Enabling Efficient Access Control with Dynamic Policy Updating for Big Data in the Cloud

Kan Yang*, Xiaohua Jia*, Kui Ren†, Ruitao Xie* and Liusheng Huang‡
*Dept. of CS, City University of Hong Kong
†Dept. of CSE, University at Buffalo, SUNY
‡Dept. of CS, University of Sci. and Tech. of China
Email: kan.yang@my.cityu.edu.hk

***Abstract*—Due to the high volume and velocity of big data, it is an effective option to store big data in the cloud, because the cloud has capabilities of storing big data and processing high volume of user access requests. Attribute-Based Encryption (ABE) is a promising technique to ensure the end-to-end security of big data in the cloud. However, the policy updating has always been a challenging issue when ABE is used to construct access control schemes. A trivial implementation is to let data owners retrieve the data and re-encrypt it under the new access policy, and then send it back to the cloud. This method incurs a high communication overhead and heavy computation burden on data owners. In this paper, we propose a novel scheme that enabling efficient access control with dynamic policy updating for big data in the cloud. We focus on developing an outsourced policy updating method for ABE systems. Our method can avoid the transmission of encrypted data and minimize the computation work of data owners, by making use of the previously encrypted data with old access policies. Moreover, we also design policy updating algorithms for different types of access policies. The analysis show that our scheme is correct, complete, secure and efficient.**

***Index Terms*—Access Control, Policy Updating, ABE, Big Data, Cloud**

I. Introduction

Big data refers to high volume, high velocity, and/or high variety information assets that require new forms of processing to enable enhanced decision making, insight discovery and process optimization. Due to its high volume and complexity, it becomes difficult to process big data using on-hand database management tools. An effective option is to store big data in the cloud, as the cloud has capabilities of storing big data and processing high volume of user access requests in an efficient way. When hosting big data into the cloud, the data security becomes a major concern as the cloud servers cannot be fully trusted by data owners.

Attribute-Based Encryption (ABE) [1]–[5] has emerged as a promising technique to ensure the end-to-end data security in cloud storage system. It allows data owners to define the access policy and encrypt the data under the policy, such that only users whose attributes satisfying the access policies can decrypt the data. When more and more organizations and enterprises outsource the data into the cloud, the policy updating becomes a significant issue as the data access policies may be dynamically and frequently changed by data owners. However, this policy updating issue has not been considered in existing attribute-based access control schemes [6]–[8].

The policy updating is a difficult issue in attribute-based access control systems, because once the data owner outsourced the data into the cloud, it would not keep a copy in local systems. When the data owner wants to change the access policy, it has to transfer the data back to the local site from the cloud, re-encrypt the data under the new access policy, and then move it back to the cloud server. By doing so, it incurs a high communication overhead and heavy computation burden on data owners. This motivates us to develop a new method to outsource the policy updating to the cloud server.

The grand challenge of outsourcing the policy updating to the cloud is to guarantee the following requirements:

1) *Correctness*: Users who possess sufficient attributes should still be able to decrypt the data encrypted under new access policy by running the original decryption algorithm.
2) *Completeness*: The policy updating method should be able to update any type of access policy.
3) *Security*: The policy updating should not break the security of the access control system or introduce any new security problems.

The policy updating problem has been discussed in key-policy structure [1] and ciphertext-policy structure [9]. However, these methods cannot satisfy the completeness requirement, because they can only delegate key/ciphertext with a new access policy that should be more restrictive than the previous policy. Furthermore, they cannot satisfy the security requirement either. For example, when a new attribute is added into a threshold gate and the threshold gate is changed from (t,n) to a $(t+1,n+1)$, both methods will set the share of the new attribute to be 0. In this case, users who only holds t attributes (excluding the new attribute) can satisfy the new $(t+1,n+1)$-gate.

In this paper, we focus on solving the policy updating problem in ABE systems, and propose an efficient outsourced policy updating method. Instead of retrieving and re-encrypting the data, data owners only send a policy updating query to the cloud server, and the cloud server can update the policy of the encrypted data without decrypting it. Our scheme can not only satisfy all the above requirements, but also avoid the

*This work was supported by RGC HK [Project No. CityU 114112], NSF China [Grant No. U1301256] and NSF US [Grant No. CNS-1262277].

978-14799-3360-0/14/$31.00 ©2014 IEEE

transfer of encrypted data back and forth and minimize the computation work of the data owner by making full use of the previously encrypted data with the old access policies in the cloud.

The contributions of this paper include:

1) We formulate the policy updating problem in ABE sytems and develop a new method to outsource the policy updating to the server.
2) We propose an expressive and efficient data access control scheme for big data, which enables efficient dynamic policy updating.
3) We design policy updating algorithms for different types of access policies, e.g., Boolean Formulas, LSSS Structure and Access Tree.

The remaining of this paper is organized as follows. In Section II, we defines the system model, the framework and the security model. Section III describes an attribute-based access control scheme for big data based on an adapted mutli-authority CP-ABE method in [5]. Section IV proposes some policy updating algorithms for different types of access policies. In Section V, we give a comprehensive analysis of our scheme in terms of correctness, completeness, security and performance. The related work is given in Section VI. Finally, this paper is summarized in Section VII. In Appendix A, we give the definition of access structures in ABE systems, and then introduce two types of access structures that are well used in constructing ABE schemes.

II. System and Security Model

A. System Model

We consider a cloud storage system with multiple authorities, as shown in Fig.1. The system model consists of the following entities: authorities (*AA*), cloud server (server), data owners (owners) and data consumers (users).

Authority. Every authority is independent with each other and is responsible for managing attributes of users in its domain. It also generates a secret/public key pair for each attributes in its domain, and generates a secret key for each user according to their attributes.

Server. The cloud server stores the data of data owners and provides data access service to users. The server is also responsible for updating ciphertexts from old access policies to new access policies.

Owner. The data owners define the access policies and encrypt the data under the policies before hosting them in the cloud. They also ask the server to update the access policies of the encrypted data stored in the cloud.

User. Each user is assigned with a global user identity and can freely get the ciphertexts from the server. The user can decrypt the ciphertext, only when the its attributes satisfy the access policy defined in the ciphertext.

B. Framework

To meet all the requirements of policy updating, we define the framework of our access control scheme as follows.

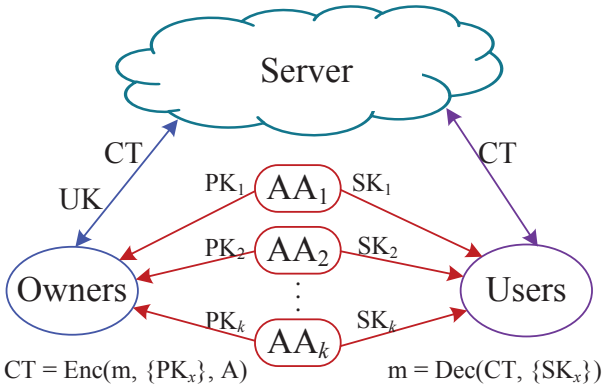

Fig. 1. System Model

Definition 1 (Framework)**.** *Our dynamic policy access control scheme is a collection of the following algorithms:* GlobalSetup, AuthoritySetup, SKeyGen, Encrypt, Decrypt, UKeyGen *and* CTUpdate.

- **GlobalSetup**$(\lambda) \rightarrow \mathsf{GP}$. The global setup algorithm takes no input other than the implicit security parameter λ. It outputs global parameters GP for the system.
- **AuthoritySetup**$(\mathsf{GP}, AID) \rightarrow (\mathsf{SK}, \mathsf{PK})$. The authority setup algorithm is run by each authority *AID* with GP and the authority identity *AID* as inputs and produce its secret/public key pair $(\mathsf{SK}_{AID}, \mathsf{PK}_{AID})$.
- **SKeyGen**$(GID, \mathsf{GP}, S_{GID,AID}, \mathsf{SK}_{AID}) \rightarrow \mathsf{SK}_{GID,AID}$. Each authority *AID* runs the secret key generation algorithm to generate a secret key $\mathsf{SK}_{GID,AID}$ for user *GID*. It takes as inputs the global identity *GID*, the global parameters, a set of attributes $S_{GID,AID}$ that issued by this authority *AID* and the secret key SK_{AID} of this authority. It outputs a secret key $\mathsf{SK}_{GID,AID}$ for this user *GID*.
- **Encrypt**$(\{\mathsf{PK}\}, \mathsf{GP}, m, \mathbb{A}) \rightarrow \mathsf{CT}$. The encryption algorithm takes as inputs a set of public keys $\{\mathsf{PK}\}$ for relevant authorities, the global parameters, the message m and an access policy $\mathbb{A}$. It outputs a ciphertext CT.
- **Decrypt**$(\mathsf{CT}, \mathsf{GP}, \{\mathsf{SK}_{GID,AID}\}) \rightarrow m$. The decryption algorithm takes as inputs the ciphertext, the global parameters and a collection of secret keys from relevant authorities for user *GID*. It outputs the message m when the user's attributes satisfy the access policy corresponding to the ciphertext. Otherwise, decryption fails.
- **UKeyGen**$(\{\mathsf{PK}\}, \mathsf{EnInfo}(m), \mathbb{A}, \mathbb{A}') \rightarrow \mathsf{UK}_m$. The update key generation algorithm is run by the data owner. It takes as inputs the relevant public keys, the encryption information $\mathsf{EnInfo}(m)$ of the message m, the previous access policy $\mathbb{A}$ and the new access policy $\mathbb{A}'$. It outputs the update key UK_m of m that is used to update the ciphertext CT from the previous access policy to the new one.
- **CTUpdate**$(\mathsf{CT}, \mathsf{UK}_m) \rightarrow \mathsf{CT}'$. The ciphertext updating algorithm is run by the cloud server. It takes as inputs the previous ciphertext CT and the update key UK_m. It

outputs a new ciphertext CT' corresponding to the new access policy $\mathbb{A}'$.

C. Security Model

The cloud server is assumed to be semi-trusted (curious-but-honest), it is curious about the data it stored and the messages it received, but it will update the ciphertext correctly for data owners. We also assume that the server may send the owners' data to the users who do not have access permission. The data owners are assumed to be fully trusted. The users are assumed to be dishonest, i.e., they may collude to access unauthorized data. The authorities can be corrupted or compromised by the attackers. We assume that the adversaries can corrupt authorities only statically, but key queries can be made adaptively.

We now describe the security model for our system by the following game between a challenger and an adversary:

Setup. The global setup algorithm is run. The adversary specifies a set $S'_A \subset S_A$ of corrupted authorities. The challenger generates the pairs of public key and the secret key by running the authority setup algorithm. For uncorrupted authorities in $S_A - S'_A$, the challenger sends only the public keys to the adversary. For corrupted authorities in S'_A, the challenger sends both the public keys and secret keys to the adversary.

Phase 1. The adversary makes secret key queries by submitting pairs $(GID, S_{GID,AID})$ to the challenger, where GID is an identity and $S_{GID,AID}$ is a set of attributes belonging to an uncorrupted authority AID. The challenger gives the corresponding secret keys $\mathsf{SK}_{GID,AID}$ to the adversary.

Challenge. The adversary submits two equal length messages m_0 and m_1. In addition, the adversary gives a set of challenge access structure $\{(M_1^*, \rho_1^*), \cdots, (M_q^*, \rho_q^*)\}$ which must satisfy the constraint that the adversary cannot ask for a set of keys that allow decryption, in combination with any keys that can obtained from corrupted authorities. The challenger then flips a random coin b, and encrypts m_b under all access structures $\{(M_1^*, \rho_1^*), \cdots, (M_q^*, \rho_q^*)\}$. Then, the ciphertext $\{\mathsf{CT}_1^*, \cdots, \mathsf{CT}_q^*\}$ are given to the adversary.

Phase 2. The adversary may query more secret keys, as long as they do not violate the constraints on the challenge access structures. The adversary can also makes update key queries by submitting the pair $(M_i^*, \rho_i^*), (M_j^*, \rho_j^*)$, the simulator returns the update key UK_{m_b} to the adversary.

Guess. The adversary outputs a guess b' of b.

The advantage of an adversary $\mathcal{A}$ in this game is defined as

$$Pr[b' = b] - \frac{1}{2}.$$

Definition 2. *Our scheme is secure against static corruption of authorities if all polynomial time adversaries have at most a negligible advantage in the above security game.*

III. Attribute-based Access Control with Dynamic Policy Updating for Big Data

We construct our dynamic-policy access control scheme based on an adapted CP-ABE method in [5]. Our scheme consists of five phases: System Initialization, Key Generation, Data Encryption, Data Decryption and Policy Updating.

A. System Initialization

The system initialization includes the global setup and the authority setup.

1) Global Setup: In the global setup, two multiplicative groups $\mathbb{G}$ and $\mathbb{G}_T$ are chosen with the same prime order p and the bilinear map $e : \mathbb{G} \times \mathbb{G} \to \mathbb{G}_T$ between them. A random oracle H maps global identities GID to elements of $\mathbb{G}$. Let g be a generator of $\mathbb{G}$, the global parameters is set to be

$$\mathsf{GP} = (\ p,\ g,\ H\ )$$

2) Authority Setup: Each authority AID runs the authority setup algorithm to generate its secret/public key pair. Let S_{AID} denote the set of all the attributes managed by the authority AID. For each attribute $x \in S_{AID}$, the authority chooses two random exponents $\alpha_x, \beta_x \in \mathbb{Z}_p$ and publishes its public key as

$$PK_{AID} = \{\ e(g,g)^{\alpha_x},\ g^{\beta_x}\ \}_{\forall x \in S_{AID}}.$$

It keeps $SK_{AID} = \{\alpha_x, \beta_x\}_{\forall x \in S_{AID}}$.

B. Key Generation

To generate the secret key for user GID, each authority AID will first assign a set of attributes $S_{GID,AID}$ to this user. Then, it runs the secret key generation algorithm to generate a set of secret keys as

$$SK_{GID,AID} = \{K_{x,GID} = g^{\alpha_x} H(GID)^{\beta_x}\}_{\forall x \in S_{GID,AID}}.$$

C. Data Encryption

The owner first encrypts the data m by running the encryption algorithm Encrypt. The algorithm takes as inputs a set of public keys $\{\mathsf{PK}\}$ for relevant authorities, the global parameters, the data m and an $n \times l$ access matrix M with ρ mapping its rows to attributes. It chooses a random encryption exponent $s \in \mathbb{Z}_p$ and a random vector $\vec{v} = (s, y_2, \cdots, y_l) \in \mathbb{Z}_p^l$, where $y_2, \cdots, y_l$ are used to share the encryption exponent s. For $i = 1$ to n, it computes $\lambda_i = M_i \cdot \vec{v}$, where M_i is the vector corresponding to the i-th row of M. It also chooses a random vector $\vec{w} \in \mathbb{Z}_p^l$ with 0 as its first entry and computes $w_i = M_i \cdot \vec{w}$. For each row i of M, it chooses a random $r_i \in \mathbb{Z}_p$ and computes the ciphertext as

$$\begin{aligned}\mathsf{CT} = (\ & C = m \cdot e(g,g)^s, \\ & \forall i = 1\ to\ n:\ C_{1,i} = e(g,g)^{\lambda_i} e(g,g)^{\alpha_{\rho(i)} r_i}, \\ & \qquad C_{2,i} = g^{r_i},\ C_{3,i} = g^{\beta_{\rho(i)} r_i} g^{w_i}\ ).\end{aligned}$$

Then, the encryption information $\mathsf{EnInfo}(m)$ of the data m contains all the random numbers r_i, i.e., $\mathsf{EnInfo}(m) = \{r_1, \cdots, r_n\}$.

D. Data Decryption

To decrypt a ciphertext, the user first obtains $H(GID)$ from the random oracle. If the user has the secret keys $\{K_{\rho(i),GID}\}$ for a subset of rows i of M such that $(1, 0, \ldots, 0)$ is in the span of these rows, then the user proceeds as follows. For each such i, the user computes

$$\frac{C_{1,i} \cdot e(H(GID), C_{3,i})}{e(K_{\rho(i),GID}, C_{2,i})} = e(g,g)^{\lambda_i} e(H(GID), g)^{w_i}.$$

The user then chooses constants $c_i \in \mathbb{Z}_p$ such that $\sum_i c_i M_i = (1,0,\ldots,0)$ and computes

$$\prod_i \left(e(g,g)^{\lambda_i} e(H(GID),g)^{w_i} \right)^{c_i} = e(g,g)^s.$$

We recall that $\lambda_i = M_i \cdot \vec{v}$ and $w_i = M_i \cdot \vec{w}$, where $\vec{v} \cdot (1,0,\ldots,0) = s$ and $\vec{w} \cdot (1,0,\ldots,0) = 0$. The data can then be decrypted as

$$m = C / e(g,g)^s.$$

E. Policy Updating

To update the access policy of the encrypted data in the cloud, we delegate the ciphertext update from the data owner to the cloud server, such that the heavy communication overhead of the data retrieval can be eliminated and the computation cost on data owners can also be reduced.

When the data owner wants to update the ciphertext from the previous access policy $\mathbb{A}$ to the new access policy $\mathbb{A}'$, it first generates an update key UK_m by running the update key generation algorithm UKGen and then owner sends the update key UK_m to the cloud server. Upon receiving the update key from the data owner, the cloud server will run the ciphertext updating algorithm $\mathsf{CTUpdate}$ to update the ciphertext from the previous access policy $\mathbb{A}$ to the new one $\mathbb{A}'$.

However, the update key generation algorithm UKGen and the ciphertext updating algorithm $\mathsf{CTUpdate}$ are related to the relationship between the previous access policy $\mathbb{A}$ and the new access policy $\mathbb{A}'$. For different types of updating operation, we have different design of UKGen and $\mathsf{CTUpdate}$, which will be described in detail in the next section.

IV. Dynamic Policy Updating

In this section, we first design the policy updating algorithms for monotonic boolean formulas. Then, we present the algorithms to update LSSS structures. Finally, we consider the general threshold access tree structures by designing the algorithms of updating a threshold gate.

A. Updating a Boolean Formula

Access policies with monotonic boolean formulas can be represented as the simplest threshold access trees, where the non-leaf nodes are AND and OR gates, and the leaf nodes correspond to attributes. The monotonic boolean formulas can be easily converted to LSSS structure, as the number of leaf nodes in the access tree is the same as the number of rows in the corresponding LSSS matrix. As shown in Fig. 2, there are four basic operations: *Attr2OR*, *Attr2AND*, *AttrRmOR* and *AttrRmAND*.

*1) Converting an attribute to an OR gate (***Attr2OR***):* This *Attr2OR* operation involves converting an existing attribute $x_j (j \in [1,n])$ to an OR gate $(x_j \vee x_{n+1})$ by adding a new attribute x_{n+1}. In this case, the new attribute x_{n+1} plays the same role as the attribute x_j in the new access policy. Therefore, we can easily construct the ciphertext component $C_{n+1} = (C_{1,n+1}, C_{2,n+1}, C_{3,n+1})$ for the new attribute x_{n+1} from the component C_j corresponding to the existing attribute x_j.

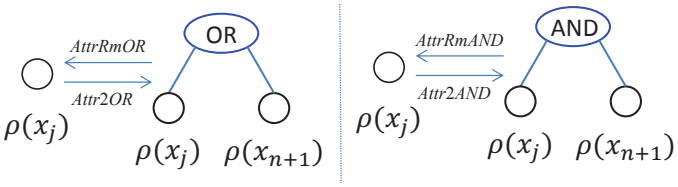

Fig. 2. Operations of Boolean Formula

To achieve this *Attr2OR* operation on data m, the update key generation algorithm UKGen takes the encryption information $\mathsf{EnInfo}(m)$ of the data m and the public keys. It chooses random $a_m, r_{n+1} \in \mathbb{Z}_p$ and generates the update key as

$$\mathsf{UK}_m = (\ a_m,\ \mathsf{UK}_{1,m} = \frac{e(g,g)^{\alpha_{x_{n+1}} r_{n+1}}}{e(g,g)^{\alpha_{x_j} r_j a_m}},$$
$$\mathsf{UK}_{2,m} = g^{r_{n+1} - r_j},\ \mathsf{UK}_{3,m} = \frac{g^{\beta_{x_{n+1}} r_{n+1}}}{g^{\beta_{x_j} r_j a_m}}\ )$$

Then, the data owner will send the tuple $(Attr2OR,\ \mathsf{UK}_m)$ to the server and ask the server to update the ciphertext CT corresponding to m. The ciphertext updating algorithm $\mathsf{CTUpdate}$ constructs the new ciphertext component $C_{x_{n+1}}$ as follows.

$$C_{1,n+1} = (C_{1,j})^{a_m} \cdot \mathsf{UK}_{1,m} = e(g,g)^{\lambda_{n+1}} \cdot e(g,g)^{\alpha_{x_{n+1}} r_{n+1}};$$
$$C_{2,n+1} = C_{2,j} \cdot \mathsf{UK}_{2,m} = g^{r_{n+1}};$$
$$C_{3,n+1} = (C_{3,j})^{a_m} \cdot \mathsf{UK}_{3,m} = g^{\beta_{x_{n+1}} r_{n+1}} \cdot g^{w_{n+1}},$$

where $\lambda_{n+1} = a_m \cdot \lambda_j$ and $w_{n+1} = a_m \cdot w_j$.

*2) Converting an attribute to an AND gate (***Attr2AND***):* This *Attr2AND* operation involves converting an existing attribute $x_j (j \in [1,n])$ to an AND gate $(x_j \wedge x_{n+1})$ by adding a new attribute x_{n+1}. In this case, the combination of the new attribute x_{n+1} and the attribute x_j in the new access policy plays the same role as the attribute x_j in the previous policy. Therefore, we can modify the previous ciphertext component C_j corresponding to x_j into a new version C'_j, and construct the new ciphertext component $C_{n+1} = (C_{1,n+1}, C_{2,n+1}, C_{3,n+1})$ for the new attribute x_{n+1}.

To achieve this *Attr2AND* operation on data m, the update key generation algorithm UKGen takes the encryption information $\mathsf{EnInfo}(m)$ of the data m and the public keys as inputs. It chooses random $a_m, \lambda', w', r_{n+1} \in \mathbb{Z}_p$, such that $\lambda'_j = \lambda_j + \lambda'$ and $\lambda_{n+1} = a_m \cdot \lambda'$, as well as $w'_j = w_j + w'$ and $w_{n+1} = a_m \cdot w'$. Then, the update key can be generated as

$$\mathsf{UK}_m = (\ \mathsf{UK}_{1,m} = e(g,g)^{\lambda'},\ \mathsf{UK}_{2,m} = g^{w'},\ C_{n+1}\ ),$$

where the new ciphertext component C_{n+1} is constructed as

$$C_{n+1} = (\ C_{1,n+1} = e(g,g)^{\lambda_{n+1}} \cdot e(g,g)^{\alpha_{x_{n+1}} r_{n+1}},$$
$$C_{2,n+1} = g^{r_{n+1}},\ C_{3,n+1} = g^{\beta_{x_{n+1}} r_{n+1}} \cdot g^{w_{n+1}}\ )$$

Then, the data owner will send the tuple $(Attr2AND,\ \mathsf{UK}_m)$ to the server and ask the server to update the ciphertext CT corresponding to m. The server first adds the new ciphertext

component C_{n+1} to the ciphertext CT, and then runs the ciphertext updating algorithm CTUpdate to update the previous ciphertext component C_j to the new version C'_j as

$$C'_j = (\ C'_{1,j} = C_{1,j} \cdot \mathsf{UK}_{1,m} = e(g,g)^{\lambda'_j} \cdot e(g,g)^{\alpha_{x_j} r_j},$$
$$C'_{2,j} = C_{2,j},\ C'_{3,j} = C_{3,j} \cdot \mathsf{UK}_{2,m} = g^{\beta_{x_j} r_{n+1}} \cdot g^{w'_j}\ ).$$

*3) Removing an attribute from an OR gate (***AttrRmOR***):* To remove an attribute x_j from an OR gate, the data owner can simply send a tuple $(AttrRM,\ m,\ j)$, where $\rho(j) = x_j$ to ask the server to delete the corresponding ciphertext component C_j in the ciphertext.

*4) Removing an attribute from an AND gate (***AttrRmAND***):* To remove an attribute from an AND gate, all the shares should be re-randomized, such that the correctness requirement can be satisfied. This can be easily achieved by using the method of converting a (t,t)-gate to a $(t-1,t)$-gate which will be described later.

B. Updating a LSSS Structure

Access policies can also be expressed in LSSS structure as in our access control scheme. To convert a LSSS structure (M,ρ) to a new LSSS structure (M',ρ'), it is too costly to choose a new encryption secret s' and re-encrypt the data under the new access policy. In order to save the communication cost and the computation cost on data owners, in our method, we do not change the encryption secret s, such that we can make full use of the previous ciphertext encrypted under the old policy (M,ρ).

To enable the data owner to re-randomize the encryption secret s, the encryption information $\mathsf{EnInfo}(m)$ of the data m should also contain two random vectors $\vec{v}$ and $\vec{w}$, and the public key of each attribute x is known to the data owner as $(g^{\alpha_x}, g^{\beta_x})$. The data owner will run the update key generation algorithm to construct the update keys and send them to the cloud server. Upon receiving the update keys, the cloud server will run the ciphertext update algorithm to update ciphertext from the previous access policy to the new policy. The update key algorithm and the ciphertext update algorithm are designed as follows.

1) Update Key Generation: The update key generation algorithm UKGen takes as inputs the public keys, the encryption information of data m, and the previous access policy (M,ρ) and the new one (M',ρ'). Suppose the new access policy is described as an $n' \times l'$ access matrix M' with ρ' mapping its rows to attributes. Since the mapping functions ρ and ρ' are non-injective, we let $num_{\rho(i),M}$ and $num_{\rho(i),M'}$ denote the number of attribute $\rho(i)$ in M and M' respectively.

It first calls the policy comparing algorithm PolicyCompare to compare the new access policy (M',ρ') with the previous one (M,ρ), and outputs three sets of row indexes $I_{1,M'}, I_{2,M'}, I_{3,M'}$ of M'. Both $I_{1,M'}$ and $I_{2,M'}$ denote the set of indexes j such that $\rho'(j)$ exists in M. But the total number of index j in $I_{1,M'}$ will not exceed the total number of attribute $\rho(i)(\rho(i) = \rho'(j))$ in M. If $num_{\rho'(j),M'} \geq num_{\rho'(j),M}$, those exceeding $num_{\rho'(j),M'} - num_{\rho'(j),M}$ indexes j will be put in $I_{2,M'}$. $I_{3,M'}$ denotes the set of indexes j such that $\rho'(j)$ does not exist in M, i.e., $\rho'(j)$ is a new attribute.

Algorithm 1 PolicyCompare

Input: new policy (M',ρ') with $l' \times k'$ matrix
Input: previous policy (M,ρ) with $l \times k$ matrix
Output: $I_{1,M'}, I_{2,M'}, I_{3,M'}$ ▷ three subsets of row indexes in M'

```
I_M ← index set of rows in M
for j = 1 to l' do
    if ρ'(j) in M then
        if I_M != ∅ & ∃i ∈ I_M s.t. ρ(i) == ρ'(j) then
            add (j,i) into I_{1,M'}
            delete i from I_M
        else
            find any i ∈ [1,l] s.t. ρ(i) == ρ'(j)
            add (j,i) into I_{2,M'}
        end if
    else
        add (j,0) into I_{3,M'}
    end if
end for
```

The algorithm first constructs two new random vectors: $\vec{v'} \in \mathbb{Z}_p^{l'}$ with s as its first entry and $\vec{w'} \in \mathbb{Z}_p^{l'}$ with 0 as its first entry. It then computes $\lambda_i = M_i \cdot \vec{v}$ and $w_i = M_i \cdot \vec{w}$, where M_i is the vector corresponding to the i-th row of M. It also computes $\lambda'_i = M'_i \cdot \vec{v'}$, and $w'_j = M'_j \cdot \vec{w'}$, where M'_j is the vector corresponding to the j-th row of M'. Let $I_M = \{1, \cdots, l\}$ be the index set of the rows of M.

For each $j \in [1,l']$, if $(j,i) \in I_{1,M'}(\mathsf{Type1})$, the algorithm generates the update key component as

$$\mathsf{UK}_{j,i,m} = (\ \mathsf{UK}^{(1)}_{j,i,m} = g^{\lambda'_j - \lambda_i},\ \mathsf{UK}^{(2)}_{j,i,m} = g^{w'_j - w_i}\ )$$

and set $r'_j = r_i$.

If $(j,i) \in I_{2,M'}(\mathsf{Type2})$, the algorithm chooses random numbers $r'_j, a_j \in \mathbb{Z}_p$ and generates the update key component as

$$\mathsf{UK}_{j,i,m} = (\ a_j,\ \mathsf{UK}^{(1)}_{j,i,m} = g^{\lambda'_j - a_j \lambda_i},\ \mathsf{UK}^{(2)}_{j,i,m} = g^{w'_j - a_j w_i}\ ).$$

If $(j,i) \in I_{3,M'}(\mathsf{Type3})$, the algorithm chooses a random $r'_j \in \mathbb{Z}_p$ and generates the update key component as

$$\mathsf{UK}_{j,i,m} = (\ \mathsf{UK}^{(1)}_{j,i,m} = g^{\lambda'_j} \cdot g^{\alpha_{\rho'(j)} r'_j},\ \mathsf{UK}^{(2)}_{j,i,m} = g^{r'_j},$$
$$\mathsf{UK}^{(3)}_{j,i,m} = g^{\beta_{\rho(i)} r'_j} g^{w'_j}\ ).$$

The update key UK_m is constructed as

$$\mathsf{UK}_m = (\ (\mathsf{Type1},\ \{\mathsf{UK}_{j,i,m}\}_{(j,i)\in I_{1,M'}}),$$
$$(\mathsf{Type2},\ \{\mathsf{UK}_{j,i,m}\}_{(j,i)\in I_{2,M'}}),$$
$$(\mathsf{Type3},\ \{\mathsf{UK}_{j,i,m}\}_{(j,i)\in I_{3,M'}})\ ).$$

Then, the data owner sends the update key UK_m to the cloud server.

2) Ciphertext Update: Upon receiving the update key UK_m, for each $j \in [1,l']$, the server runs the ciphertext updating al-

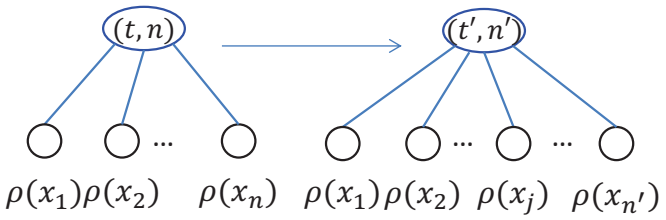

Fig. 3. Updating a (t,n)-gate to a (t',n')-gate

gorithm to compute each ciphertext component C'_j as follows.

If Type1($j \in I_{1,M'}$), the ciphertext component C'_j is computed as

$$C'_j = (\ C'_{1,j} = C_{1,i} \cdot e(g, \mathsf{UK}^{(1)}_{j,i,m}) = e(g,g)^{\lambda'_j} \cdot e(g,g)^{\alpha_{\rho'(j)} r'_j},$$
$$C'_{2,j} = C_{2,i},\ C'_{3,j} = C_{3,i} \cdot \mathsf{UK}^{(2)}_{j,i,m} = g^{\beta_{\rho'(j)} r'_j} \cdot g^{w'_j}\ ),$$

where $r'_j = r_i$.

If Type2($j \in I_{2,M'}$), the ciphertext component C'_j is computed as

$$C'_j = (\ C'_{1,j} = (C_{1,i})^{a_j} \cdot e(g, \mathsf{UK}^{(1)}_{j,i,m}) = e(g,g)^{\lambda'_j} \cdot e(g,g)^{\alpha_{\rho'(j)} r'_j},$$
$$C'_{2,j} = (C_{2,i})^{a_j} = g^{r'_j},$$
$$C'_{3,j} = (C_{3,i})^{a_j} \cdot \mathsf{UK}^{(2)}_{j,i,m} = g^{\beta_{\rho'(j)} r'_j} \cdot g^{w'_j}\ ),$$

where $r'_j = a_j r_i$.

If Type3($j \in I_{3,M'}$), the ciphertext component C'_j is computed as

$$C'_j = (\ C'_{1,j} = e(g, \mathsf{UK}^{(1)}_{j,i,m}) = e(g,g)^{\lambda'_j} \cdot e(g,g)^{\alpha_{\rho'(j)} r'_j},$$
$$C'_{2,j} = \mathsf{UK}^{(2)}_{j,i,m},\ C'_{3,j} = \mathsf{UK}^{(3)}_{j,i,m} = g^{\beta_{\rho'(j)} r'_j} \cdot g^{w'_j}\ ).$$

The new ciphertext CT' can be constructed as

$$\mathsf{CT}' = (m \cdot e(g,g)^s,\ C'_j\ \forall j \in [1,l']).$$

As we can see, in our method, all the pairing computations are moved to the cloud server, while the data owner only does the minimum computation.

C. Updating a Threshold Gate

The problem of updating a threshold gate from (t,n)-gate to (t',n')-gate has been discussed in key-policy structure [1] and ciphertext-policy structure [9]. However, the existing methods would introduce a security problem in the new threshold gate. For example, when increasing the threshold value from t to $t+1$, existing methods will set the $t+1$ share λ_{t+1} of the secret s to be 0, such that the secret s can be reconstructed by using $t+1$ shares as $\sum_{i=1}^{t+1} \lambda_i = \sum_{i=1}^{t} \lambda_i + \lambda_{t+1} = s + 0 = s$. In this case, any t shares are still be able to reconstruct the secret, which should not be allowed in a $(t+1,n)$-gate.

To solve the security problem, instead of setting the value of new share to be 0, our method is to re-randomize the secret s under the new policy (t',n')-gate, as shown in Fig. 3. The data owner first transforms the threshold gate into LSSS structure by running the policy converting algorithm Threshold2LSSS,

Algorithm 2 DNF2LSSS

Input: $\mathcal{A}$ ▷ an access structure $\mathcal{A}$
Input: s ▷ the secret to be shared
Output: $\mathcal{M}$ ▷ Monotone Span Program
1: let sss be DNF2SSS$(s,\mathcal{A})$;
2: let $\mathcal{M}$ be SSS2MSP(s,sss);
3: **return** s, M;

i.e., transforming (t,n)-gate and (t',n')-gate to (M,ρ) and (M',ρ') respectively. Then, we can apply the above algorithms to update the LSSS structure (M,ρ) to the new one (M',ρ').

To convert a threshold gate to LSSS structure, the algorithm Threshold2LSSS first converts the threshold gate into DNF boolean formulas, and then converts the DNF boolean formulas into LSSS structure by calling the algorithm DNF2LSSS. For example, a $(2,3)$-gate on attributes A, B, C can be simply represented as $(A \wedge B) \vee (B \wedge C) \vee (A \wedge C)$.

The algorithm DNF2LSSS used to change DNF boolean formulas to LSSS structures is a combination of two algorithms:
.

- DNF2SSS. The algorithm is adapted from [10] and used to construct a Secret Sharing Scheme from monotone DNF boolean formula.
- SSS2MSP: The algorithm is adapted from [11] and used to convert Secret Sharing Scheme into Monotone Span Program (LSSS Structure).

When converting (t,n)-gate to (M,ρ), we can derive the size of the access matrix M as $m \times l$, where $m = n \cdot C^{t-1}_{n-1}$ and $l = (t-1) \cdot C^{t}_{n} + 1$.

V. Analysis of Our Scheme

In this section, we give the analysis of our scheme to show that it can satisfy all the three requirements we defined. Then, we also give the performance analysis of our scheme.

Algorithm 3 DNF2SSS

Input: $\mathcal{A}_n$ ▷ a node n from an access structure $\mathcal{A}$
Input: s ▷ the secret to be shared
Output: $\{s_1, \cdots, s_l\}$ ▷ a set of shares
1: $I_M \leftarrow$ index set of rows in M;
2: **if** *Node Type* $= OR$ **then**
3: **for** each child c of $\mathcal{A}_n$ **do**
4: DNF2SSS(s, c); ▷ pass the s to child nodes
5: **end for**
6: **end if**
7: **if** *Node Type* $= AND$ **then**
8: let gate have t inputs;
9: select $r_1, \cdots, r_{t-1} \in_R \mathbb{Z}_p$;
10: **for all** r_i **do**
11: DNF2SSS$(r_i, i^{th}$ child of $\mathcal{A}_n)$;
12: **end for**
13: let $r_t = s - r_1 - \cdots - r_{t-1}\ (mod\ p)$;
14: DNF2SSS$(r_i, t^{th}$ child of $\mathcal{A}_n)$;
15: **end if**

Algorithm 4 SSS2MSP

Input: $\{\vec{V}_i\}$ ▷ Set of piece vectors for each attribute $Attr_i$
Input: s ▷ the secret to be shared
Output: $\mathcal{M}$ ▷ Monotone Span Program

let $\vec{Z}$ be a vector and set $\vec{Z}(0) = s$;
let M be a matrix;
let ρ be a labelling function;
for all $Attr_i$ **do**
  for each piece vector $\vec{V}_i$ for $Attr_i$; **do**
    apppend each random value in $\vec{V}_i$ to $\vec{Z}$;
    construct the position vector $\vec{v}_i$ for $Attr_i$;
    append $\vec{v}_i$ to M;
    let $\rho(M_{v_i})$ to $Attr_i$;
  end for
end for
pad M with the same row size;
return $(\vec{Z}, M, \rho)$;

A. Correctness Proof

Theorem 1. *Our access control scheme is still correct after the policy updating.*

Proof: We prove the correctness for each operation in our scheme as follows.

- *LSSS Structure/Threshold Gate Updating*: The secret s is re-randomized under the new access policy, such that the correctness is guaranteed by the secret sharing scheme.
- *Attr2OR*: Suppose c_j is the coefficient chosen for x_j during the decryption. We can simply get the coefficient $c_{n+1} = c_j/a_m$ for x_{n+1} when the new attribute x_{n+1} is chosen instead of x_j during the decryption.
- *Attr2AND*: Suppose c_j is the coefficient chosen for x_j during the decryption under the previous access policy. To decrypt the ciphertext under the new access policy, we still choose c_j as the coefficient for x_j and set the coefficient $c_{n+1} = -\frac{c_j}{a_m}$ for x_{n+1}.
- *AttrRmOR*: This operation will not affect the coefficient choices of the remaining attributes in the OR gate.
- *AttrRmAND*: The same as threshold gate updating.

■

B. Completeness Proof

Theorem 2. *Our scheme is complete for updating any types of access policies.*

Proof: The proposed policy updating method is designed based on access policies with LSSS structures, i.e., it can convert any (M, ρ) LSSS structure to a new one (M', ρ'). For any access tree policy, which is constructed with several threshold gates and a set of attributes, we have proposed an operation to update a threshold gate to any other threshold gate by converting the threshold gates into LSSS structure. Thus, our scheme can also update any access tree policy. Considering the boolean formula is a special access tree only with AND and OR gates, we propose a more efficient method to update the policy with boolean formulas. Any new boolean formulas can be derived from the previous boolean formulas by iteratively doing the operations *Attr2AND*, *Attr2OR*, *AttrRmAND* and *AttrRmOR*. Therefore, our proposed operations are complete for updating any access policies. ■

C. Security Proof

Our access control scheme is constructed on prime order groups, because the group operations on prime order groups are much faster than the ones on composite order groups. In this section, we will prove that our dynamic policy access control scheme is secure in the generic bilinear group model [2], [12], [13] and random oracle model [14]. However, our scheme can also be easily extended to be provable secure in the random oracle model by using groups with composite orders.

Theorem 3. *Our scheme is secure in the generic bilinear group model and random oracle model, if no polynomial time adversary can get non-negligible advantage in the security game defined in Section II-C.*

Proof: Our access control scheme is constructed based on the CP-ABE method with primer group order (CP-ABE-Primer) in [5], which is proved to be secure under generic bilinear group model and random oracle model. At an intuitive level, this means that if there are any vulnerabilities in the scheme, then these vulnerabilities must exploit specific mathematical properties of elliptic curve groups or cryptographic hash functions used when instantiating the scheme. Let $\mathcal{A}$ be an adversary who can break our scheme with non-negligible advantage. and we will construct an $\mathcal{A}'$ such that it can break the CP-ABE-Primer scheme in [5] with non-negligible advantage.

Different with the security game in [5], in our security game, the adversary returns a tuple of policies $\{(M_1^*, \rho_1^*), \cdots, (M_q^*, \rho_q^*)\}$ together with two messages (m_0, m_1), and the adversary receives an encryption of M_b under each of these policies. Moreover, our security game also allows the update key query for the challenging messages (m_0, m_1) between two access policies (M_i^*, ρ_i^*) and (M_j^*, ρ_j^*).

$\mathcal{A}'$ initializes the CP-ABE-Primer security game and forwards the public key PK to $\mathcal{A}$. To simulate the key generation oracle of $\mathcal{A}$, $\mathcal{A}'$ queries its key generation oracle for all $x \in S$ to respond to a $\mathsf{SK}(GID, S)$ query. The simulation of challenge ciphertext of $\mathcal{A}$ is the same with the one of $\mathcal{A}'$.

Now, we prove that the update key query in our security game will not increase the advantage of $\mathcal{A}'$. Considering two update key queries $\mathsf{UK}(m_0, (M_i^*, \rho_i^*), (M_j^*, \rho_j^*))$ and $\mathsf{UK}(m_1, (M_i^*, \rho_i^*), (M_j^*, \rho_j^*))$, the update key generation oracle returns the same update keys which do not involve with the challenge data, if we consider the encryption random numbers are the same in encrypting m_0 and m_1 (this is because only one challenge message is chosen by the simulator by tossing a coin in the security game). Therefore, the update key will not reveal any information on the chosen challenging message. This completes the proof. ■

TABLE I
SIZE OF UPDATE KEY

Operation	*Attr*2*OR*	*Attr*2*AND*	*Type*1	*Type*2	*Type*3
Size(UK)	4$\|p\|$	5$\|p\|$	2$\|p\|$	3$\|p\|$	3$\|p\|$

D. Performance Analysis

In our method, the data owner only needs to send the update keys to the cloud server, instead of the whole encrypted big data. Therefore, our method can significantly reduce the communication cost during the policy updating. Suppose $|p|$ is the element size in the $\mathbb{G}, \mathbb{G}_T, \mathbb{Z}_p$. Table I shows the size of update keys in our scheme. We can see that Type1 operation incurs the smallest size of update keys. When updating an access policy to a new one, the most common operations are Type1 operations, such that our scheme incurs a small communication cost.

Compared with Sahai, Seyalioglu and Waters's Scheme (SSW's scheme) [9], our scheme makes full use of the previous ciphertexts encrypted under the old access structure. That is if an attribute in the new access policy has ever appeared in the previous access policy, the new ciphertext component of this attribute can be derived from the previous ciphertext component with the update key. The data owner only needs to compute ciphertext components for new attributes. Moreover, in our scheme, we also delegate all the pairing operations to the server, such that the workload of the data owner can be further reduced.

We also simulate the operation time for each type of operation, including *Attr*2*OR*, *Attr*2*AND* for Boolean Formulas Updating and Type1, Type2, Type3 for LSSS Structure/Threshold Gates Updating, as described in Fig. 4. The simulation is run on a Linux system with an Intel Core 2 Duo CPU at 3.16GHz and 4.00GB RAM. The code uses the Pairing-Based Cryptography library version 0.5.12 to simulate the access control schemes. We use a symmetric elliptic curve α-curve, where the base field size is 512-bit and the embedding degree is 2. The α-curve has a 160-bit group order, which means p is a 160-bit length prime. All the simulation results are the mean of 20 trials. From Fig. 4, we can see that Type1 operations incurs less computation cost on data owners, as well as less total computation cost. We know that the Type1 operation is the most common operation when converting an access policy to a new one. Therefore, our scheme can minimize the workload of data owners, as well as the one of cloud servers.

VI. RELATED WORK

Recently, some attribute-based access control schemes [6]–[8] were proposed to ensure the data confidentiality in the cloud. It allows data owners to define an access structure on attributes and encrypt the data under this access structure, such that data owners can define the attributes that the user needs to possess in order to decrypt the ciphertext. However, the policy updating becomes a difficult issue when applying ABE methods to construct access control schemes, because once data owner outsource the data into cloud, they won't store in local systems. To change the access policies of encrypted data in the cloud, a trivial method is to let data owners retrieve the data and re-encrypt it under the new access policy, and then send it back to the cloud server. But this method will incur a high communication overhead and heavy computation burden on data owners.

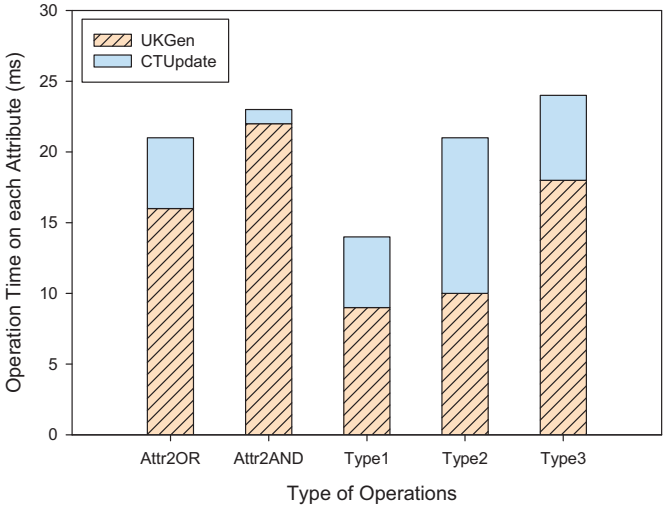

Fig. 4. Comparison of Computation Cost for Different Operations

In [1], the authors proposed a Key-Policy Attribute-Based Encryption method and discussed on how to change the policies on keys. In [9], the authors also proposed a ciphertext delegation method to update the policy of ciphertext. However, these methods cannot satisfy the completeness requirement, because they can only delegate key/ciphertext with a new access policy which is more restrictive than the previous policy. Furthermore, they cannot satisfy the security requirement either. For example, when a new attribute is added into a threshold gate and the threshold gate is changed from (t,n) to a $(t+1,n+1)$, both of their methods will set the share of the new attribute to be 0. In this case, users who only holds t attributes (excluding the new attribute) can satisfy new $(t+1,n+1)$-gate. Thus, a new outsourced policy updating method is desired for ABE systems.

VII. CONCLUSION

In this paper, we investigated the policy updating problem in ABE systems and formulated some challenging requirements of this problem. We developed an efficient method to outsource the policy updating to the cloud server, which can satisfy all the requirements. We also proposed an expressive attribute-based access control scheme for big data in the cloud, which enables efficient dynamic policy updating. Furthermore, we designed policy updating algorithms for different types of access policies. We also analyzed our scheme in terms of correctness, completeness, security and performance. Although the policy updating algorithms were designed based on Lewko and Waters' scheme, our ideas and methods of outsourced policy updating can also be applied to other ABE systems.

REFERENCES

[1] V. Goyal, O. Pandey, A. Sahai, and B. Waters, "Attribute-based encryption for fine-grained access control of encrypted data," in *CCS'06*. ACM, 2006, pp. 89–98.
[2] J. Bethencourt, A. Sahai, and B. Waters, "Ciphertext-policy attribute-based encryption," in *S&P'07*. IEEE Computer Society, 2007, pp. 321–334.
[3] B. Waters, "Ciphertext-policy attribute-based encryption: An expressive, efficient, and provably secure realization," in *PKC'11*. Springer, 2011, pp. 53–70.
[4] A. B. Lewko, T. Okamoto, A. Sahai, K. Takashima, and B. Waters, "Fully secure functional encryption: Attribute-based encryption and (hierarchical) inner product encryption," in *EUROCRYPT'10*. Springer, 2010, pp. 62–91.
[5] A. B. Lewko and B. Waters, "Decentralizing attribute-based encryption," in *EUROCRYPT'11*. Springer, 2011, pp. 568–588.
[6] S. Ruj, A. Nayak, and I. Stojmenovic, "DACC: Distributed Access Control in Clouds," in *TrustCom'11*. IEEE, 2011, pp. 91–98.
[7] J. Hur and D. K. Noh, "Attribute-based access control with efficient revocation in data outsourcing systems," *IEEE Trans. Parallel Distrib. Syst.*, vol. 22, no. 7, pp. 1214–1221, 2011.
[8] S. Yu, C. Wang, K. Ren, and W. Lou, "Achieving secure, scalable, and fine-grained data access control in cloud computing," in *INFOCOM'10*. IEEE, 2010, pp. 534–542.
[9] A. Sahai, H. Seyalioglu, and B. Waters, "Dynamic credentials and ciphertext delegation for attribute-based encryption," in *CRYPTO'12*. Springer, 2012, pp. 199–217.
[10] J. C. Benaloh and J. Leichter, "Generalized secret sharing and monotone functions," in *CRYPTO'88*, 1988, pp. 27–35.
[11] A. Beimel, "Secure schemes for secret sharing and key distribution," *DSc dissertation*, 1996.
[12] D. Boneh, X. Boyen, and E.-J. Goh, "Hierarchical identity based encryption with constant size ciphertext," in *EUROCRYPT'05*, 2005, pp. 440–456.
[13] V. Shoup, "Lower bounds for discrete logarithms and related problems," in *EUROCRYPT'97*, 1997, pp. 256–266.
[14] M. Bellare and P. Rogaway, "Random oracles are practical: A paradigm for designing efficient protocols," in *CCS'93*, 1993, pp. 62–73.

APPENDIX A
STRUCTURE OF ACCESS POLICY IN ABE SYSTEM

We first give the definition of access structures in ABE systems, then we introduce two types of access structures that are well used in constructing ABE schemes: LSSS Structure and Access Tree Structure.

Definition 3 (Access Structure)**.** *Let $\{P_1, P_2, \cdots, P_n\}$ be a set of parties. A collection $\mathbb{A} \subseteq 2^{\{P_1, P_2, \cdots, P_n\}}$ is monotone if $\forall B, C$ if $B \in \mathbb{A}$ and $B \subseteq C$ then $C \in \mathbb{A}$. An access structure (respectively, monotone access structure) is a collection (respectively, monotone collection) $\mathbb{A}$ of non-empty subsets of $\{P_1, P_2, \cdots, P_n\}$, i.e., $\mathbb{A} \subseteq 2^{\{P_1, P_2, \cdots, P_n\}} \setminus \{\emptyset\}$. The sets in $\mathbb{A}$ are called the authorized sets, and the sets not in $\mathbb{A}$ are called the unauthorized sets.*

In attribute-based encryption scheme, the role of the parties is taken by the attributes. Thus, the access structure $\mathbb{A}$ will contain the authorized sets of attributes. We restrict our attention to monotone access structures.

A. LSSS Structure

Definition 4 (Linear Secret-Sharing Schemes (LSSS))**.** *A secret-sharing scheme Π over a set of parties $\mathcal{P}$ is called linear (over $\mathbb{Z}_p$) if*

1) *The shares for each party form a vector over $\mathbb{Z}_p$.*
2) *There exists a matrix M called the share-generating matrix for Π. The matrix M has l rows and n columns. For all $i = 1, \cdots, l$, the i-th row of M is labeled by a party $\rho(i)$ (ρ is a function from $\{1, \cdots, l\}$ to $\mathcal{P}$). When we consider the column vector $v = (s, r_2, \cdots, r_n)$, where $s \in \mathbb{Z}_p$ is the secret to be shared and $r_2, \cdots, r_n \in \mathbb{Z}_p$ are randomly chosen, then Mv is the vector of l shares of the secret s according to Π. The share $(Mv)_i$ belongs to party $\rho(i)$.*

Algorithm 5 AccessTree

Input: T ▷ A threshold tree
Input: s ▷ The secret to be shared
Output: a set of shares $\{s_1, \cdots, s_l\}$

```
Let q_x be a polynomial for node x;
Set q_root(0) := s;
Set degree d_root := k_root − 1; ▷ k_root is the threshold value of the root
Let rest of points in q_root be randomly chosen;
for all x ∈ T do
    Set degree of q_x as d_x := k_x − 1;
    Set q_x(0) = q_parent(x)(index(x));
    Let rest of points in q_x be chosen randomly;
end for
```

Every linear secret sharing-scheme according to the above definition also enjoys the *linear reconstruction* property: Suppose that Π is a LSSS for the access structure $\mathbb{A}$. Let $S \in \mathbb{A}$ be any authorized set, and let $I \subset \{1, 2, \cdots, l\}$ be defined as $I = \{i : \rho(i) \in S\}$. Then, there exist constants $\{w \in \mathbb{Z}_p\}_{i \in I}$ such that, for any valid shares $\{\lambda_i\}$ of a secret s according to Π, we have $\sum_{i \in I} w_i \lambda_i = s$. These constants $\{w_i\}$ can be found in time polynomial in the size of the share-generating matrix M. We note that for unauthorized sets, no such constants $\{w_i\}$ exist.

B. Access Tree Structure

An access tree is defined as a tree $\mathcal{T}$, where every no-leaf node x represents a threshold gate (k_x, num_x) (num_x is the number of its children), and every leaf x is described by an attribute $att(x)$ and a threshold value k_x. The function $\mathsf{parent}(x)$ denotes the parent of the node x in the tree. T also defines an ordering number for each child node and the function $\mathsf{index}(x)$ returns such a number associated with the node x.

Policy Checking: To determine whether or not an access tree $\mathcal{T}$ is satisfied by a set of attributes S, the policy checking algorithm recursively computes as follows. If node x is a non-leaf node, it returns 1 iff at least k_x children return 1, otherwise returns 0; If x is a leaf node, then it returns 1 iff $att(x) \in S$, otherwise returns 0.

Tree Implementing: The secret s to be shared is first assigned to the root node of the tree by using a random polynomial q_{root} with $q_{root}(0) = s$. The rest of the nodes are used to distributed this *root* polynomial amongst the other nodes as smaller degree polynomials, as illustrated in Algorithm 5.