OAuth and ABE based Authorization in Semi-Trusted Cloud Computing

Anuchart Tassanaviboon
Department of Electrical and Computer Engineering
University of Waterloo
Ontario, Canada
atassana@uwaterloo.ca

Guang Gong
Department of Electrical and Computer Engineering
University of Waterloo
Ontario, Canada
ggong@calliope.uwaterloo.ca

ABSTRACT

In cloud computing, inter-operations between data-storage and web-application providers can protect users from locking their data and applications into a single cloud provider. Currently, web-based access control standards are applicable only when data owners and cloud service providers are in the same trusted domain. Unfortunately, this condition cannot be satisfied in untrusted clouds, where cloud providers may access sensitive information without authorization. Most previous studies require end-user certificates or specific APIs and depart from existing standards. In this paper, we propose a new authorization scheme **(AAuth)** that builds on the OAuth standard by leveraging ciphertext-policy attribute based encryption and an ElGamal-like mask over the HTTP protocol. Our scheme provides end-to-end encryption and ABE-based tokens to enable authorization by both authorities and owners and to move policy enforcement from clouds to destinations. With our user-centric approach, owners can take control of their data when it rests in semi-untrusted cloud storage. Moreover, with most cryptographic functions delegated from owners to authorities, owners can gain computation power from clouds. Security analysis shows that our scheme maintains the same security level as the original encryption scheme and protects users from exposing their credential to application providers. In our extensive simulation, AAuth's greater overhead was balanced by greater security than OAuth's. Furthermore, our scheme works seamlessly with storage providers by retaining the providers' APIs in the usual way.

Categories and Subject Descriptors

D.4.6 [**Operating System**]: Security and Protection—*Access controls*; K.6.5 [**Management of Computing and Information Systems**]: Security and Protection—*Unauthorized access*

General Terms

Security

Keywords

Authorization, access control, cloud computing

1. INTRODUCTION

In computer security, access control is an essential component in approving the access privileges of consumers. Traditionally, a centralized access control system has only two roles: 1) clients who act as consumers on behalf of resource owners, 2) a centralized server that authenticates consumers and authorizes access to resource according to consumers' capabilities or resources' Access Control Lists (ACLs). With the development of Single Sign-On (SSO) systems like Kerberos [8], an authenticator and an authorizer can be separated from a centralized server, while the owner and consumer are left to act in the same client.

Web application ubiquity on the Internet has made, sharing resources between HTTP-service providers a dramatic requirement for both users and providers. For example, a photo lab prints a customer's pictures from her Flickr. Consequently, an IETF working group is exploring an authorization standard, called OAuth [5] [6], that allows one service provider (aka a third-party client), who is not an owner, to access the resources of another service provider (aka a resource server) on behalf of the owner without exposing the owner's secret credentials to the third-party client. To this end, the OAuth uses HTTP redirections and tokens to introduce an authorization layer, which separates an owner's role from a third-party client and provides a secure way for an owner to allow one provider to access his/her resources that are hosted by other providers.

Although standards exist for authorization, they all rely on that users trust their service providers to keep and process their data. This assumption is rational if all parties are in the same trusted domain or owners never expose sensitive data outside their on-premise data center. With the emergence of cloud computing, this assumption may not be true since IT infrastructure is outsourced to public clouds whose Cloud Service Providers (CSPs) may be dishonest. In this hostile environment, users must trade control of data for flexibility, scalability, and reduced expenses from untrusted CSPs. Finally, users may decide to lock in a single CSP if they believe it can mitigate concerns about data security. This belief impedes the open-cloud concept, and no one can

guarantee security because of the following problems: 1) An authorizer may arbitrarily grant accesses to its conspirators. 2) Resource servers hosting sensitive information may reveal this information. 3) A resource server may refuse to obey predefined capabilities or ACLs. 4) On large-scale like the Internet, assuming that all participants are in a single trusted domain is infeasible and non-scalable.

Rather than suffering from vendor lock-in or hoping for trusted CSPs, we propose a novel authorization scheme **ABE-based Authorization (AAuth)** based on the OAuth authorization standard and Ciphertext-Policy Attribute Based Encryption (CP-ABE): 1) AAuth employs a user-centric approach by using a web browser preloaded with CA certificates as an HTTP trust-platform for owners. 2) Our scheme also replaces the traditional tokens with ABE-based tokens (termed ABE-tokens). Thus, the ACLs are bundled in ciphertext and enforced when consumers try to decrypt the ciphertext with the private keys in ABE-tokens. 3) To achieve our user-centric approach, we allow owners to limit the lifetime and scope of ABE-tokens by adding other owner-controlled attributes, called confined attributes. We also modify the CP-ABE scheme proposed by Bethencourt [1] so that an authority, an authorizer, and an owner contribute to the key generation. 4) Owners delegate their secret shares of times slot to an authorizer. This delegation allows an authorizer to re-encrypt the header of protected files to avoid owners staying on-line to limit token lifetimes.

With the modification and combination of OAuth, CP-ABE, ElGamal-like masks, proxy re-encryption, and lazy re-encryption, AAuth achieves a user-centric and end-to-end cryptographic functions that support the following features: 1) Access grants are made with cooperation among owners, an authorizer, and authority(s). 2) Access policies are bundled into resources by owners and enforced at the destination. 3) Data is encrypted when resting with a resource server then decrypted at the destination. 4) AAuth integrates existing standards and available cloud entities for simplicity and compatibility. 5) Moreover, AAuth is designed in a distributed fashion to gain efficiency and scalability from cloud servers.

The rest of this paper is organized of follows: Section 2 discusses our models and assumptions. Section 3 describes definitions and notations for our scheme. Section 4 presents our construction, procedures and protocols. Section 5 analyses our scheme in terms of security from internal and external adversaries. Section 6 evaluates the performance of our scheme by using a simulation and compares our scheme to existing standards. Finally, Section 8 concludes the paper.

2. SYSTEM AND ADVERSARY MODELS

2.1 System Model

There are five main parties in the system:

Data owners (O): (owners for short) entities, i.e., end-users or software applications, who have resource ownerships and the right to grant access to protected data.

Cloud servers (S): (servers for short) cloud-storage or cloud-database providers that host protected data and provide basic data services, i.e., read ,write, and delete.

Consumers(C): web or traditional application providers that use owners' data to provide services to the owners.

Authority(AA): trusted organizations or agencies who legitimately define descriptive attributes to eligible consumers.

The authorizer(AZ): the server who runs the AAuth protocol, then issues ABE-based tokens to eligible consumers.

To protect data from unauthorized access, an owner encrypts it with an access policy defined over public-key components, then stores it in a server. When a consumer wants to use data, it asks an owner and authorities jointly to issue an ABE-token. Thus, only consumers who satisfy the policy can decrypt the data file. We assume that without end-users' certificates, the authentication systems, e.g., user-password databases, Active Directory/LDAP, or OpenID [9], are available for authorizers to verify owners. Meanwhile, all service providers, i.e., authorizers, authorities, consumers, and servers, register for public-key certificates from Certificate Authorities (CA) in order to support SSL/TLS security channels. For simplicity, we also assume that only owners have read and write privilege on their data, while consumers can only read the authorized data.

2.2 Adversary Model

In this scenario, our semi-trusted environment means that although no entity trusts the others, everyone trusts the protocol. That is, all entities will follow the proposed protocol in general because protocol violation is easy to detect. However, each entity may exploit some threats to attack the system as follows. 1) We assume servers are trusted to provide data-services properly but may be curious about sensitive information and prone to reveal data to ineligible parties. 2) The authorizer may disobey owners' orders to issue tokens, or issue arbitrary tokens to its conspirators. 3) Consumers may try to get unauthorized files from honest servers by fabricating tokens to obtain unauthorized access or resubmitting previous tokens (replay attacks). 4) Without user public-key certificates, owners may propose tokens on behalf of others. 5) Internet users may launch general network attacks on encrypted data or tokens. However, we assume that the communications among CSPs are secure and authentic under SSL/TLS secure channels. Adversaries do not have enough computing power to break cryptographic primitives. Based on the above system model, adversary model, and design goals, we next explain how to construct our scheme and how the protocols are manipulated.

3. DEFINITIONS AND NOTATIONS

Constructing AAuth requires five main components: attributes, access policies, access trees, meta-data, a modified CP-ABE scheme and archive files as follows.

3.1 Attributes

AAuth divides the attribute universe into two disjoint sets: confined and descriptive. To restrict the scope and limit the lifetime of tokens, the confined attributes are mandatory and issued by an authorizer on behalf of owners. The syntax and semantics of confined attributes are defined according to token restrictions, and reserved to disjoin them from descriptive attributes, defined below. Hence, an authorizer publishes the syntax as $\langle attribute \rangle = \langle value \rangle$, and the semantics are as follows:

FILE-LOC $= URI$**:** a file identifier consisting of $URL/absolute\ path/filename$;

OWNER $= ownerId$**:** the identifier of a file owner;

PERMIS $= \langle r|w \rangle$**:** file permissions, where 'r' is read only and 'w' is write;

SEC-CLASS $= \langle 1-5 \rangle$**:** a security class of a file, defined in ascending order;

TIMESLOT $= yyyy/mm/dd/hh/nn$**:** the digits of year, month, date, hour, and minute in the timeslot.

Since our ABE-tokens are a set of private keys, rather than issue a token for each file, it is more flexible and efficient to issue a token for multiple files or time slots when the other attributes are shared in common. Furthermore, the granularity of a time slot can be adjusted by masking the fine-grained components with * in ascending order. However, every token of the same file must have the same granularity level. Note that for some cloud storage, FILE-LOC attributes are not physical locations, for instance, objects in a bucket of Amazon S3.

On the other hand, descriptive attributes, which describe consumer characteristics, are defined by authority(s) who monitor and control the consumer. We define the syntax of descriptive attributes as $\langle attribute \rangle @ \langle url \rangle = \langle value \rangle$, where $\langle url \rangle$ is a URL of the authority issuing the attribute. However, authorities freely define the semantics of attributes under their control, then publish in any public server.

3.2 Access Policies

To encrypt protected data with access policies, we first define the policy in a boolean algebraic expression that combines confined and descriptive attributes together at the root node. Therefore the algebra is constructed by ANDing each confined-attribute term and the whole set of descriptive-attribute terms as follows:

$$\begin{aligned} Policy\ \mathbb{A} = & [\text{FILE-LOC}]\ \text{AND}\ [\text{OWNER}]\ \text{AND} \\ & [\text{SEC-CLASS}]\ \text{AND}\ [\text{PERMIS}]\ \text{AND} \\ & [\text{TIMESLOT}]\ \text{AND} \\ & [(\text{OWNER@AUTHOR})\ \text{OR} \\ & (Descriptive\ Boolean\ Algebra)]. \end{aligned}$$

To ignore the descriptive term when an owner accesses her own data, a special attribute 'OWNER@AUTHOR' will be OR with the descriptive term. Hence, an owner must request 'OWNER@AUTHOR' from an authorizer when the owner wants a token that does not depend on descriptive attributes.

3.3 Access Tree

From [10], any monotonic access policy can be represented by an access tree, a conjunction between a linear secret-sharing scheme and tree structure. In access trees, each non-leaf node is a (t, n) threshold gate described by its n child nodes and the threshold value t of a $(t-1)$-degree polynomial. Thus, each non-leaf node can represent an 'AND' or 'OR' gate in an access policy by using an (n, n) or $(1, n)$ threshold gate respectively. Meanwhile, each leaf-node is associated with an attribute term in the access policy. Figure 1 shows the three primitive constructions of non-leaf nodes in an access tree: a 1-of-3 threshold gate (an OR gate), a 2-of-3 threshold gate, and a 3-of-3 threshold gate (an AND gate). Basically, in a top-down manner, we can construct an access tree according to a monotonic access policy as in the following algorithm.

1. Staring with the root node, to create a t_R-degree polynomial $q_R(\cdot)$, an algorithm sets the point at zero $q_R(0) = s$ for a secret value $s \in \mathbb{Z}_p$ and randomly chooses other $(t_R - 1)$ points.

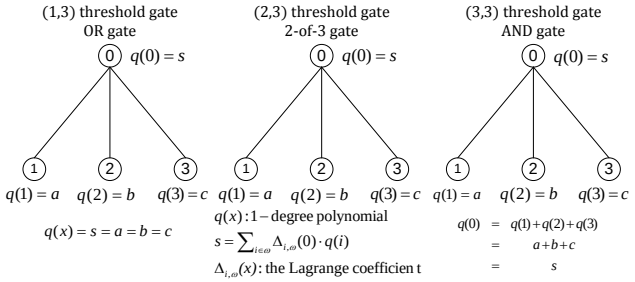

Figure 1: $(1,3)$**,**$(2,3)$**, and** $(3,3)$ **Threshold gates**

2. For other nodes x, the algorithm sets $q_x(0) = q_{parent(x)}(index(x))$ and randomly chooses other $(t_x - 1)$ points to define t_x-degree polynomial $q_x(\cdot)$.
3. At each leaf node x, an associated attribute $att(x)$ is assigned to node x.

Here $parent(x)$ denotes the parent node of node x, $att(x)$ denotes the attribute associated with node x if x is a leaf node, $index(x)$ denotes an index number associated with node x and the index number is assigned uniquely and ascendantly along the tree from the root node ($index = 0$) to the last leaf nodes. As a result, the access policy $\mathbb{A}$, consisting of both confined and descriptive attributes, will convert to an access tree τ as shown in Figure 2.

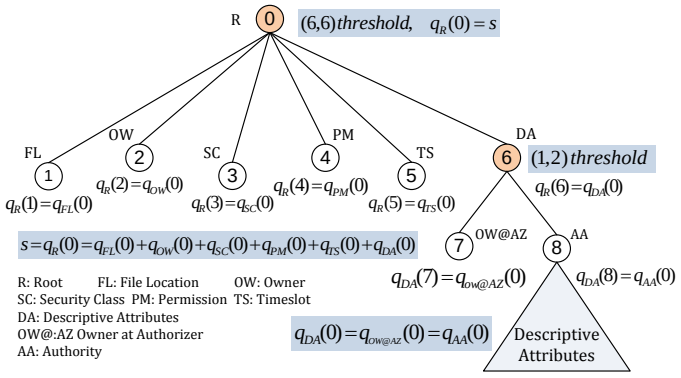

Figure 2: Top level of an AAuth access tree

3.4 Modified CP-ABE

Bethecour *et al.*[1] proposed a CP-ABE construction based on a bilinear map $e : \mathbb{G}_1 \times \mathbb{G}_1 \to \mathbb{G}_2$, a Linear Secret-Sharing (LSS) scheme for access-tree construction, and a hash function $H : \{0,1\}^* \to \mathbb{G}_1$ that maps any attribute binary-string to a random element in $\mathbb{G}_1$. For adopting their scheme in our distributed scheme, we modify the original scheme in the following five algorithms:

$Setup(k)$. According to a security parameter k, the algorithm chooses a bilinear map $e : \mathbb{G}_1 \times \mathbb{G}_1 \to \mathbb{G}_2$ of prime order p with generator g of group $\mathbb{G}_1$ and a hash function $H : \{0,1\}^* \to \mathbb{G}_1$. The authorizer chooses a random exponent β for a Master Secret Key $MSK = \langle \beta \rangle$ and publishes a Master Public Key $MPK = \langle \mathbb{G}_1, g, h = g^\beta, f = g^{1/\beta} \rangle$. Meanwhile, each owner chooses a random exponent α for an Owner Secret Key $OSK = \langle g^\alpha \rangle$ and publishes an Owner Public Key $OPK = \langle e(g,g)^\alpha \rangle$ in the user-directory of the authorizer.

$Encrypt(MPK, OPK, m, \tau)$. This algorithm encrypts a message m under the access tree τ. An owner first chooses a

random value $s \in \mathbb{Z}_p$, then constructs an access tree τ according to $q_R(0) = s$ and an access policy $\mathbb{A}$. Let Y be the set of leaf nodes in τ. Then a ciphertext CT is computed by

$$CT = \langle \tau, \tilde{C} = m \cdot e(g,g)^{\alpha s}, C = h^s, \\ \forall y \in Y : C_y = g^{q_y(0)}, C'_y = H(att(y))^{q_y(0)} \rangle.$$

$KeyGen(MSK, OSK, \omega)$. The algorithm will take as input a set ω of attributes. The authorizer chooses a random value $r \in \mathbb{Z}_p$ and random values $r_i \in \mathbb{Z}_p$ for each confined attribute $i \in \omega'$; an authority chooses random values $r_j \in \mathbb{Z}_p$ for each descriptive attribute $j \in \omega''$; and an owner chooses a random value a. Let $\omega = \omega' \cup \omega''$ denote the attribute set and $k \in \omega$ denote each attribute. With an ElGamal-like mask, the authority, the authority, and the owner jointly compute a private key SK for a user without revealing their own private keys to each other as follows:

$$SK = \langle D = g^{(\alpha + ra)/\beta}, \forall k \in \omega : D_k = g^{ra} H(k)^{r_k}, D'_k = g^{r_k} \rangle.$$

$Delegate(SK, \tilde{\omega})$. The algorithm takes as input a secret key SK, which is for an attribute set ω and another attribute set $\tilde{\omega} \supseteq \omega$. The algorithm first chooses a random value $\tilde{r}$ and $\{\tilde{r}_l \mid \forall l \in \tilde{\omega}\}$. Then it creates a new private key $\widetilde{SK}$ for an attribute set $\tilde{\omega}$ as

$$\widetilde{SK} = \langle \tilde{D} = D \cdot f^{\tilde{r}}, \forall l \in \tilde{\omega} : \tilde{D}_l = D_l \cdot g^{\tilde{r}} \cdot H(l)^{\tilde{r}_l}, \tilde{D}'_l = D'_l \cdot g^{\tilde{r}_l} \rangle.$$

$Decrypt(CT, SK)$. This algorithm is a recursive algorithm over the access tree τ for a ciphertext CT. Let $DecryptNode$ (CT, SK, x) denotes a node algorithm, which takes as input a ciphertext CT, a private key SK, and a node x in τ. From the node algorithm, $Decrypt(CT, SK)$ can be computed by calling the node algorithm from the root node R of the access tree τ. If the access tree τ is satisfied by ω, then we have

$$A = DecryptNode(CT, SK, R) = e(g,g)^{raq_R(0)} = e(g,g)^{ras}$$

by recursive computing and interpolation. Now the decryption can be computed by

$$Decrypt(CT, SK) = \tilde{C} / (e(C, D)/A) \\ = \tilde{C} / (e(h^s, g^{(\alpha + ra)/\beta}) / e(g,g)^{ras}) = m.$$

3.5 Meta-data

AAuth maintains meta-data in a user-directory and file-directory hosted by an authorizer. A user-directory consists of the owner lists of which IDs and credentials can be maintained by a local database or an external identity management center, e.g., LDAP, Active Directory, or OpenID. A file-directory is a repository that an authorizer uses to maintain the last authorized time slot, last share $q_{TS}(0)$ of TIMESLOT node, and granularity in each file, information used for time slot synchronization (Subsection 4.7).

3.6 Archive file

Although we can directly use an ABE scheme to encrypt protected data, AAuth separates encryption into two levels: header and data. This separation can improve efficiency because 1) asymmetric-key encryption itself is not as efficient as symmetric-key encryption, 2) our policy change and time slot synchronization do not necessitate data re-encryption, so it can be delayed until the data is changed. To this end, we encapsulate protected data in an archive file consisting of a header encrypted with an ABE key and the protected data encrypted with a symmetric key encryption.

$$\langle Archive \rangle = \{\langle Header \rangle\}_{ABE} \parallel \{\langle Data \rangle\}_{KE} \parallel \\ \langle \mathbb{A} \rangle \parallel \langle IntegTag \rangle,$$

Next, we define the parameters in a header as follows:

$$\langle Header \rangle = \langle FileDesc \rangle \parallel \langle EncryMeth \rangle \parallel \langle IntegMeth \rangle \parallel \\ \langle KE \rangle \parallel \langle KV \rangle \parallel \langle \mathbb{A} \rangle$$

$\langle FileDesc \rangle$**:** the description of protected-file content.

$\langle EncryMeth \rangle$**:** a symmetric-key algorithm is used to encrypt protected data, such as AES-128, AES-192, RSA-1024, RSA-2048, etc..

$\langle IntegMeth \rangle$**:** a set of algorithms is used to generate an integrity tag, such as RSA-MD5, RSA-SHA1, DSA-MD5, DSA-SHA1, etc..

$\langle KE \rangle$**:** a symmetric key is used to encrypt protected data.

$\langle KV \rangle$**:** an asymmetric key is used to verify an integrity tag.

$\langle \mathbb{A} \rangle$**:** an access policy in plaintext form.

$\langle IntegTag \rangle$**:** an integrity tag generated from clear $\langle Header \rangle$ and encrypted data.

Note: the notation introduced in this section and shown in Table 1 is used throughout the paper. In addition to all components in this section, our scheme requires procedures and protocols described in the next section to provide security in untrusted clouds.

Table 1: Notation used in our scheme explanation

Notation	Description
MPK, MSK	System Public and Private keys of an authorizer
OPK, OSK	System Public and Private keys of owners
Att_x, PK_x	Attribute x, User public key for PK_x
SK	User private key consisting of two parts: common D and {part1 D_k, part2 D'_k} for each Att_k
D_i, D'_i	Part1 and part2 key components for confined Att_i
D_j, D'_j	Part1 and part2 key components for descriptive Att_j
$\hat{D}_i, \hat{D}_j$	The partial part1 of confined and descriptive key components D_i, D_j
$\{U\}_X$	The ciphertext of a string U encrypted with X's public key
$[U]_X$	A string U attached with digital signature over a string U generated with X's private key

4. ABE-BASED AUTHORIZATION (AAUTH) SCHEME

AAuth mainly requires three off-line procedures (i.e., setup, file encapsulation, and file decapsulation) and four on-line protocols (i.e., service request, token request, file access, and time slot synchronization). Optionally, AAuth can provide key delegation, policy change, and data update. This section explains these procedures and protocols.

4.1 Setup

Before AAuth starts to provide inter-operations, an authorizer and owners must initialize the system parameters in Procedure 1.

Procedure 1. Setup Phase

1. An authorizer chooses a security parameter k and runs the CP-ABE algorithm $Setup(k)$ that outputs a bilinear group $\mathbb{G}_1, \mathbb{G}_2$, a bilinear map e, a generator g of $\mathbb{G}_1$, and a hash function H. All outputs in this step are published in a public server.

2. The authorizer chooses a random value $\beta \in \mathbb{Z}_p$ and keeps it secretly, then generates a master public-key $MPK = \langle \mathbb{G}_1, g, h = g^{\beta}, f = g^{1/\beta} \rangle$ and publishes it in a public server.
3. Each owner contributes to key generation by randomly selecting $\alpha \in \mathbb{Z}_p$, generates an owner public-key $OPK = e(g,g)^{\alpha}$ that is published in a user-directory of the authorizer (not really published), then keeps an owner private-key $OSK = g^{\alpha}$ secretly.

Our explanation for the scheme is based on the following example from the OAuth standard:
Example 1. Jane (an owner) has recently uploaded some private photos (protected resources) to her sharing site 'photos.com' (a server). She would like to use the 'printer.com' website (a consumer) that is certified by a trust authority (authority.org), to print her photos. Jane does not have a public-key certificate and does not wish to share her identity and credentials with 'print.com'. However, she has registered with a trusted mail provider (mail.net) that also provides OAuth services for its subscribers.

4.2 File Encapsulation

Before uploading files to a cloud server, an owner encrypts and encapsulates data files into archives file by using Procedure 2.
Procedure 2. File encapsulation phase

1. Define an access policy $\mathbb{A}$ from both confined and descriptive attributes as follows:

```
# Confined attributes
[FILE-LOC=http://photos.com/2010/brunce/pic-1]
AND  [OWNER=Jane@photos.net]  AND  [SEC-CLASS=3]
AND  [TIMESLOT=2011/06/27/13/**]  AND  [PERMIS=r]
AND  # Descriptive attributes
      [(OWNERe@mail.net=Jane@mail.net) OR
      [(NAME@authority.org=printer.com) AND
       (SERVICE@authority.org = print) AND
       (LOCAT@authority.org = canada) OR
       (TRUST-LEV@authority.org = 3)]].
```

   Then convert $\mathbb{A}$ to an access tree τ by using the algorithm in Subsection 3.3.
2. Randomly choose an encryption key KE, and a generate signature key KS and verification key KV, then define all parameters in the header H according to the format in Subsection 3.6.
3. Encrypt a data file with the key KE, and generate an integrity tag from the encrypted data-file and the clear header with the key KS.
4. Encrypt the header H by choosing a random value s then using the CP-ABE algorithm $Encrypt(MSK, m, \tau)$ to compute a ciphertext CT.
5. Construct an archive file from the encrypted header and encrypted data file, the access policy, and the integrity tag in the format described in Subsection 3.6. Then the owner stores the archive file in the cloud server using a regular API.

Note that we compute an integrity tag from an unencrypted header to avoid the effect of time slot synchronization (see Subsection 4.7). We have now described how to create and store ciphertext; next we show how to obtain an ABE token and how to retrieve and decrypt the ciphertext.

4.3 Service Request

In this scenario, owners first ask consumers for services that require the owners' data. Before a consumer can access data, it must ask an authorizer for ABE-tokens issued on behalf of owners. Then the consumer uses a private key in a token to prove the authorization by performing a challenge-response protocol with a cloud server. If the verification succeeds, the server will provide the archive file according to the token. So our scheme starts from an owner requesting a service from a consumer, e.g., to print photos, that require the owner's data files, as described in Protocol 1 and shown in Figure 3. Note that owners registered with an authorizer and consumers registered with an authority are beyond the scope of this work.
Protocol 1. Service request phase

1. First an owner sends a commend $REQ\text{-}PRT$ to request a printing service from the consumer 'printer.com'.
2. Since the service requires the owner's data, the consumer sends a server a file location with a command $REQ\text{-}POL(FileLoc)$ to request the required file's access policy from the server 'photos.com'.
3. The server extracts the access policy '$\mathbb{A}$' from the archive file and signs $\mathbb{A}$. Then the signed access-policy $[\mathbb{A}]_S$ is replied to the consumer.
4. To initiate a token request, the consumer sends the owner an HTTPS-redirect command RED to redirect the owner's user-agent to an authorization page (https://authorizer.net/authorize) on an authorizer. To this end, the consumer ID and a redirection URI (https://printer.com/ready) signed by the consumer received earlier are included in the command as $RED([ID_C, RED\text{-}URI]_C, [\mathbb{A}]_S)$.

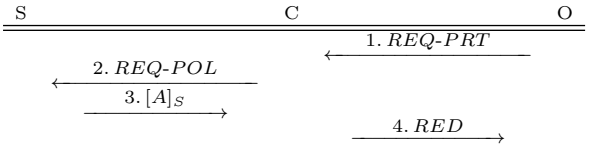

Figure 3: The message flow of protocol 1: service request

4.4 Token Request

Once an owner's user-agent is redirected to an authorizer, the authorizer first verifies the owner's ID and credentials. If the authentication succeeds, the owner and authorizer cooperate to issue an ABE-token for the consumer as pictured in Figure 4 and shown in Protocol 2.
Protocol 2. Token request phase

1. From the redirect command, an owner's browser will pass $[ID_C, RED\text{-}URI]_C, [\mathbb{A}]_S$ to an authorizer.
2. The authorizer thus uses an HTTP form page to authenticate the owner and receive the owner's decision.
3. The owner sends her ID and credentials to authenticate herself to the authorize, then defines confined attributes that she allows and sends these to the authorizer.
4. If the authentication succeeds, the authorizer generates an authorization code *Autz-code* (a nonce) and sends the authority a command $REQ\text{-}DES1$ $(ID_C, RED\text{-}URI, Authz\text{-}Code)$ to request descriptive components.

5. The authority retrieves the consumer's attributes, then generates and signs the partial part-1 $[\{\hat{D}_j\} = \{H(j)^{r_j}\}]_{AA}$ of descriptive components, and replies it to the authorizer. The part-2 $\{D'_j\} = \{g^{r_j}\}$ is directly sent to the consumer in the file access protocol (see Subsection 4.5).
6. The authorizer generates the partial part-1 $\{\hat{D}_i\} = \{H(i)^{r_i}\}$ and part-2 $\{D'_i\} = \{g^{r_i}\}$ according to the owner's decision, and randomly selects $r \in \mathbb{Z}_p$. Then the part-1 and the part-2 signed by the authorizer, g^r, and the descriptive part-1 $\{\hat{D}_j\}$ received earlier are sent to the owner.
7. The owner verifies whether the confined components are associated with the attributes by computing bilinear pairing $e(H(i)^{r_i}, g) \stackrel{?}{=} e(g^{r_i}, H(i))$. If the verification succeeds, the owner randomly chooses a, computes $g^{\alpha+ra}$ from a, g^r, and her $OSK = g^\alpha$, then replies to the authorizer with the result $g^{\alpha+ra}$.
8. Now the authorizer is ready to generate the common part D from the MSK β and $g^{\alpha+ra}$ received earlier. In this cooperation, by an ElGamal-like mask, the authorizer only knows $g^{(\alpha+ra)/\beta}$, and the owner only knows g^{ra}. The authorizer encrypts the common part $D = g^{(\alpha+ra)/\beta}$ and the authorization code *Authz-code* with the consumer's public key and signed with the authorizer's private key, before sending the owner a redirect command $RED([\{g^{(\alpha+ra)/\beta}, Authz\text{-}Code\}_C]_{AA})$ to redirect the user-agent back to the consumer at the redirection URI endpoint.
9. The owner binds both the partial part-1 $\{\hat{D}_i\}, \{\hat{D}_j\}$ of confined and descriptive components by multiplying them with g^{ra} and sends all keys $\{D_i = g^{ra}H(i)^{r_i}\}$, $[\{g^{r_i}\}]_{AZ}$, $\{D'_j = g^{ra}H(j)^{r_j}\}$ to the consumer.
10. The consumer sends the authority a command $REQ\text{-}DES2(ID_C, RED\text{-}URI, Authz\text{-}Code)$ to authenticate itself and to request the descriptive part-2 .
11. If the authentication succeeds, the authority will reply to the consumer with the part-2 $\{D'_j = g^{r_j}\}$ of descriptive components.

Now the consumer occupies an ABE token and is ready to access data files in the next protocol.

4.5 File Access

After a consumer obtains an ABE token, the consumer must prove private-key possession by performing a challenge-response with a cloud server, as shown in Figure 5 and described in Protocol 3.

Protocol 3. File access phase

1. First a consumer sends a server a command $REQ\text{-}FILE$ $(FileLoc)$ that includes a file location to request a challenge from the server.
2. The server generates a challenge value $chall = Encrypt$ $(MPK, nonce, \mathbb{A} \setminus Att_{TS})$ from a nonce encrypted using CP-ABE encryption with the access policy, excluding the time slot attribute, and sends it to the consumer.
3. Now the consumer has a complete ABE private-key that satisfies the access policy and so can generate a response $Resp = Decrypt(Chall, SK)$ by using the ABE decryption algorithm and replies to the server with the response *Resp*.
4. The server verifies the challenge-response by $Resp \stackrel{?}{=} nonce$. If it succeeds, the server replies to the consumer with the archive-file *Arcvive*.

Now the consumer has both the archive file and the ABE private-key, so the next step is data verification and decryption of an archive file.

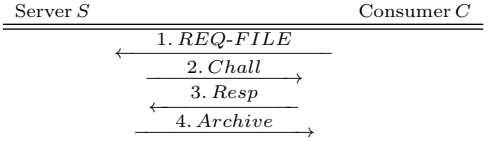

Figure 5: The message flow of protocol 3: file access

4.6 File Decapsulation

Before file decryption, a consumer must verify the integrity of the archive file. Thus, a consumer performs decapsulation of an archive file as in Procedure 3.

Procedure 3. File decapsulation phase

1. To extract all parameters from the header, the consumer decrypts the header by using the CP-ABE algorithm $Decrypt(CT, SK)$.
2. The data integrity of the decrypted header and encrypted data must be checked by the algorithm and key, which are defined in $IntegMeth$ and KV. Then the consumer moves to the next step if the integrity is valid.
3. The $FDesc$ and $\mathbb{A}$ parameters are used to check whether the archive file is the correct one.
4. The encrypted data is decrypted by the algorithm and key, which are defined in $CrypMeth$ and KE.

Using all the above procedures, the consumer can access the plaintext of a data file. In addition, the consumer can verify the integrity and correctness of the data. In the remaining subsection, we present other procedures that facilitate our scheme.

4.7 Time slot Synchronization

In AAuth, the access permission in each time slot is granted by matching the key component $D_{ts} = g^r H(ts)^{r_{ts}}$ in SK to the ciphertext component $H(ts)^{q_{TS}(0)}$ in the access tree (see Subsection 3.3). Hence, AAuth controls token lifetime by defining the key component(s) of the time-slot attribute in a token, and automatically re-encrypting the header of a file with the current time slot attribute. As a result, a consumer can access a file only in the time slot(s) defined in the token. To this end, an owner (encrypter) must send the share $q_{TS}(0)$ of the time slot attribute to an authorizer when performing file encapsulation. For each time slot, the authorizer maintains the last time-slot share $q_{TS}(0)$ of each file in its file directory in order to compute two ciphertext components C_y, C'_y and two update values $h^{\tilde{s}}, e(g,g)^{\alpha\tilde{s}}$. These components and values are sent to a cloud server to update ciphertext components and re-mask the header. Thus, the previous mask $e(g,g)^{\alpha s}$ which other consumers (decrypters) have already occupied will be disabled. Note that re-masking header does not affect integrity tags because the tags are computed from the unencrypted headers. Updating ciphertext components and re-masking headers must be performed by cooperation between an authorizer and a server in Protocol 4 in real-time.

Protocol 4. Time slot synchronization phase

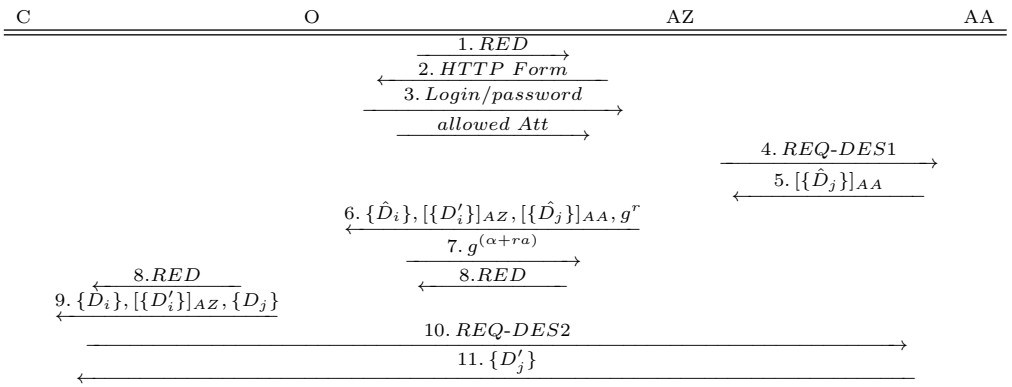

Figure 4: The message flow of protocol 2: token request

1. In each time slot t, an authorizer chooses a new random value $\tilde{s}(t)$, computes a new share $q_{TS}(0,t) = q_{TS}(0,t-1) + \tilde{s}(t)$ of a TIMESLOT node, then saves the new share as the last share in the file directory.
2. New ciphertext components for a new time slot can be computed by $C_{TS}(t) = g^{q_{TS}(0,t)}, C'_{TS}(t) = H(Att_{TS}(t))^{q_{TS}(0,t)}$, where $Att_{TS}(t)$ is the string of the *t-th* time slot.
3. Two update values $h^{\tilde{s}(t)}$ and $e(g,g)^{\alpha\tilde{s}(t)}$ are computed from $MPK = g^{\beta}, OPK = e(g,g)^{\alpha}$.
4. Then the ciphertext components and update values are sent to a server.
5. The server replaces two ciphertext components C_{TS}, C'_{TS} with the received components according to the current time slot.
6. The server also updates the value $C(t) = C(t-1) \cdot h^{\tilde{s}(t)} = h^{s(t-1)+\tilde{s}(t)}$ and re-masks the header $\tilde{C}(t) = C(t-1) \cdot e(g,g)^{\alpha\tilde{s}(t)} = m \cdot e(g,g)^{\alpha(s(t-1)+\tilde{s}(t))}$ in a ciphertext.

The above procedure obviously retains the consistency between the TIMESLOT share in the access tree and the secret valve masking the header. That is, in each time slot t, its share $q_{TS}(0,t)$ can be combined with the other shares to construct the corresponding secret value $s(t)$. This fact arises because the root node in our access tree is an AND gate, i.e., (n,n) threshold gate, which causes $s = q_{FL}(0) + q_{OW}(0) + q_{SC}(0) + q_{PM}(0) + q_{TS}(0) + q_{DA}(0)$ (see Figure 2). Then the protocol adds two sides of the equation with a new random value $\tilde{s}(t)$. Note that the TIMESLOT attribute in a header and tail is a time stamp of when the protected file was encapsulated, not the time slot that encrypted the header.

Since an authorizer participates in token issues, it knows which time slots are authorized for each file. With this knowledge, the authorizer will not continue synchronization if no other authorization occurs. For example, if TIMESLOT=2011|06|27|13|** and TIMESLOT= 2011|06|27|14|** are authorized, then only the synchronization data, i.e., ciphertext components and update values of time slot 2011|06|27|13|**, 2011|06|27|14|**, and 2011|06|27|15|** can be sent out. The authorizer can also deliver multiple time slots simultaneously by combining the components and values of all time-slots into one message sent to a cloud server. The cloud server can also reduce computing cost by caching the synchronization data received from the authorizer until file access occurs. From the synchronization data in the cache, the server can aggregate update values by multiplying all update values and select the last received components for time slot synchronization. Consequently, both the authorizer and a cloud server can optimize both computation and communication cost by leveraging two cryptographic primitives: proxy re-encryption [2] and lazy re-encryption [7].

4.8 Key Delegation

A consumer may ask another provider to process a data file for which authorization already exists. For example, the web site 'printer.example.com' has already obtained a token with two time slots and two files, e.g.,

```
FILE-LOC=http://photos.com/2010/brunce/pic-1,
FILE-LOC=http://photos.com/2010/brunce/pic-2,
SEC-CLASS=3, PERMIS=r,
TIMESLOT=2011$|$06$|$27$|$13$|$**,
TIMESLOT=2011$|$06$|$27$|$14$|$**.
```

Using the key delegate algorithm of CP-ABE, the web site 'printer.com' can ask the website 'poster.com' to print a poster for a file 'pic-1' in the time slot '2011|06|27|13|**' by generating a new private-key set associated with

```
FILE-LOC=http://photos.com/2010/brunce/pic-1,
SEC-CLASS=3, PERMIS=r,
TIMESLOT=2011$|$06$|$27$|$13$|$**.
```

4.9 Policy Changing

Regardless of confined or descriptive attributes, an archive file header must be re-encrypted to change the access policy. In a naive process, an owner rebuilds a new access policy, re-encrypts a header, recomputes an integrity tag, then asks a server to replace both the header and the tail in the archive file. As an advantage, protected data need not be re-encrypted.

4.10 Data Updating

Updating data is straightforward. An owner re-encrypts protected data with the encryption key KE in the header of an archive file, recomputes an integrity tag, then asks a server to replace the encrypted data and the integrity tag. If both the data and access policy are changed, an owner repeats all steps to rebuilt a new archive file and stores it in a server. However, we can obviously provide write permission to consumers by separating headers into read and write. The

read part is built as usual, while the write part includes a signing key and is encrypted by CP-ABE encryption with a write policy.

5. SECURITY ANALYSIS

We analyze our scheme from the perspectives of internal and external adversaries. For internal adversaries, all entities in the system are considered to be semi-trusted, in the sense that they can exploit threats to subvert authorization control and data security, but still honestly follow the protocol. External adversaries may not run the protocol but try to launch general attacks to violate data security. To analyze the AAuth security from internal attacks, we consider the following actions.

Action of Cloud Server: We consider that cloud servers host sensitive information and are curious about the data or may reveal information to unauthorized consumers. Because the data is stored in ciphertext with an integrity tag, neither cloud servers nor unauthorized consumers can decrypt the data without decryption keys, and they cannot fabricate or modify the data without signature keys. Thus, data confidentiality and integrity are secure even if the cloud server is compromised. Our authorization is performed in an end-to-end manner, an access policy is bundled into a ciphertext by an owner; a consumer must then use a key satisfied by the access policy to decrypt. Hence, the access policy is enforced in the decryption algorithm, not by cloud servers, thereby achieving data confidentiality and integrity.

Action of an Authorizer: In the original Kerberos and OAuth, as well as in cloud servers, an authorizer is another dominating entity since it can issue tokens to anyone without owners' permission. This situation is rational if an authorizer is on an owner's premises, not in a public cloud. Our scheme enables secure authorization in public clouds by allowing owners to contribute to token generation and assign confined attributes. Thus, an authorizer alone cannot generate the common part of SK ($D = g^{(\alpha+ra)/\beta}$) for unauthorized consumers because the authorizer has no knowledge about the owner's SK ($OSK = g^{\alpha}$). In addition, an authorizer cannot arbitrarily generate an ABE token because a confined SK proposed by an owner can be verified by the owner.

Action of Owners: Although an owner id is unlikely to counterfeit the tokens she proposes, she may ask an authority to generate a token with 'OWNER' attributed in other people's name to access files not belonging to him/her. In other words, the owner pretends to be someone else. In this case, an authorizer can easily detect this misbehavior since the owner must be authenticated to an authorizer. Also an owner may fabricate the part-1 of confined and descriptive SK ($H(i)^{r_i}, H(j)^{r_j}$) and combine it with her combining term g^{ra}. In our scheme, the part-2 of confined SK g^{r_i} is signed by an authorizer, and an authority directly sends the part-2 of confined SK g^{r_j} to a consumer. Thus, the owner must compute r_i, r_j from g^{r_i}, g^{r_j} respectively. This problem can be reduced to a Discrete Logarithm Problem (DLP), and so fabrication is unsuccessful.

Action of Consumers: Consumers may modify or fabricate their own tokens, for example, they may change the time slot components in their own tokens. Hence, consumers must try to select the confined-SK component $g^{ra}H(i)^{r_i}$, g^{r_i} for the time slot attribute that can satisfy the policy as the original key component. This problem can be reduced to a Decisional Bilinear Diffie-Hellman (BDH), thereby failing on modification. Our scheme can also resist collusion attacks as the original CP-ABE scheme can because in each authorization, an authority and an owner randomly choose new random values (r, a respectively) to combine confined SKs and descriptive SKs. Therefore, one consumer cannot combine her keys to create more powerful keys, and multiple consumers cannot collude in combining difference keys to create a new key.

Resistance to Other Attacks In addition, we consider general attacks from external adversaries, such as eavesdropping, Man-In-The-Middle (MITM), and Denial-of-Service (DoS), etc..

Eavesdropping and active attacks. Data files are encrypted and signed by owners, then verified and decrypted by consumers thereby performing all cryptographic operations in end-to-end fashion. Therefore, eavesdropping on data files cannot disclose data confidentiality, and active attacks cannot corrupt data integrity. Since the header parameters can be verified from the tag, these parameters can be trusted to check the properties, i.e., owner, location, and security class of the data file to verify the correctness of data properties.

MITM attacks. All service providers and authorities in the system must register with CA for public-key certificates. Hence, the communication in our scheme can be protected by SSL/TLS channels. Therefore, our scheme can resist MITM attacks. Moreover, the part-2 of confined and descriptive SK (D'_i, D'_j) originates from an authorizer and authority(s) respectively. These two parts are sent over different SSL/TLS channels and combined by a consumer, so an adversary must intercept two SSL/TLS sessions at the same time to obtain a complete SK; this situation rarely happens in practice.

Off-line attacks. Our authorization is divided into two levels: token request and file access. In the token request, a consumer must get permission from an owner and qualify with respect to the policy in order to get a satisfactory ABE-token. The consumer then uses the ABE-token to generate a response value for file access. To obtain data files from a cloud server, a consumer must prove the authorization by sending a response value satisfying the challenge value. Adversaries thus cannot obtain ciphertext if they have no authorization, so our scheme can protect encrypted data from off-line attacks.

Credential protection. As in OAuth, an owner can grant assess permission to a consumer, even though the owner does not expose her credentials, e.g., passwords, certificates, information cards, etc., to the consumer.

Certified entities. An authority is a certifier who describes consumers according to their characteristics by issuing descriptive attributes for eligible consumers. Thus, consumers cannot declare forged characteristics to obtain protected data.

PRE primitive. In time slot re-encryption, although a master authority must send two ciphertext components $\tilde{C}_{TS}, \tilde{C}'_{TS}$ and two update values $h^{\tilde{s}}, e(g,g)^{\alpha\tilde{s}}$ to a server, all of them are new random values used to replace or multiply (mask) the existing values, so the server cannot gain any advantages to break cryptographic functions. Note that our model is still based on the same assumption that a server can be

trusted to do data operations, i.e., storing and multiplication.

6. PERFORMANCE EVALUATION AND SIMULATIONS

In this section, we evaluate the cost of three off-line procedures (i.e., setup, file encapsulation, and file decapusulation) and four on-line protocols (i.e., service request, token request, file access, and time slot synchronization) in terms of communication and computation cost. Then we show the performance by simulations.

6.1 Evaluation

Setup procedures. This procedure can be divided into two parts: First an authorizer defines underlying bilinear groups and a hash function, then computes MSK and MPK with two exponentiations on $\mathbb{G}_1$. The second part is individually performed by each owner who computes OSK and OPK with one exponentiation on $\mathbb{G}_1$.

File encapsulation. An owner performs this procedure before uploading files to cloud servers. The computation cost in this procedure results from a symmetric-key encryption for data files, a signature, and a CP-ABE encryption for the header. The first encryption depends on the size of data files, and the signature load is fixed by the signature algorithm, while the CP-ABE encryption causes $2|L|+2$ exponentiations on $\mathbb{G}_1$, where L denotes a set of leaf nodes in an access tree.

File decapsulation. After obtaining an archive file, a consumer decrypts the header with CP-ABE, verifies a signature, and decrypts the data file. The CP-ABE decryption cost is dominated by $2|I \bigcap L| + 1$ pairing operations, where I denotes the number of attributes in the CP-ABE SK. The verification load is also fixed by the verify algorithm. The symmetric-key decryption load also depends on the size of the data file.

Next we analyze on-line protocols in both cryptographic cost and message rounds when compared to the OAtuh standard, as follows.

Service request. This protocol requires two more messages between consumers and servers, and one signing operation at a server.

Token request. The computation of CP-ABE key generation causes $2|I| + 1$ exponentiations. In our scheme, this computing cost is distributed to an authorizer and an authority according to the number of confined and descriptive SKs. Also there are three signings (two at an authorizer, one at an authority) and three verifications (one at an owner, two at a consumer). In addition, an owner computes six pairing operations (based on our construction) to verify confined attributes he/she proposes. For message rounds, this protocol requires six more messages: two authorizer-authority messages, one owner-authorizer message, one consumer-owner message, and two consumer-authority messages.

File access. Unlike OAuth, our protocol has no extended message. However, for a challenge message, the cryptographic cost at a server is CP-ABE encryption ($2|L| + 2$ exponentiations); and for a response message, the computation cost at a consumer is CP-ABE decryption ($2|I \bigcap L| + 1$ pairing operations).

Time slot synchronization. An authorizer updates the time slot of each file until the last authorized time slot (on-demand), and a server can delay re-encryption until the file is requested (lazy re-encryption). Its computation and communication cost are four exponentiations at an authorizer, two writes and two multiplications at a server, and only one message between them. In comparison with CP-ABE encryption ($2|L| + 2$ exponentiations), the cost of time slot re-encryption is much lighter.

6.2 Simulations

Based on the number of attributes, leaf nodes, and our construction (six confined attributes), We summarize the computation cost of on-line protocols of each entity for an ABE-token in Table 2.

Table 2: On-line cryptographic cost

	Signing	Verify	Exponent	Paring
Owner		1	1	5
Consumer		2		$2(\|I \bigcap L\|) + 1$
Authorizer	2		11	
Authority	1		$2\|I - 5\|$	
Server	1		$2\|L\| + 2$	

Table 2 shows that the number of users and attribute universe size do not affect computation cost per token. Most computation cost is expended by a consumer (web-application); some cost is expended by a cloud server and an authority. An owner and a authorizer pay a fixed cost; other entities' costs depend on the number of attributes in a token or policy.

Our protocol trades eight additional-messages for security advantages: user-centric property, 2-level authorization, and end-to-end encryption. To evaluate this trade off, we compare our scheme's communication cost to the original standard (OAuth)'s, without the cryptographic cost. To this end, we built the prototype of AAuth and OAuth protocols on the framework of OMNet++ network simulation 4.1. The simulation conditions are defined as follows: the cloud network has 400 packets/second bandwidth; each owner continuously requests services in exponential distribution; each service request transfers three 256 KB-files as a dummy load; the number of owners (users) ranges between 100 to 700. Figure 6 shows the latency time from the protocol and the dummy load.

Figure 6 shows the latency time of both OAuth and AAuth and the latency time differences between both. Both latency time and difference increase as the number of owners increases from 100 to 700 nodes. The former observation is a general behavior of a limited-bandwidth network. The latter exhibits that the difference has no significance if the number of owners is less than 300 nodes, and the difference increases from 5 to 12 seconds after 300 nodes. The increase occurs because we limit the network bandwidth to 400 packets/second in our simulation model. Compared to OAuth, AAuth strikes an acceptable balance between increased network cost and improved security. Moreover, in real situations, the network among cloud servers should have no limitations.

7. RELATED WORK

The taxonomy of cloud computing services goes by the acronym 'SPI', which stands for Software-as-a-Service, Platform-as-a-Service, and Infrastructure-as-a-service. Recently, many

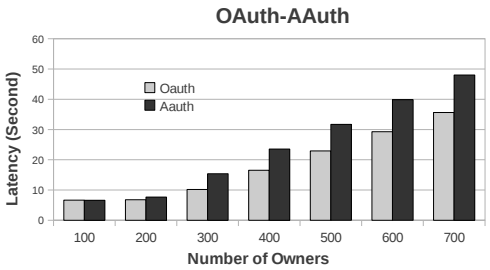

Figure 6: Latency time of OAuth and AAuth

new cloud services have emerged; the most primitive and significant one is Storage-as-a Service. Many researcher are exploring this research area, starting from cryptographic storage, access control, to cloud storage. In 2003, Kallahalla *et al.* proposed Plutus, a cryptographic storage system. This secure file system exploits cryptographic and key management in a decentralized manner in which all operations are performed by clients; the server incurs very little cryptographic overhead. This approach contradicts cloud-computing behavior in which servers have unlimited resources but clients may be restricted devices. Thus, clients may not have enough power for the high complexity of key management in fine-grained access control. In 2009, Bowers *et al.* [3] proposed a framework for a Proof of Retrievability (POR) system that focuses on archival or backup files in cloud storage. Later, they also proposed another POR work [4] that was implemented by a distributed cryptographic scheme and launched on the multi-server of a distributed file system. Wang *et al.* [11] proposed cryptographic-based access control for owner-write-users-read applications, which encrypts every data block of cloud storage and adopts a key derivation method to reduce the number of keys. Yun *et al.* [13] proposed a cryptographic network file system based on MAC tree construction and universal-hash-based-statefull MAC that can guarantee the data confidentiality and integrity of files. In 2010, Yu *et al.* [12] proposed fine-grained and scalable access control in cloud computing that exploits KP-ABE to reduce complexity in key management and distribution. They also use proxy re-encryption to offload cryptographic operations to cloud servers, and lazy re-encryption to reduce cryptographic cost on servers. In 2011, Zarandioon *et al.* [14] proposed K2C, a scalable ABE-based access hierarchy that can couple access control with the folder structure of file systems. By combining KP-ABC and a key-updating scheme, K2C users can use new keys to decrypt data encrypted with old keys and so reduce re-encrypting cost on servers when access hierarchies update keys. The combination of KP-ABE and a signature scheme allows users to prove that others own keys that satisfy the policy in order to provide signing and verification in K2C.

The researchers first focused on secure and authentic distributed file systems for outsourcing storage. Next, they proposed key management systems to deploy in public clouds, while data is encrypted and signed with keys distributed in the systems, then stored in cloud storage. All works achieve their goals with the assumption that users have public-key certificates. In contrast, we extend the OAuth standard by using ABE-tokens to establish an abstract layer that separates authentication from authorization for no restriction to any user credentials and delegates both authentication and authorization from owners to consumers for end-to-end approaches. AAuth puts no key management systems in CSPs, rather it constructs ABE-tokens from the cooperation of existing cloud entities and owners for a user-centric approach and distribution of key knowledge to minimize risks in semi-trusted cloud environments. Thus, our scheme is more abstract and generic than others.

8. CONCLUSIONS

This paper proposes a new authorization scheme that combats untrusted cloud servers by adopting CA-ABE, ElGamal-like masks, proxy re-encryption, and lazy re-encryption to achieve user-centric and end-to-end security. The main benefit is that it allows users to securely share resources across providers in semi-trusted cloud environments. To achieve this end, our scheme provides 1) an ABE-token for each authorization grant, 2) a user-centric system in which an owner controls the authorization system to protect her resources, 3) an end-to-end cryptographic function and an authorization from an owner to a consumer, 4) a light-weight encryption for time slot synchronization. Performance evaluation results show that our scheme has no significant computation cost for users, and AAuth's cost is independent of the system's number of users. Simulation results show an acceptable cost increase compensated for by better security than the current OAuth. Security analysis shows that our modified CP-ABE is as secure as the original scheme, and our protocols can resist both internal and external adversaries.

9. REFERENCES

[1] J. Bethencourt, A. Sahai and B. Waters, Ciphertext-Policy Attribute-Based Encryption, In *IEEE Symposium on Security and Privacy*, pp. 321-334, 2007.

[2] M. Blaze, G. Bleumer, and M. Strauss, Divertible protocol and atomic proxy cryptography, In *Proceedings of EUROCRYPT'98*, vol. 1402, pp. 127-144, 1998.

[3] K. D. Bowers, A. Juels and A. Opera, Proof of Retrievability: Theory and Implementation, In *ACM CCSW 2009*, 2009.

[4] K. D. Bowers, A. Juels and A. Opera, HAIL: A High-Availability and Integrity Layer for cloud Storage, In *ACM CCS 2009*, 2009.

[5] E. Hammer-Lahav, The OAuth 1.0 Protocol (RFC 5849), *Internet Engineering Task Force (IETF)*, April 2010.

[6] E. Hammer-Lahav, D. Recordon, D. Hardt, The OAuth 2.0 Authorization Protocol (Draft-ietf-oauth-v2-15), *Internet Engineering Task Force (IETF)*, April 2011.

[7] M. Kallahalla, E. Riedel, R. Swaminathan, Q. Wang, and K. Fu, Plutus-scalable secure file sharing on untrusted storage, In *Proceedings of Second USENIX Conference on File and Storage Technologies*, March 2003.

[8] C. Neuman, S. Hartman and K. Raeburn, The Kerberos Network Authentication Service (V5) (RFC 4120), *Internet Engineering Task Force (IETF)*, July 2005.

[9] specs@openid.net, The OpenID Authentication 2.0-Final *http://openid.net/specs/openid-authentication-2_0.html*, December 2007.

[10] B. Waters, Ciphertext-Policy Attribute-Based Encryption: An Expressive, Efficient, and Provably Secure Realization, In *Public Key Cryptography—PKC*, vol 6571 of LNCS, pp. 53-70, Springer, 2011

[11] W. Wang, Z. Li, R. Ownes and B. Bhargava, Secure and Efficient Access to Outsource Data, In *ACM CCSW 2009*, 2009.

[12] S. Yu, C. Wang, K. Ren, and W. Lou, Acieveing Secure, Scalable, and Fine-grained Data Access Control in Cloud Computing, In *Proceeceding of IEEE INFOCOM'10*, pp. 534-542, 2010.

[13] A. Yun, C. Shi and Y. Kim, On Protecting Integrity and Confidentiality of Cryptographic File System for Outsource Storage, In *ACM CCSW 2009*, 2009.

[14] S. Zarandioon, D. Yao, and V. Ganapathy, K2C: Crytographic Cloud Storage with Lazy Revocation and Anonymous Access, In *International ICST Conference on Security and Privacy in Communication Networks (Securecomm'11)*, 2011.