

In our scheme, the following trust relationships are considered: owners trust ACPs to authorize consumers, and owners and consumers trust CPs to respect the decisions of ACPs. The first type of trust relationship can be trivially established if the ACP is implemented by the owner (e.g., the ACP leverages the enterprise's user management system). The second type of trust relationship is a relaxed form of the trust relationship that currently exists between an owner and a Cloud provider: in a contemporary Cloud system where access control is implemented in the Cloud, an owner trusts a Cloud provider to (i) securely store some enterprise-specific security policies (ii) to use these policies correctly, i.e., understand their semantics, and (iii) to enforce the outcome of the access control decision.

As illustrated in Figure 1 a typical transaction in our system takes place as follows. Initially, an owner stores an access control policy in an ACP (step 1) and obtains a *URI* for that policy (step 2). As a next step, she implements an operation over some data in a CP and stores the URI of the policy that protects this operation (step 3). When a consumer tries to perform a protected operation for the first time (step 4), she receives in response the URI of the access control policy that protects the operation and a unique token (step 5). Then, the consumer authenticates herself to a suitable ACP by providing some form of identification data and requests authorization for the access control policy specified in the obtained URI (step

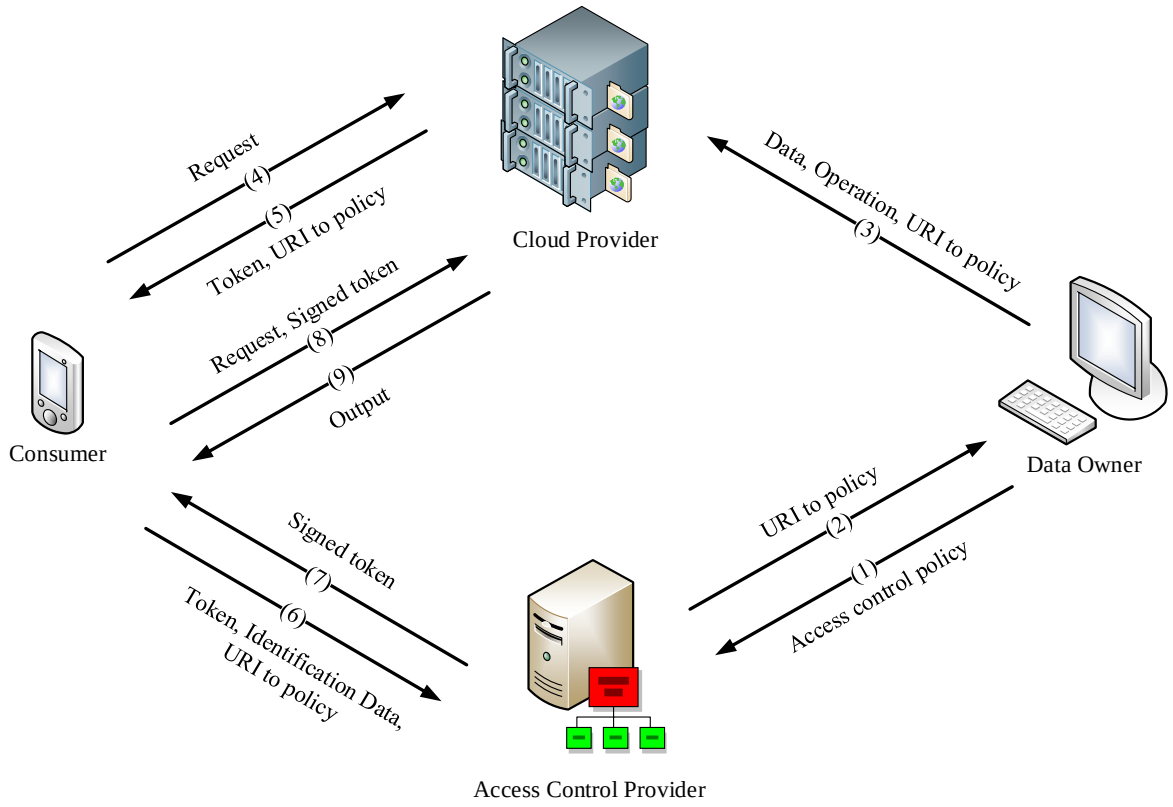

Fig. 1. Scheme overview.

6). If the consumer "satisfies" the access control policy, the ACP *signs* the token and sends it back to the consumer (step 7). The consumer repeats her request to the CP including this time the signed token (step 8). The CP checks the validity of the token and if the token is valid it executes the desired operation and returns its output (step 9).

B. Goals

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

- *The system is secure:* Provided that all system entities respect the trust relationships described previously, it shall not be possible for an unauthorized user to perform a protected operation.
- *Consumer privacy is preserved:* A CP shall gain minimal information about the identity of a consumer. Ideally it will only learn that a consumer can be authorized by a specific ACP. Moreover an ACP should not be able to tell the operation a consumer wants to perform or the data she accesses.
- *Data can be easily migrated among different Cloud providers:* The only entities that should be aware of an access control policy and its implementation details are the ACP and the owner. CPs shall be oblivious about the access control policy implementation details. Therefore, if two CPs implement our solution, moving data from one CP to another shall be almost as trivial as copy-pasting it.
- *An access control policy does not reveal anything about the data and the operations it protects:* Access control policies should be decoupled from the data and the operations they protect. An access control policy should be defined taking into account solely consumer attributes.
- *Access control policies are re-usable:* An access control policy should not be bound to a particular operation. It should be possible to protect many and diverse data items, stored in multiple CPs.
- *An access control policy can be easily modified:* The modification of an access control policy shall not involve CPs: the only entity involved in the modification of an access control policy should be the ACP where the policy is stored.

C. Detailed system description

We now detail our system design (Figure 2). We have made the following assumptions: (i) ACPs and CPs have a public-private key pair, (ii) ACP's and CP's public keys are known to the consumers and (iii) all messages are exchanged over a secure channel. Throughout this section the notation of Table I is used.

Our system consists of the following procedures:

Access control policy creation and data storage: With this procedure an owner creates and stores an access control policy

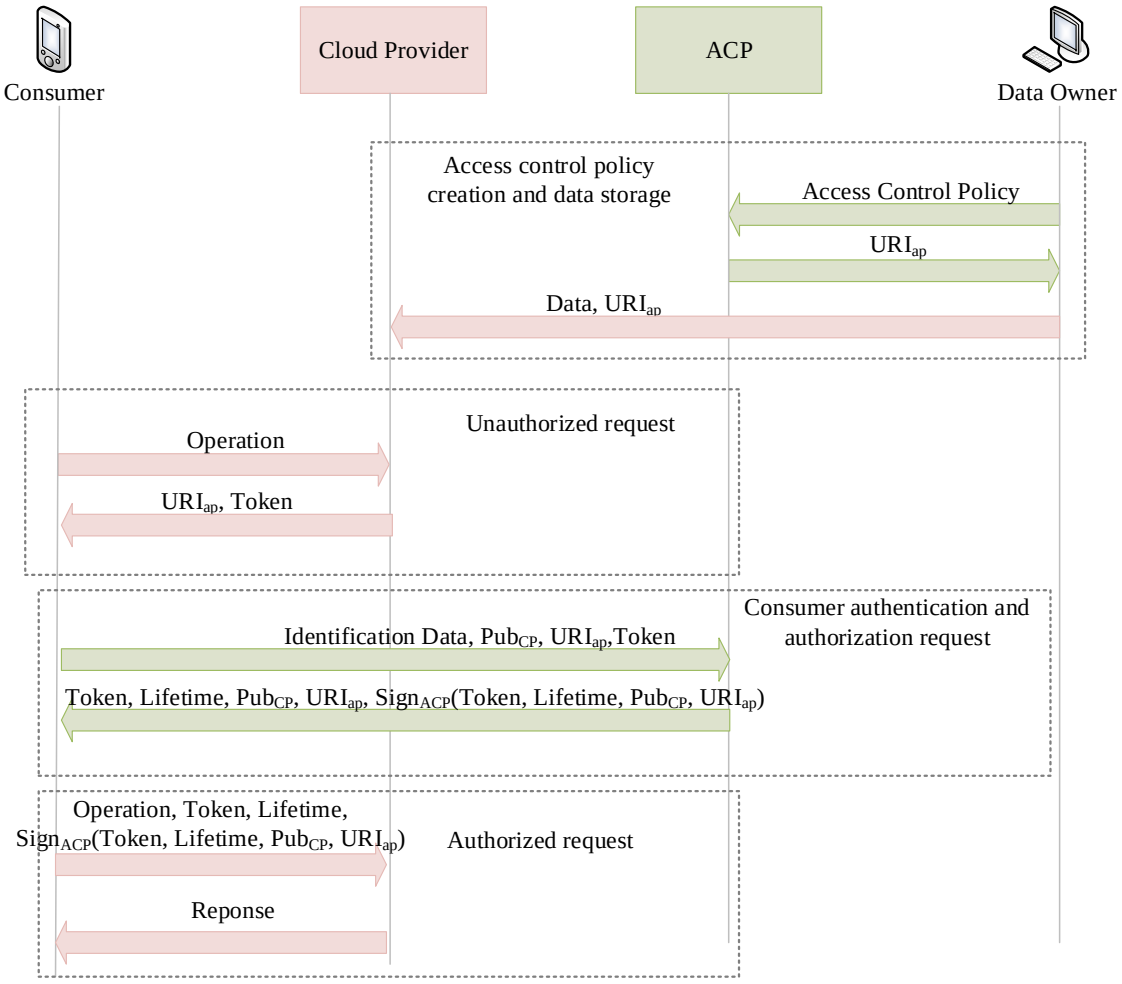

Fig. 2. System procedures.

TABLE I
NOTATION.

Pub_{CP}	The public key of a CP
Pub_{ACP}	The public key of an ACP
URI_{ap}	The URI of an access control policy
$Sign_{ACP}(Y)$	The digital signature of plaintext Y generated using the private key of an ACP

in an ACP. The ACP in return provides a URI_{ap}. For each protected operation implemented in a CP, the owner defines the URI_{ap} of the policy that protects it and the Pub_{ACP} of the ACP where the policy is stored. This information is maintained in the CP's *Access Table* that contains tuples of the form:

$$[operation, URI_{ap}, Pub_{ACP}]$$

A URI_{ap} is re-usable, i.e., it can be used for protecting multiple operations stored in various *CPs*. The mechanisms for creating an access control policy and for updating an Access Table are ACP specific and CP specific, respectively.

Unauthorized request: This procedure is executed by a consumer in order to perform an operation for the first time. The consumer sends an operation request message to the CP. Upon receiving the request the CP creates a unique *token* (i.e., an adequately large random number) and sends it back to the consumer, along with the corresponding URI_{ap}. Therefore, the following exchange of messages takes place:

$$(1) : Consumer \rightarrow CP : operation\ request$$
$$(2) : CP \rightarrow Consumer : URI_{ap}, Token$$

In order to keep track of the generated tokens, a CP maintains a *Token Table* that contains entries of the form:

$$[Token, authenticated, expires, URI_{ap}]$$

When a new token is generated, a new entry is added to this table. The value of the $authenticated$ field of this entry is set to $false$ and the value of the $expires$ field to the generation time plus a very small amount of time, sufficient to obtain an authorization.

Consumer authentication and authorization request: This procedure is executed by a consumer upon receiving a response to an unauthorized request. Firstly, the consumer sends her identification data, Pub_{CP}, URI_{ap} and the token

to an *ACP* responsible for evaluating the access control policy identified by URI_{ap}. If the consumer satisfies URI_{ap}, the ACP creates an *authorization* message that contains the token, the amount of time that the token should be valid (i.e., its lifetime), URI_{ap}, and Pub_{CP}. Then it signs this message and sends it back to the consumer. Therefore, the following messages are exchanged:

$(3) : Consumer \rightarrow ACP : ID, Pub_{CP}, URI_{ap}, Token$
$(4) : ACP \rightarrow Consumer : auth, Sign_{ACP}(auth)$

where:
$auth = Token, Lifetime, URI_{ap}, Pub_{CP}$

Authorized request: This procedure is executed by an ACP authorized consumer in order to perform an operation. The consumer sends a message that includes the operation request, the token, the token's lifetime and the signature of the authorization message (i.e., message (4)). Therefore the following message is sent:

$(5) : Consumer \rightarrow CP : operation\ request,$
$Token, Lifetime, Sign_{ACP}(Auth)$

Upon receiving this message, a CP should decide if the consumer is allowed to perform the requested operation. Therefore, it executes the following algorithm. (Figure 3):

1) Retrieve the entry of the Token Table that contains the token and check if the token 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 the Pub_{ACP} that corresponds to the operation from the Access Table
   b) Retrieve the URI_{ap} that corresponds to the token from the Token Table
   c) Reconstruct the authorization message
   d) Verify $Sign_{ACP}(auth)$, using Pub_{ACP}
   e) If the signature verification succeeds, update the *Token Table* entry as follows: set the $expires$ field equal to the $LifeTime$ field of the authorization message and set the $authenticated$ field to $true$. Proceed to Step 3a below.
   f) If the signature verification fails, return an error
3) If the $authenticated$ field of the corresponding record in the Token Table is $true$ then
   a) Find the URI_{ap} that corresponds to the token from the *Token Table*
   b) Find the URI_{ap} of the requested operation from the *Access Table*
   c) Check if the retrieved values match. If they match return, else return an error

If this procedure is successful then any subsequent *authorized request* may include only the token. Moreover, the same token can be used multiple times, even for invoking different operations protected by the same URI_{ap}.

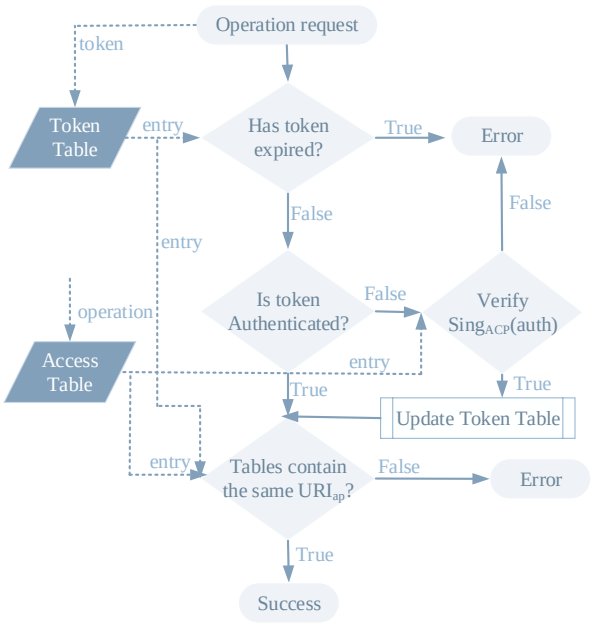

Fig. 3. Authorized request decision process.

D. Use case

Let us now illustrate our scheme through a use case. Enterprise A has outsourced sales records storage and analysis to Cloud provider CP_A. The operations implemented in CP_A are: *update sales records*, *calculate statistics*, and *view statistics*. Enterprise A has the following access control policies:

- Policy 1: All sales department employees can update sales records
- Policy 2: Only the sales department director can calculate statistics
- Policy 3: All shareholders can view the statistics

Enterprise A implements the above access control policies in an ACP owned by itself. The public key of this ACP is denoted by Pub_{ACP}. For each policy the ACP generates a URI, i.e., *entA.com/Policy1*, *entA.com/Policy2* and *entA.com/Policy3*. CP_A's Access Table is updated with the following entries:

TABLE II
CP_A ACCESS TABLE NEW ENTRIES.

Operation	$\mathbf{URI_{ap}}$	ACP public key
Update records	*entA.com/Policy1*	Pub_{ACP}
Calculate statistics	*entA.com/Policy2*	Pub_{ACP}
View statistics	*entA.com/Policy3*	Pub_{ACP}

The sales department director issues an unauthorized request for the *calculate statistics* operation. CP_A generates a token, namely $Token_1$, and responds by sending the following message (entA.com/Policy2, $Token_1$). CP's Token Table is then updated with the following entry:

As a next step, the sales department director authenticates himself to the ACP, which responds with the follow-

TABLE III
CP_A TOKEN TABLE NEW ENTRIES.

Token	authenticated	expires	URI_{ap}
$Token_1$	$false$	$timestamp1$	entA.com/Policy2

ing, digitally signed, authorization message: ($Token_1$, timestamp2, entA.com/Policy2, Pub_{CP_A}). Then, the sales department director issues the following authorized request: ("calculate statistics", $Token_1$, timestamp2, $Sign_{ACP}(auth)$). CP_A checks if $Token_1$ has expired. Then, it reconstructs the authorization message by retrieving the URI_{ap} associated with the *calculate statistics* operation (i.e., entA.com/Policy2) from the Access Table and verifies $Sign_{ACP}(auth)$ using Pub_{ACP} (also found in the Access Table). Finally, CP_A checks if the URI_{ap} found in the Access Table matches the URI_{ap} included in the entry for $Token_1$ in the Token Table. If all these steps are successful, CP_A executes the *calculate statistics* operation and modifies the entry for $Token_1$ in the Token Table as follows:

TABLE IV
CP_A TOKEN TABLE MODIFIED ENTRY.

Token	authenticated	expires	URI_{ap}
$Token_1$	$true$	$timestamp2$	entA.com/Policy2

Since $Token_1$ is now marked as *authenticated*, the sales department director can use it in all subsequent requests, until it expires. Moreover, as long as $Token_1$ remains valid, $Sign_{ACP}(auth)$ does not have to be included in subsequent requests.

E. The "level" extension

In the above use case, it can be observed that if the sales department director wishes to invoke the *update records* operation, he has to re-authenticate himself, since this operation is protected by a different URI_{ap}. The *level* extension mitigates this shortcoming by adding a new field to an Access Table: the consumer level. The consumer level is a number that denotes the *minimum* level that a consumer should have in order to invoke an operation. Using this extension, the Access Table of the Cloud provider considered in the use case of Section III-D can be modified as follows:

TABLE V
CP_A ACCESS TABLE USING LEVEL EXTENSION. THE **ACP PUBLIC KEY** COLUMN IS NOT SHOWN.

Operation	$\mathbf{URI_{ap}}$	level
Update records	*entA.com/Policy2*	100
Calculate statistics	*entA.com/Policy2*	200
View statistics	*entA.com/Policy3*	100

With this extension, an ACP has to include the consumer level in the authorization messages. Moreover, a CP now takes part in the access control decision, since it has to check if the level included in the authorization message is greater or equal to the level included in the Access Table. Finally, if the level extension is used, Token Tables should, additionally, include the level that corresponds to a token.

Suppose that the level of the sales department director in the previous use case is 200. Then, he would be able to successfully invoke the *update records* operation, using $Token_1$, without re-authenticating himself.

IV. IMPLEMENTATION

As a proof of concept we implemented a secure file storage service using a popular open source Cloud stack, the OpenStack[2], as well as a Web application that allows the incorporation of our solution in Google Drive[3]. The ACP and the consumer software used in both implementations are the same. Our implementation supports the level extension. As a public-key encryption system we use RSA. Public keys are encoded in JSON format using the keyCzar[4] python library. The keyCzar library is also used for generating digital signatures.

A. ACP and consumer software

The ACP of our proof of concept is implemented as a PHP application hosted in an Apache web server. An SQLite database is used for storing username-password pairs, as well as username to URI_{ap}-level mappings. Usernames are unique and a username can be mapped to many URI_{ap}-level pairs (e.g., Table 6). The consumer software implements the *authentication and authorization request*, by encoding the username, the password and the request parameters in a JSON object and by POSTing this object to a particular URL, using HTTPS. The response to this request is again encoded in a JSON object. The consumer software has been pre-configured with the public keys of the *CP* and the *ACP* components

TABLE VI
AN INSTANCE OF THE USER MANAGEMENTS SYSTEM.

Username	Password
fotiou	12345
machas	12345
polyzos	12345
xylomenos	12345

Username	$\mathbf{URI_{ap}}$	level
fotiou	*mmlab/Policy1*	100
fotiou	*mmlab/Policy2*	200
machas	*mmlab/Policy1*	200
machas	*mmlab/Policy3*	300
polyzos	*mmlab/Policy3*	100
polyzos	*mmlab/Policy4*	200
xylomenos	*mmlab/Policy3*	100
xylomenos	*mmlab/Policy4*	200

B. OpenStack-based implementation

For our OpenStack-based CP (Figure 4), we leveraged the functionality of the OpenStack component *Swift*, which is used for building object storage systems. A Swift-based object storage system is composed of two networks: the internal (private) network that consists of *storage nodes*, and the external (public) network that consists of a *proxy* server and

[2]http://www.openstack.org/
[3]https://drive.google.com/
[4]https://code.google.com/p/keyczar/

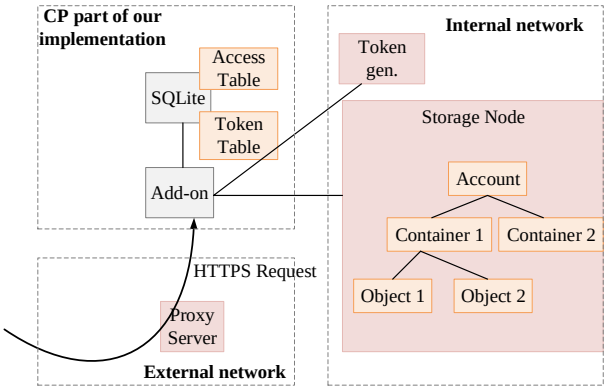

Fig. 4. OpenStack-based implementation.

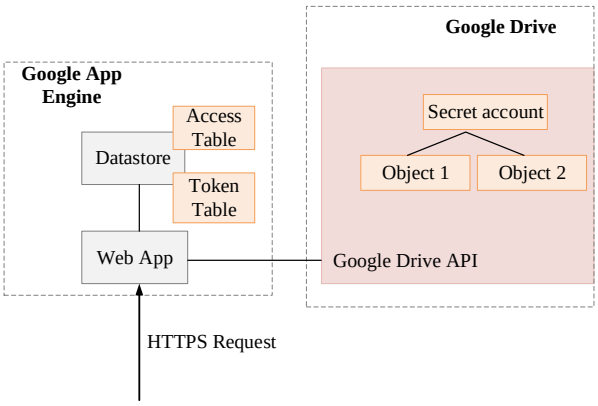

Fig. 5. Google Drive-based Web application.

(optionally) an *authentication* server. The proxy server accepts HTTP(S) requests and processes them using a Web Server Gateway Interface. The parameters used in each request are encoded in 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 system 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, i.e., it is of the form "https://CP HostName/account name/container name/object name".

We implemented our system as a Swift add-on added in the pipeline of the add-ons that process incoming requests. Our implementation 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. Token and Access Tables are implemented as SQLite tables. An owner hard codes in the Access Table records of the form: [path,URI_{ap},level,Pub_{ACP}]. A path may be account-wide, container-wide, or object-wide.

Initially, the consumer software sends an unauthorized request over HTTPS. The desired operation is specified in a HTTP header and the URL of the request denotes the object (or the container, or the account) that will be used as input to the operation. When an unauthorized request is pipelined through our add-on, the add-on checks if a URI_{ap} exists in the Access Table for the URL specified in the request: if such a URI_{ap} exists, the add-on generates a new token, using the token generation mechanism provided by Swift, and creates a response (as described in Section III-C); each part of the response is encoded in a HTTP header. The add-on then creates a new entry in the Token Table. The initial expiration time of a token is set equal to the current time plus 10 sec. Upon receiving the response, the consumer software initiates the authentication and the authorization process described in Section IV-A. As a next step, the consumer software sends an authorized request, encoding all request parameters in HTTP headers. The add-on executes the authorized request decision algorithm and produces the appropriate output.

C. Google Drive-based Web application

Google Drive is a popular Cloud based storage service. Google Drive provides a rich API that can be used for building applications that interact with the service over HTTPS. In our implementation we used this API and built a Web application that extends (part of) the Google Drive API, thus providing support for our protocol (Figure 5). Our application is built using the Google App Engine[5] and the Python language. Access Tables and Token Tables have been implemented using the Google App Engine Datastore. Currently, our application supports operations for uploading and downloading files. Each operation can be invoked by making an HTTPS call to the operation-specific URL. All call parameters are encoded in HTTP headers.

Our application has been configured with a Google Drive account which is kept secret. Instead of interacting with the "drive" directly, the consumer software interacts with the application, which acts as a middleware, ensuring that only an authorized consumer can perform the implemented operations. The consumer learns no information about the Google Drive account.

The owner hard codes in the web application a URI_{ap} that controls who can invoke the *upload file* operation. A consumer initially performs an unauthorized request for uploading a file (the file is not included in this request). The web application generates a token using the UUID Python function, it responds to the consumer by encoding the token in an HTTP header and updates the Token Table. The consumer software initiates the authentication and the authorization process described in Section IV-A. Then, it issues an authorized request, by

[5]https://developers.google.com/appengine/

encoding the request parameters in HTTP headers and the file as raw POST data. The web application executes the authorized request decision algorithm and if the consumer is allowed to upload the file, it stores it in the Google Drive. When uploading files, consumers are able to specify a URI_{ap} that controls who can invoke the *download file* operation for that specific file.

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 his trust relationship with a particular ACP; if the level extension is used, the CP also learns his level. Of course, the latter can be encoded in a way that reveals no meaningful information. Any other sensitive information is stored in a (trusted) ACP. Moreover, regardless of the lifetime of a token, a consumer may drop it and request a new one in order to avoid CP *profiling*. Finally, an ACP gains no information about the operations a consumer invokes and the data he accesses: the only information that an ACP learns is the public key of the CP with which the consumer interacts.

Another security feature of our system is that access control policies can be easily modified. Access control policies are stored in a single point (i.e., the ACP) and all CPs have *pointers* to policies. Therefore, the modification of an access control policy does not involve communication with any CP. When an access control policy is modified, 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. [16], adapted to our system. In our analysis we consider three different attack scenarios. In all scenarios we assume that messages are exchanged over a secure channel and communication endpoints cannot lie about their identity. 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 protected by an access control policy URI_{leg} stored in ACP_A. Con_M can only be authorized for the access control policy URI_{mal}, also stored in ACP_A. Con_M's goal is to obtain an authorization message of the form $(Token, Level, Lifetime, \mathbf{URI_{leg}}, Pub_{CP})$. By following our protocol Con_M will receive an authorization message of the form $(Token, Level, Lifetime, \mathbf{URI_{mal}}, Pub_{CP})$. If Con_M includes the signature of this message in his *authenticated request*, the *authorized request decision* algorithm will result in an error, since the CP will generate a different authorization message for which this signature is not valid (Figure 6). The only way for Con_M to obtain a valid signature is to include URI_{leg} in the *authentication and authorization request*, i.e., Con_M should send to ACP_A an authentication and authorization request of the following form: $(ID, Pub_{CP}, URI_{leg}, Token)$. However, since Con_M does not abide by URI_{leg} this message will result in an error.

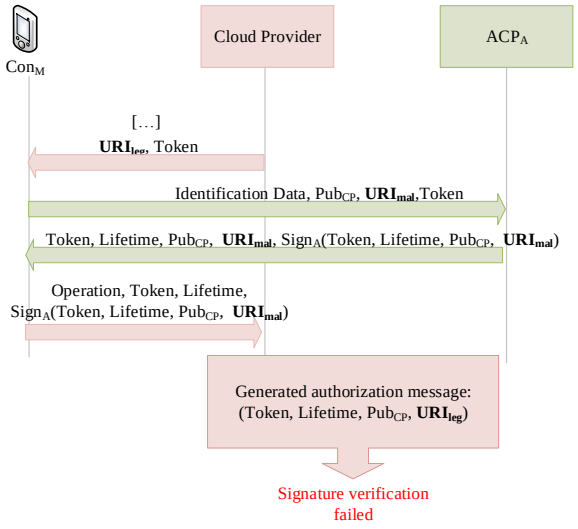

Fig. 6. Malicious entity acting as a consumer.

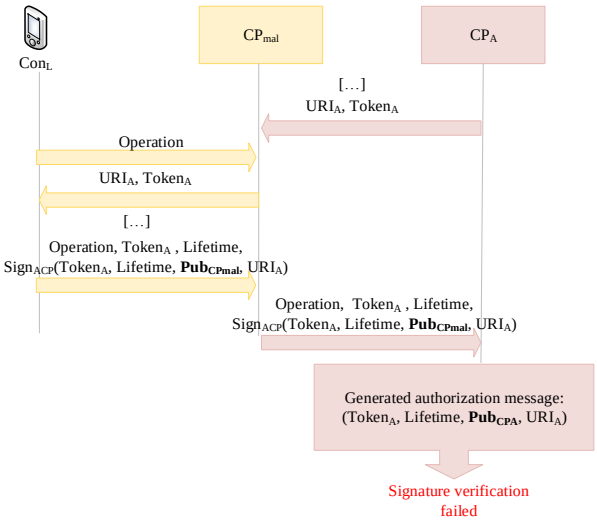

Fig. 7. Malicious entity acting as a CP.

2) Malicious entity acting as a CP: In this attack scenario the attacker's goal is to perform an operation in CP_A, protected by an access control policy URI_A stored in ACP_A. The attacker is able to pretend to be a Cloud provider, CP_{mal}, as well as to lure a consumer Con_L that can be authorized for URI_A, to perform this operation. Therefore, this is a man-in-the-middle type of attack.

The attacker initially sends an unauthorized request to CP_A and receives $Token_A$ and URI_A. In order for this attack to be successful the attacker has to obtain an authorization message of the form $(\mathbf{Token_A}, Level, Lifetime, URI_A, \mathbf{Pub_{CP_A}})$. Con_L is lured to send an unauthorized request to CP_{mal} (i.e., to the attacker), which responds with a message of the form: $(URI_A, \mathbf{Token_A})$. Subsequently, Con_L sends

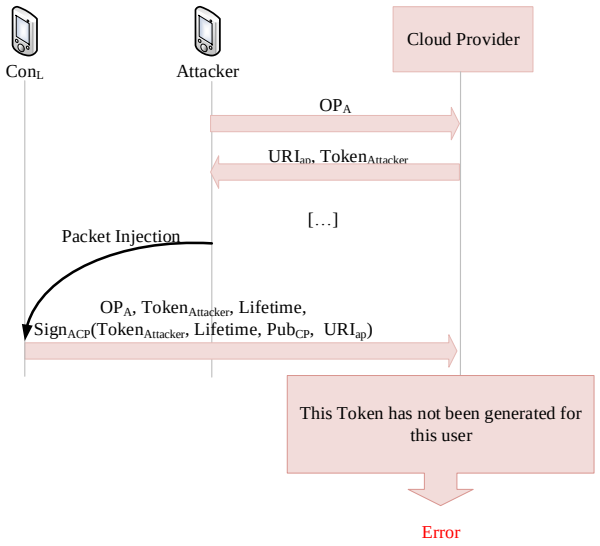

Fig. 8. Malicious entity co-located with a consumer.

an authentication and authorization request to ACP_A of the following form: $(ID, Pub_{CP_{mal}}, URI_A, \mathbf{Token_A})$, and receives the following authorization message $(\mathbf{Token_A}, Level, Lifetime, URI_A, \mathbf{Pub_{CP_{mal}}})$. If the attacker sends an authorized request using the signature of the previous message the *authorized request decision* algorithm will result in an error, since CP_A will generate an authorization message that includes Pub_{CP_A} and not $Pub_{CP_{mal}}$ (Figure 7).

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 tis scenario a CP also 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 a protected operation. In this scenario the attacker is a valid CP user and he is eligible to perform the same operations as Con_L. Moreover, the attacker is able to inject messages on behalf of Con_L.

In this attack scenario, the attacker requests to perform an operation OP_A and proceeds through all steps until he receives the authorization message. At this point, instead of sending an authorized request 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 this message will be rejected (Figure 8). It should be noted, however, that this is possible due to our design choice to have the CP generating the tokens, which is not always the case in other similar systems. This attack, for example, was successfully exploited by Wang at al. [16] against three popular websites that were using Facebook Connect and Twitter OAuth for associating their user accounts with their corresponding Facebook and Twitter profiles.

B. Overhead

In our implementation, HTTP methods are used for invoking the desired operation. As a public-key encryption system we use RSA. The size of an RSA public key is 2048 bits, whereas the size of a JSON encoded public key is 400 bytes. Tokens are encoded in 32 byte hex-strings, digital signatures in 512 byte hex-strings and token lifetimes in 8 byte hex-strings. Finally, a single byte is used for representing access levels. When a consumer wants to invoke an operation in a CP, protected by a URI_{ap}, a number of messages has to be exchanged. If an ACP has already generated for the consumer an authorization message for URI_{ap} and the corresponding token 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.

It can therefore be observed that an ACP and a consumer have a strong motive to use long-lasting tokens[6]: the longer the duration of a token, the less the communication overhead for an ACP and a consumer. On the other hand, long-lasting tokens increase the state that a CP has to maintain in its Token Table. In order to illustrate this tradeoff, we simulate the following scenario: we consider a CP that hosts files of 100 different enterprises. Each enterprise has defined a single protected operation. Moreover, each enterprise has 100 employees who invoke the operation stored in the CP following a Poisson process with rate 0.1/min. We simulate a usage period of 8 hours and every 5 min we measure the average network load of each enterprise (caused by the messages exchanged with the ACP), as well as the size of the CP's Token Table (the measured size is the average value of all the sizes the Token Table had within the 5 min measurement period). We consider two types of tokens: a token with short lifetime (20 min) and a token with long lifetime (2 hours). Figure 9 illustrates the average Token Table size of the CP throughout the simulation period, whereas Figure 10 illustrates the average number of messages transmitted inside each enterprise's network, throughout the simulation period.

C. Comparison with existing systems

We now compare our solution with two popular related systems: Google Drive and Amazon S3.

1) Google Drive: The Google Drive Cloud-based storage service, enables users to access, share, and organize their files in the Cloud. The Google Drive API provides a limited set of policies, namely, "full access", "read only access", "metadata only access", and "specific file access". These policies are not applied per stored item, instead they are granted in the form of "permissions" to applications that want to access a specific drive. Before using a "drive", an application requests from the drive owner one of the aforementioned permission types; the drive owner authenticates himself using a Google account and grants permissions using OAuth2.0. In most cases, the user that executes the application that requests permissions and the owner of the drive are the same entity. Permissions are

[6]Provided that this does not jeopardize the security of the scheme.

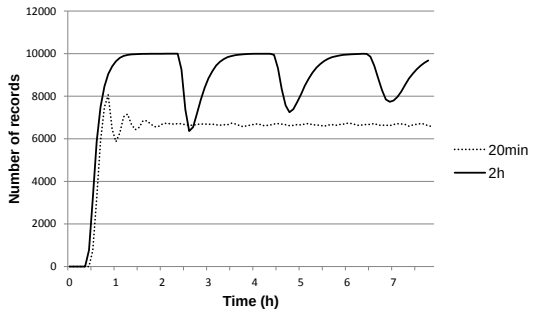

Fig. 9. Average number of Token Table entries as a function of token lifetime, using 5 minute sampling periods.

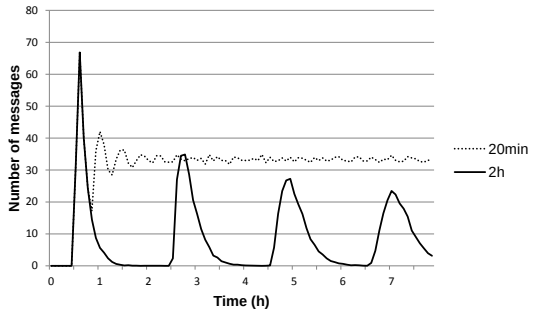

Fig. 10. Number of messages exchanged between a consumer and an ACP as a function of token lifetime, using 5 minute sampling periods. During the lifetime of a token, no messages are exchanged.

granted in the form of a token that never expires: in order for a drive owner to remove permissions for a specific application, she has to revoke the token manually. Google Drive does not support integration with enterprise specific authentication and authorization systems[7].

In order for an application to perform an operation the following messages have to be exchanged (here we consider that the user executing the application is the drive owner, referred to as the consumer):

1) $Consumer \rightarrow Google\ Auth$: Request permission
2) $Consumer \rightarrow Google\ Auth$: Authenticate
3) $Consumer \rightarrow Google\ Auth$: Grant permission
4) $Google\ Auth \rightarrow Consumer$: Token
5) $Consumer \rightarrow Google\ Drive$: Operation, Token

Compared to our system the same number of messages is required. Nevertheless, messages 1 to 4 are usually sent once, since tokens never expire. It should be also noted that the entity that performs the authorization is the drive owner herself (the consumer), therefore authorization is a manual process.

2) Amazon S3: Amazon Simple Storage Service, or S3 for short, is a well-known Cloud-based file storage service. S3 provides Web services that allow users to store and organize their files in the Cloud. Files are organized in "buckets". A user may set *Access Control Lists* (ACLs) that define the permissions that a user or a group of users have over a specific bucket, or over a specific file. ACLs are encoded in XML and the permissions that can be granted are "read", "write", "read ACL", "modify ACL", "full control". For more fine grained access control, S3 provides an "access control policy language", that allows users to create bucket-specific policies. These policies can control the access to a bucket, and its objects, based on user identities, source IP addresses, time and date, and some other parameters.

S3 provides an API that allows users (consumers) to be authenticated using their own (enterprise specific) *identity provider*. In order for an operation to be performed the following messages have to be exchanged:

1) $Consumer \rightarrow Identity\ Provider$: Authenticate
2) $Identity\ Provider \rightarrow Amazon\ Token\ Service$: Request Token
3) $Amazon\ Token\ Service \rightarrow Identity\ Provider$: Token
4) $Identity\ Provider \rightarrow Consumer$: Token
5) $Consumer \rightarrow Amazon\ S3$: Operation, Token

It can be seen that the same number of messages is required, as in our system. Nevertheless, in the S3 system the authorization is performed by Amazon and not by the identity provider, therefore access control policies have to be stored in an Amazon server. This, combined with the fact that policies are defined using Amazon's specific policy definition language, creates a "lock-in" risk. Moreover, all the users who are identified by their own identity provider are considered to have the same role (i.e., "federated users"), limiting the flexibility of the access control policies. Finally, a secret has to be shared between the identity provider of the user and Amazon's token service, in order for steps 2 and 3 to take place successfully.

VI. Discussion

So far we have explored the possibilities that our solution offers in a "traditional" usage model: an enterprise that uses Cloud computing for outsourcing data storage and computations. However, the introduction of a new role, that of the ACP, and the decoupling of the data storage and access control assessment functions creates many new business opportunities.

One area that can benefit from our solution is that of B2B applications. Suppose that enterprise A wants to offer access to some of its (Cloud-based) services to a department of enterprise B. Enterprise B can expose a URI_{ap} that authenticates and authorizes the users of that particular department. Enterprise A can use this URI_{ap} in order to protect the shared services. With this, enterprise A can perform access control without learning anything about the internal user management system of enterprise B. Enterprise A may also offer services for the customers of enterprise B using a similar approach.

Our solution also creates a new business opportunity. We envision that a new market can arise due to our solution, that

[7]Google provides a SAML based SSO system that can be used to integrate enterprise specific authentication systems, but only in Web applications.

of the access control providers. 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 services and 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.

VII. Conclusions

In this paper we proposed a solution to a thorny problem that prevents Cloud technology adoption: that of access control. The proposed solution enables data owners to outsource data storage and computation, without losing governance of their assets. In our solution access control is provided as a service by a new entity, the Access Control Provider (ACP). Access control as a service relieves Cloud providers from the burden of implementing complex security solutions and enables enterprises to deploy their own specific access control mechanisms. We demonstrated the feasibility of our scheme through proof of concept implementations. In particular, we implemented our system as an add-on for the open source Cloud stack OpenStack and we developed a Web application that allows the incorporation of our system in Google Drive. We show that our scheme is secure and has significant privacy properties. The proposed system adds minimal overhead, does not require any particular Cloud implementation or ACP structure and, therefore, it constitutes a realistic solution to the problem. Finally, we believe that the proposed solution can open the floor for new exciting applications and business opportunities.

Acknowledgment

This research was supported in part 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, "The notorious nine cloud computing top threats in 2013," 2013. [Online]. Available: https://cloudsecurityalliance.org/
[6] A. Armando, R. Carbone, L. Compagna, J. Cuellar, and L. Tobarra, "Formal analysis of SAML 2.0 web browser single sign-on: breaking the SAML-based single sign-on for google apps," in *Proc. of the 6th ACM workshop on Formal Methods in Security Engineering*, 2008, pp. 1–10.
[7] J. Somorovsky, A. Mayer, J. Schwenk, M. Kampmann, and M. Jensen, "On breaking SAML: Be whoever you want to be," in *Proc. of the 21st USENIX Security Symposium*, vol. 12, 2012, pp. 21–21.
[8] N. Fotiou, A. Machas, G. Polyzos, and G. Xylomenos, "Access control delegation for the cloud," in *Computer Communications Workshops (INFOCOM WKSHPS), 2014 IEEE Conference on*, April 2014, pp. 13–18.
[9] 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.
[10] D. Hardt (ed.), "The OAuth 2.0 authorization framework," *RFC 6749*, October 2012.
[11] 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.
[12] 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.
[13] R. Yavatkar, D. Pendarakis, and R. Guerin, "A framework for policy-based admission control," *RFC 2753*, January 2000.
[14] D. Durham (ed.), " The COPS (Common Open Policy Service) Protocol," *RFC 2748*, January 2000.
[15] 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.
[16] 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.