Hindawi Publishing Corporation
Mobile Information Systems
Volume 2016, Article ID 5232846, 9 pages
http://dx.doi.org/10.1155/2016/5232846

Research Article

F2AC: A Lightweight, Fine-Grained, and Flexible Access Control Scheme for File Storage in Mobile Cloud Computing

Wei Ren, Lingling Zeng, Ran Liu, and Chi Cheng

School of Computer Science, China University of Geosciences, Wuhan 430074, China

Correspondence should be addressed to Wei Ren; weirencs@cug.edu.cn

Received 6 September 2015; Revised 11 November 2015; Accepted 19 November 2015

Academic Editor: Jong-Hyouk Lee

Current file storage service models for cloud servers assume that users either belong to single layer with different privileges or cannot authorize privileges iteratively. Thus, the access control is not fine-grained and flexible. Besides, most access control methods at cloud servers mainly rely on computationally intensive cryptographic algorithms and, especially, may not be able to support highly dynamic ad hoc groups with addition and removal of group members. In this paper, we propose a scheme called F2AC, which is a lightweight, fine-grained, and flexible access control scheme for file storage in mobile cloud computing. F2AC can not only achieve iterative authorization, authentication with tailored policies, and access control for dynamically changing accessing groups, but also provide access privilege transition and revocation. A new access control model called directed tree with linked leaf model is proposed for further implementations in data structures and algorithms. The extensive analysis is given for justifying the soundness and completeness of F2AC.

1. Introduction

With the pervasive usage of mobile handheld computing devices such as mobile phones, tablets, and laptops, mobile business processing becomes possible and grows largely during commercial traveling. As storage services in cloud, such as Google, Alibaba, and Baidu, are freely provided, users may rely on these services to share and edit business files with others remotely and cooperatively. For example, after one user creates or uploads a file into cloud servers, the others can access the file remotely and edit the file cooperatively.

Currently, mobile storage cloud services impose a typical security problem, access control for distributed users who will access the shared file in cloud servers. In particular, the related access control policies should be determined by user themselves, which result in sophisticated requirements. Moreover, many service providers simply assume that the other users have almost the same access privileges as the original user who creates and uploads files. It can simplify management logics at cloud but raise the risks of file leakage at client. That is, the other users can arbitrarily read, modify, and update uploaded files, which usually imposes security risks in mobile scenarios.

For example, a user (called A) uploads a file into a storage cloud. She tells the other two users (called B and C) user name and password for logging into the storage cloud. B and C can use the user name and password to successfully log into the storage cloud and access the file, like A. In other words, A, B, and C have the same privilege for the file, which is not fine-grained. It may result in security risks such as the leakage or damage of files.

When the number of uploaded files and shared users increases, the fine-grained access control of those files for those users is mandatory. It is worth to note that because access control policies are not determined by the administrator of cloud servers, control methods should be so easy that ordinary cloud users can understand and conduct straightforwardly. It should be flexible in that access control policies are user-centric and can be defined on demand via operational interfaces provided by cloud servers. Moreover, the access control should be lightweight; otherwise, the response delay upon accessing will not be endured and user experiences will be worse to damage the QoS (Quality of Service) of storage cloud services.

Recently, access control in storage cloud has attracted more and more attention [1–3]. However, in those methods

a fine-grained, flexible, and lightweight solution has not been thoroughly explored. We make the first attempt to solve it in this regard. However, such an access control scheme poses three challenges as follows:

(1) The flexible property requires that the proposed method must tackle complicated situations such as highly dynamic ad hoc groups, in which accessing users can join and leave groups conveniently. Moreover, all users may not be in the same layer (in terms of file sharing relationships). That is, a leader of a group can be authorized by the initiator and the leader can further assign and revoke privileges to other group members. A subgroup (or sublayer) can be formed in a group or a layer, so the authorization can be conducted in an iterative manner.

(2) The fine-grained property requires that user privileges should be easily defined, changed, revoked, verified, and managed via various rules and policies. And especially, those can be determined and controlled by cloud user themselves. The access control mechanism should smoothly make it possible for users to edit shared files remotely, cooperatively, and securely.

(3) The lightweight property requires that computational overhead should be managed in cloud side, as the total number of users in storage cloud is always huge. Thus, cryptographic algorithms should be avoided as least as possible. The computational overhead at mobile clients should be as least as possible for better performance in energy consumption and user experiences.

In this paper, we propose a scheme called F2AC (lightweight, fine-grained, and flexible access control) to tackle the above challenges. We present and analyze the design rationale in an incremental way for better understanding. Some formal presentations are presented for better clarity and rigorous generality.

The contributions of the paper are listed as follows.

(1) F2AC can create, add, and delete users in a group (and a subgroup iteratively), authorize a user as a group leader who can authorize privileges to other group users iteratively, and revoke privileges for a user in the group.

(2) F2AC can manage access control such as merge, delete, and retrieve user or privileges in a lightweight manner via a proposed access control model, directed tree with linked leaf model.

(3) F2AC can permit users to define various access control rules as they demand and separate the access control for system users and file users, which simplifies user experiences and management flows in cloud.

The rest of the paper is organized as follows. Section 2 gives an overview on relevant prior work. In Section 3 we discuss the basic assumption used throughout the paper. Section 4 provides the detailed description of our proposed models and analysis. Finally, Section 5 concludes the paper.

2. Related Work

Access control methods for mobile cloud have attracted more and more attention [1, 3, 4]. Ghafoor et al. [5] suggested to enable users to create and manage access control policies on their resources according to their own security and access control requirements. They proposed a framework, standard policy definition language, and user interface to specify and manage access control. Tang et al. [6] designed and implemented a secure overlay cloud storage system that achieves fine-grained, policy-based access control and file assured deletion. Habiba et al. [7] presented framework and different modules along with their functionalities. They also described Multiagent Based System (MAS) and an enhanced authorization scheme. Ruj et al. [4] proposed a new decentralized access control scheme that supports anonymous authentication. Their scheme prevents replay attacks and supports creation, modification, and reading data stored in the cloud. Wan et al. [8] argued that attribute-based encryption (ABE) suffers from inflexibility in implementing complex access control policies. They thus proposed hierarchical attribute-set-based encryption (HASBE) by extending ciphertext-policy attribute-set-based encryption (ASBE) with a hierarchical structure of users. Hajivali et al. [9] proposed agent-based user authentication and access control algorithm based on discretionary and role-based access control model for increasing the reliability and rate of trust in cloud computing environments. Their model uses a cloud-based software-as-a-service application and a client-based user authentication application. Ortiz et al. [10] discussed the industrial application of the extensions to traditional role-based access control to enable secure and mobile collaboration among manufacturing enterprisers. Lv et al. [11] proposed a modified CP-ABE algorithm to set up a fine-grained access control method, in which user revocation is achieved based on the theory of Shamir's secret sharing. Yang et al. [12] proposed to delegate the computation intensive task, such as data reencryption, key distribution, and derivation to cloud servers. Their scheme required bilinear pairing and random padding. Yao et al. [2] proposed a lightweight cipher-text access control mechanism based on authorization certificates and secret sharing for mobile cloud computing. Jung et al. [3] proposed to control privilege and anonymity by fully anonymous ABE. Shen et al. [13] studied the problem of keyword search with access control over encrypted data in cloud. They proposed a framework where user can use his attribute values and a search query to locally derive a search capability, and a file can be retrieved only when its keywords match the query and the user's attribute values can pass the policy check. They also proposed a scheme that utilizes HPE to enforce fine-grained access control, and support the derivation of the search capability. We argue that most of current works extensively rely on ABE, which may not be lightweight due to encryption operations. Moreover, ABE-based schemes cannot support flexible self-defined access policies or cannot be fine-grained in terms of dynamical and iterative privilege authorization (and revocation) when ad hoc groups change and group members vary.

3. Problem Formulation

3.1. Network Model. There exist at least three major entities in file storage cloud: an entity A who uploads files and shares

those files, an entity B who is asked to access these files, and a storage server C who is at cloud side. Simply speaking, A uploads files into C. A gives login account information to B. B logs in and accesses files according to her privilege that is set by A.

In this paper, we will tackle more complicated scenarios and manage access privileges in a fine-grained, flexible, and lightweight manner.

Definition 1 (original user). This is the user who applies for an account from a cloud storage service for uploading files, for example, Alibaba's AliYun. She uploads files into the cloud server. She is denoted by $User_A$ in further examples.

Definition 2 (accessing user). After original user logs in to cloud and uploads files, she will be asked to assign access privileges to some accessing users for those files. Original user also needs to specify accessing user's account information for logging in to cloud systems and token information for distinguishing different users for file accessing.

Definition 3 (team leader). In accessing users, some are team leaders who can assign privileges for shared files in this team. Team leader can also add and remove accessing users in this team.

Definition 4 (team member). Accessing user who is added by a team leader or original user is a team member in this team.

Definition 5 (privilege). An accessing user accesses an uploading file according to her privilege that is authorized by team leader or original user.

Definition 6 (authorize privilege). Original user and team leader can have authorize privilege, who can assign privileges to other users, and add or remove accessing users.

3.2. Design Goals. The design of F2AC should tackle the following situations: (1) Original user and team leader can authorize privileges for a highly dynamic group in which users can be added or removed. (2) Those privileges can be flexibly managed by original user or team leader in an iterative way and accessing hierarchical layers can be diverse and arbitrary. (3) The access control can be conducted in a lightweight manner at cloud, especially when system access control is separated from file access control, which is a realistic deployment requirement in storage cloud.

4. Proposed Scheme: F2AC

4.1. Basic Setting. There exist two tables for facilitating control mechanism at *Cloud*. We denote them as $Cloud ::= \langle ACL, UCL\rangle$, where $::=$ can be looked as "is defined to." ACL is used for access control; UCL is used for user authentication.

(1) ACL is a table for access control that has four fields $\langle File_*, User_*, P, C\rangle$, where $File_*$ denotes a file name, $User_*$ denotes a user name, P denotes a privilege, and C denotes one or more conditions. We denote it as $ACL ::= \langle File_*, User_*, P, C\rangle$.

The privileges usually have four types, *Create*, *Update*, *Modify*, and *Read*, which denote create, update, modify, and read, respectively. Users who are assigned *Read* privilege can only view files. Users who are assigned *Modify* can read and modify files but not update files. Users who are assigned *Update* can read, modify, and update the modified content into files. Users who are assigned *Create* can read, modify, and update files and, especially, can assign privileges to other users. We denote it as $P ::= \langle Create, Update, Modify, Read\rangle$. The former one is the superset of the latter one. That is, *Create* contains the privilege *Update*, *Update* contains *Modify*, and *Modify* contains *Read*. In shorthand, $Read \subset Modify \subset Update \subset Create$.

Extra conditions (column C) can include more requirements for access, for example, locations, device MAC addresses, or any other requirements. The default value for it is Null, denoted as $*$. We denote it as $C ::= \langle *, Loc, MAC\rangle$.

(2) UCL is a table for user control that has two fields $\langle User_*, Token_*\rangle$, where $User_*$ is a username and $Token_*$ is a password for user authentication. We denoted it as $UCL ::= \langle User_*, Token_*\rangle$.

F2AC is composed of the following steps.

Step 1 (user: firstly login and upload). For the first time login, $User_A$ can upload one or more than one files into $Cloud_C$.

Step 2 (user: assign one user and one privilege for one file). Once $User_A$ uploads a file successfully (e.g., $File_{A1}$); *Cloud* will add two records into ACL as follows.

Step 2.1. One record is $\langle File_{A1}, User_A, Create, *\rangle$ that is added into ACL automatically.

Step 2.2. $User_A$ will be asked to set up the access privilege of the file ($File_{A1}$) by *Cloud*. That is, $User_A$ specifies the privilege (e.g., *Read*) of $File_{A1}$ to $User_B$ by checking the options (in user interface) provided by *Cloud*. The setting results will be stored in ACL. That is, the record $\langle File_{A1}, User_B, Read, *\rangle$ is added into ACL.

Step 3 (user: assign more users and their privileges for one file). $User_A$ continues to set up more users and more privileges for this file. *Cloud* add more records in ACL correspondingly. That is, Step 2.2 will be conducted iteratively for $File_{A1}$ for assigning more users and privileges.

Step 4 (user: upload more files and assign more users and their privileges for more files). If $User_A$ uploads additional files, for example, $File_{A2}$, $File_{A3}$, Steps 2 and 3 will be reconducted iteratively for those files by *Cloud* and $User_A$.

Step 5 (cloud: store records into ACL). After $User_A$ uploads all files and sets up all options for access control, *Cloud* will store corresponding records into ACL.

Step 6 (user: assign UCL). $User_A$ sets up corresponding token for each user in ACL including herself by checking options (in user interface) provided by *Cloud*.

Step 7 (cloud: store records into *UCL*). After that, *Cloud* stores corresponding records into *UCL*. For example, $\langle User_A, Token_A\rangle$, $\langle User_B, Token_B\rangle$, $\langle User_C, Token_C\rangle$.

Step 8 (user: send account information and tokens to accessing users). $User_A$ gives her account and password for *Cloud*, together with $Token_B$, to $User_B$; $User_A$ gives her account and password for *Cloud*, together with $Token_C$, to $User_C$. The account and password of $User_B$ and $User_C$ are the same as those of $User_A$. The token is used for authentication and distinction of different accessing users.

Step 9 (user: secondly login and present token). After $User_A$ logs in to *Cloud* at the second time, she will be asked for her token by *Cloud*. The reason is that $User_A$ is in *UCL* at this time.

Step 10 (cloud: retrieve *UCL* and *ACL*). After $User_A$ responds her token, *Cloud* will retrieve *UCL* to get the user name, namely $User_A$. *Cloud* lists all files for $User_A$ by retrieving *ACL* according to her privileges.

Step 11 (other users: login and present token). After $User_B$ logs in to *Cloud*, she will be asked for showing her token by *Cloud*. After $User_B$ responds with her token, *Cloud* will conduct processes similar to Step 10.

Remarks 1. (1) It is required to separate the authentication for logging in to *Cloud* and the authentication for access control, namely, user name in *UCL*. It will simplify the implementation and management of user authentication at *Cloud*. The security of original authentication system of *Cloud* will not be damaged. It is also easy for $User_A$ to understand (for better user experience) and conduct user addition for file accessing.

(2) In Step 2.2, one privilege is already enough, as relationships between privileges are subset (or superset). The highest (or largest) privilege is set for one user aiming at one file.

(3) The condition C could be extra accessing rules on demand, which is set "$*$" in the description for simplicity. Indeed, it can be set as *Loc*, *MAC*, and so on. Hereby, $C ::= \langle *, Loc, MAC\rangle$. C can be considered as the extension of P.

(4) Step 8 is usually accomplished offline or out of the channel, which is out of the scope of the scheme.

4.2. Notations. The major notations used in the remainder of the paper are listed as follows for better understanding.

Cloud: File Storage Cloud

P: Privilege, $P ::= \langle Create, Update, Modify, Read\rangle$

$Login(Cloud, User_*)$: User Login Function. Login into *Cloud* as $User_*$

$Upload(File_*)$: User Upload File Function. Upload file $File_*$

$SaveACL(\langle File_*, User_*, P, *\rangle)$: Cloud Save ACL Function

$SetupACL(\langle File_*, User_*, P, *\rangle)$: User Setup ACL Function

$SetupUCL(User_*, Token_*)$: User Setup UCL Function

$SaveUCL(\langle User_*, Token_*\rangle)$: Cloud Save UCL Function

$RequestToken()$: Function Request Token by Cloud after User Login Returning $Token_*$

$RetrieveUCL(Token_*)$: UCL Retrieve Function by Cloud, Taking as input $Token_*$, Returning $User_*$

$RetrieveACL(User_*)$: ACL Retrieve Function by Cloud, Taking as input $User_*$, Returning $File_*, P, C$.

4.3. Abstract Model. The overall procedures are stated as follows in a shorthand for simplicity and clarity as follows.

Abstract Model for F2AC

Step 1

$$User_A : Login(Cloud, User_A),\ User_A : Upload(File_{A1});$$

Step 2.1

$$Cloud : SaveACL(\langle File_{A1}, User_A, Create, *\rangle);$$

Step 2.2

$$User_A : SetupACL(\langle File_{A1}, User_B, Read, *\rangle),$$
$$Cloud : SaveACL(\langle File_{A1}, User_B, Read, *\rangle);$$

Step 3

$$User_A : SetupACL(\langle File_{A1}, User_C, Modify, *\rangle),$$
$$Cloud : SaveACL(\langle File_{A1}, User_C, Modify, *\rangle);$$
$$User_A : SetupACL(\langle File_{A1}, User_D, Update, *\rangle),$$
$$Cloud : SaveACL(\langle File_{A1}, User_D, Update, *\rangle); \ldots$$

Step 4

$$User_A : Upload(File_{A2});$$
$$User_A : Upload(File_{A3}); \ldots$$

Step 5

Redo Steps 2.1, 2.2, and 3;

Step 6

$$User_A : SetupUCL(User_B, Token_B);$$
$$User_A : SetupUCL(User_C, Token_C);$$

Step 7

$$Cloud : SaveUCL(\langle User_A, Token_A\rangle);$$
$$Cloud : SaveUCL(\langle User_B, Token_B\rangle);$$
$$Cloud : SaveUCL(\langle User_C, Token_C\rangle);$$

Step 8

Off-line Operations.

Step 9

$User_A : Login(Cloud, User_A)$;

Step 10

$Cloud : Token_A \Leftarrow RequestToken()$;
$Cloud : U \Leftarrow RetrieveUCL(Token_A)$;
$Cloud : File_*, P, C \Leftarrow RetrieveACL(U)$;

Step 11

$User_B : Login(Cloud, User_A)$;

Step 12

$Cloud : Token_B \Leftarrow AskToken()$;
$Cloud : User_B \Leftarrow RetrieveUCL(Token_B)$;
$Cloud : File_*, P, C \Leftarrow RetrieveACL(User_B)$;
...

Remarks 2. (1) In Step 1, $User_A$ logs in for file uploading. After this login, *UCL* is created. In $User_A$'s next login (e.g., in Step 9), $User_A$ will be requested for her token. If this time no file is uploaded, *UCL* will not be created. That is, once a file is uploaded by a user, *UCL* and *ACL* will be created at *Cloud*.

(2) In Step 11, if a user (e.g., $User_X$) logs in via not using $User_A$'s account (including user name and password), *Cloud* will respond that this user (namely, $User_X$) is not a file sharing user related to $User_A$. It is an independent login, not being related to the file sharing of $User_A$ (i.e., this login may be used for $User_X$'s file uploading and sharing).

In contrast, if $User_A$ logs in with $User_B$'s account, instead of $User_A$'s, $User_A$ will be regarded as $User_B$'s file sharer and be requested for showing the token that is set by $User_B$.

(3) Usually, *Create* can be only assigned to the user who uploads the file into *Cloud*, by *Cloud* automatically in Step 2.1. In the following sections we will extend it for more flexible functions in more complicated application scenarios.

4.4. Extension for Flexibility. In the above abstract model, the major steps for F2AC are described. In this setting, only user that uploads the file can obtain *Create* privilege and can authorize privileges to others. It can simplify the management of privileges, but it may not be convenient when the file is shared in a dynamic group whose members are changed frequently, or a group with a large number of users or different layers.

For example, $User_A$ uploads $File_A$, but $User_A$ is willing to let $User_B$ be a proxy of her for managing the file editing. Thus, $User_A$ hopes that $User_B$ can authorize privileges to other users as a team leader and represent as a proxy of $User_A$; for example, $User_B$ can further assign privilege such as *Read* to $User_C$. To extend this flexibility, we propose an extension of basic settings and abstract model.

Firstly, $P ::= \langle Create, Update, Modify, Read\rangle$ in basic settings will be extended to $P ::= \langle Create, Authorize, Update, Modify, Read\rangle$. That is, the user with *Authorize* privilege will be able to authorize privileges to other users like $User_A$. Certainly, *Authorize* is a superset of *Update*, *Modify,* and *Read*.

Secondly, in abstract model after Step 12, if P that is returned by $RetrieveACL(User_B)$ is Authorize, $User_B$ will be able to set up *ACL*. Four policies are proposed hereby on methods for setting up *ACL* in the following.

Policy 1 (none-transitive *Authorize* privilege and none-additive users). After $User_B$ with the privilege *Authorize* logs in into *Cloud* as $User_A$, all files are listed. $User_B$ selects a file, for example, $File_{A1}$, and the related users and their privileges are listed. $User_B$ can change those privileges that are *Update, Modify,* and *Read*. That is, only the user with *Create* can assign *Authorize* privilege; the user with *Authorize* cannot assign *Authorize* privilege to others but can change privileges such as *Update*, *Modify*, and *Read*. After any chance happens, *ACL* will be updated for column P for corresponding records.

Policy 2 (none-transitive of *Authorize* privilege and additive users). The difference between this policy and Policy 1 is that the user with *Authorize* privilege can add more users for current files. If so, *ACL* will not only be updated at column P, but also be appended more records.

If the trace back on who adds the user into the group is required, *ACL* will be extended to $\langle File_*, User_*, P, C, ByWhom\rangle$. *ByWhom* will record the user who adds $User_*$. For example, if $User_A$ adds $User_B$ into *ACL*, *ByWhom* will be $User_A$. If $User_B$ with privilege *Authorize* adds $User_C$ into *ACL*, *ByWhom* in this row will be set as $User_B$.

If a new user is appended into *ACL*, this user must be appended into *UCL*. The token of the new user is set up by the user who adds the new user. For example, if $User_B$ with privilege *Authorize* adds $User_C$ into *ACL*, $User_B$ should set up $\langle User_C, Token_C\rangle$ in *UCL*.

The deletion of users in *ACL* is only permitted by the user who matches *ByWhom* in this row. That is, if *ByWhom* is $User_A$ in this row in *ACL*, only $User_A$ can delete this row (namely, the user) from *ACL* (by user interface provided by *Cloud*). Once a user is deleted in *ACL*, the user will be also deleted in *UCL* automatically. For example, if $ProjectColumn(ACL, User_*) ! = User_B$ (i.e., all records like $\langle File_*, User_B, *, *, ByWhom = User_A\rangle$ are deleted in *ACL* by $User_A$), $\langle User_B, Token_B\rangle$ will be deleted from *UCL*.

Policy 3 (transitive of *Authorize* privilege and nonadditive users). The difference between this policy and Policy 1 is that the user with *Authorize* privilege can change privileges *Update, Modify,* and *Read* into *Authorize* in *ACL*. That is, not only the user with *Create* can assign *Authorize* privilege, but also the user with *Authorize* can assign *Authorize* privilege to others. If so, the column P in *ACL* will not only be updated

among *Update, Modify,* and *Read*, but also be updated from *Update*, *Modify*, and *Read* into *Authorize*. However, the user with *Authorize* privilege cannot add more users for current files, namely, nonadditive users.

Policy 4 (transitive of *Authorize* privilege and additive users). This one will be most complicated and flexible. It combines functions in Policies 2 and 3. That is, the user with *Authorize* can assign *Authorize* privilege to others, and the user with *Authorize* privilege can also add more users for current shared files.

Remarks 3. (1) Each user has only one possible *ByWhom* in *ACL*. Once a user is added into *ACL*, the *ByWhom* field will be set. For better understanding, if we consider *ByWhom* as a parent (namely, father) and consider each user as a node, user graph will be considered as a directed tree. The direction is from a child or all children to the father. Recall that, for user deletion, only the father can delete directed child or children in the tree (if user *B*'s *ByWhom* in *ACL* is user *A* who currently logs in to *Cloud*, user *A* will be presented an option on whether deleting *B* or not in user interface).

Note that, for consistence, if a none-leaf node is deleted, the node's all children will replace their fathers by the node's father. (All children's *ByWhom* will be replaced by the node's *ByWhom*.)

(2) A user's privilege for a file can be modified by anyone who has *Create* or *Authorize* privilege for this file. For better understanding, we can look users with *Creat* and *Authorize* for a file as team leaders for this file. Other users with privileges such as *Update*, *Modify*, or *Read* for this file will be considered as group members for this file.

(3) Once a user is added into *ACL*, she will be added into *UCL*. The same user name presents only once in *UCL*. For better understanding, all users in *UCL* can be considered as a group whose members share the same login account information (namely, login user name and password) in *Cloud* as the original user who uploads shared files.

(4) For each file, only one user can have *Create* privilege. The user who has *Create* privilege is the original user (the one who uploads the file into *Cloud*). All users in *UCL* will share the same account information with this user who has *Create* privilege.

(5) A user's father (according to *ByWhom* field) has responsibility to deliver the account information and token to the user, usually offline.

4.5. Directed Tree with Linked Leaf Model. Policy 4 is the most flexible (or powerful) one in all policies for file sharing in a dynamic group or a group with a large number of members. Intuitively, someone may suspect that this policy may result in some inconsistency due to the complexity of user addition, deletion, and transitive authorization. To make it clear, we propose a concept and implementation model for F2AC, which presents as a directed tree with linked leaf (leaves).

(1) Users are organized in a tree structure. Each user presents as one node at the tree. The root of the tree is the user who uploads all current sharing files, namely, one who has *Create* privilege for those files. All users in the tree share the same login account information as the root, but they are distinguished via their tokens.

(2) Each nonroot node has one parent (namely, father). Anyone of these nodes has a directed edge pointed to its father. The father is the user who adds a child node or children nodes in the tree. Only users who have *Authenticate* or *Create* privilege can add a child or children. The father of a child can be fetched from *ACL* by looking up *ByWhom* column.

(3) Each node has three properties, namely, *UserName*, *Token*, *OneParent*. *UserName* and *Token* are assigned by node's father. The property *OneParent* points to node's father.

(4) Each node links to one or multiple properties, called *NodeLink*. Each link has two related properties. One is *AccessFileSet*; the other is *OnePrivilege*. *AccessFileSet* property is a set consisting of one or more files; *OnePrivilege* property is one privilege for files in *AccessFileSet*. Each node may have more than one *AccessFileSet*, but each *AccessFileSet* has one corresponding *OnePrivilege* (namely, the largest privilege). That is, each node links to one or more sets that represent access files, and each set has one bit to represent the privilege for this set. We look on one or multiple properties as linked leaf (leaves) for this node. The model is thus called a tree with linked leaf (leaves). Note that linked leaf is not a leaf node; leaf node is a node that has no child, but linked leaf is a property for every node.

(5) The file set (usually consisting of multiple files) at the father node should be the superset of that (namely, file set) at a child node or children. It is nontrivial to be aware of but can be understood for the reason that father's files are reassigned to a child or children to edit. Thus, a merging set of a child's or children's *AccessFileSet* is upper-bounded by the father's *AccessFileSet*.

(6) *ACL* and *UCL* can be constructed by the above tree model. The retrieval of *ACL* and *UCL* can be accomplished by underlying tree data structures and algorithms.

We denote the tree model in the following.

Directed Tree with Linked Leaf Model

$$
\begin{aligned}
&Tree ::= \langle Root, NodeSet, Edges\rangle,\\
&FileSet ::= \{files \mid uploadedbyRoot\},\\
&Root ::= \langle UserName, Token, OneParent = NULL, RootLink\rangle,\\
&RootLink ::= \langle AccessFileSet = FileSet, OnePrivilege = Create\rangle,\\
&NodeSet ::= \langle Node\rangle, \ldots, \langle Node\rangle\\
&Node ::= \langle UserName, Token, OneParent, NodeLink\rangle,\\
&NodeLink ::= \langle AccessFileSet, OnePrivilege\rangle, \ldots, \langle AccessFileSet, OnePrivilege\rangle,\\
&AccessFileSet ::= \{onefile \in FileSet\} \parallel morefiles \subseteq FileSet,\\
&OnePrivilege ::= \{P \mid P = Authorize \parallel Update \parallel Modify \parallel Read\},\\
&Edges ::= \langle Edge\rangle, \ldots, \langle Edge\rangle,\\
&Edge ::= \langle Node_1, Node_2\rangle \mid \{Node_1.OneParent = Node_2,
\end{aligned}
$$

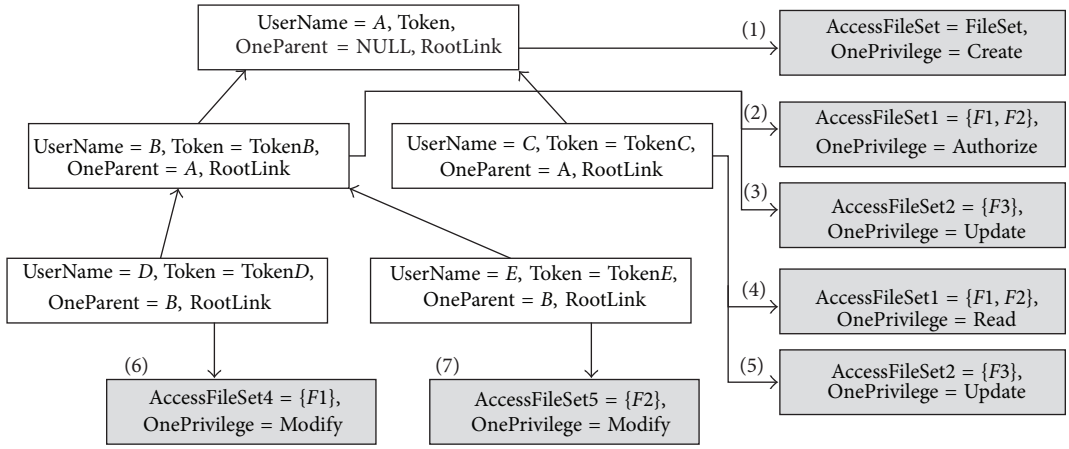

Figure 1: An example for illustration the logic of setup.

$Node_1, Node_2 \in NodeSet$, $Node_2.OnePrivilege = \{Create \parallel Authorize\}\}$,

Example 7. We explain the logic in following example, which follows the sequence number listed in Figure 1 (grey nodes are linked leaves).

(1) $FileSet = F1, F2, F3$. User A (e.g., original user, manager) uploads all three files into *Cloud*. A's privilege for all files is *Create*. A's login account information for *Cloud* will be used for all other users, namely, B, C, D, and E.

(2) $F1$ and $F2$ are two files that will be edited by a team led by B (B is the leader of team 1, e.g., technical team).

(3) $F3$ can be edited by B (by herself, e.g., $F3$ is written by B and will be reported to manager A).

(4) $F1$ and $F2$ can only be read by C (C is from another team, e.g., testing team).

(5) $F3$ can be edited by C (by herself, as $F3$ is written by C and will be reported to manager A). Note that, hereby $F3$ can be edited by B and C cooperatively.

(6) B is the team leader. She adds two users into her team, namely, D and E. B assigns D to edit $F1$, but D cannot update $F1$. The modification on $F1$ by D can be reviewed by B and updated by B.

(7) B assigns E to edit $F2$, but E cannot update $F2$. The modification on $F2$ by E will be reviewed by B and updated by B.

4.6. Constraints and Principles. Next, we point out some nontrivial insightful constraints to depict some inner principles in F2AC to evaluate its soundness.

Principle 1. The number of leaves at *Root* is one. The total number of leaves ($|NodeLink|$, namely, $\langle AccessFileSet, OnePrivilege\rangle$ pairs) at a child of *Root* is upper bounded by $|FileSet|$. The upper bound can be achieved, because each one file has one privilege. We will explain how to merge the leaves that have the same privilege. Thus, the number of total possible leaves for a node would be the number of possible privileges (namely, four, they are *Authorize*, *Update*, *Modify*, and *Read*).

Principle 2. The merging set of *AccessFileSet* in leaves at a child of *Root* is upper bounded by *FileSet*.

Principle 3. The merging set of *AccessFileSet* in leaves at all children of *Root* is upper bounded by *FileSet*.

Principle 4. The number of children of *Root* (namely, the number of users added by *Root*) has no limit. It depends on the editing logic of shared files; thus, the number of members in an initial group could be very large.

Principle 5. The total number of leaves on a child of a node (not *Root*) is upper bounded by $|AccessFileSet|$ in the node's leaf whose *OnePrivilege* is *Authorize*. It is similar to Principle 1.

Principle 6. The merging set of *AccessFileSet* in leaves on a child of a node (not *Root*) is upper bounded by *AccessFileSet* in the node's leaf whose *OnePrivilege* is *Authorize*. It is similar to Principle 2.

Principle 7. The merging set of *AccessFileSet* in leaves at all children of a node (not *Root*) is upper bounded by *AccessFileSet* in the node's leaf whose *OnePrivilege* is *Authorize*. It is similar to Principle 3.

Principle 8. The number of children of a node has no limit. It is similar to Principle 4.

Principle 9. The birth sequence of nodes in the tree is from an upper layer to a lower layer (for a subtree). That is, the growth of the tree is from upper to lower. Thus, users are added into the tree from upper to lower sequentially.

Principle 10. Usually, a physical person has one *Token* (and corresponding *UserName*). Our scheme has the flexibility that one physical person can be given more than one *Token*. There is another way to achieve the functionality that more *Token*s are held by one physical person. That is, usually *Token*s in $\langle UserName, Token\rangle$ are distinct for different *UserName*, but same *Token* for different *UserName* will achieve the goal

of more *Token*s at a physical person. (The reason is that *Token* is used for the authentication of access control.)

Principle 11. One node cannot have two parents (fathers). That is, if one *UserName* with corresponding *Token* is created and assigned, the *UserName* will not be reused by other nodes for assigning. This principle will simplify the tree. Nonetheless, the F2AC can also achieve that one node has more than one father. The method is that you add the node at first and let two children's *Token* be the same.

Principle 12. The deletion of a user can be done by the deletion of a node. If the node has children (namely, not a leaf node), the deletion of the node will let the node's father be the node's children's father. It seems to replace the team leader. If the node has no children (at lowest layer), the deletion of the node will be done directly. Certainly, the deletion of a node will remove the node's linked leaves together.

Principle 13. The modification of user's privilege can be done by modifying the *OnePrivilege* on the linked leaves of the node. *Update*, *Modify*, and *Read* can be changed into each other among them. *Update*, *Modify*, and *Read* can be changed into *Authorize*. We reserve the flexibility that *Authorize* can also be changed into *Update*, *Modify*, and *Read*, but it may need to delete subtree of this node for the consistence of principles.

Principle 14. For simplicity and better understanding, we do not include the further accessing conditions, denoted as C, in the design of linked leaves. Indeed, it is without loss of generality. The C can be considered as an extension of P, as C specifies more requirements in accessing policies. Therefore, *NodeLink* can be extended into $\langle AccessFileSet, OnePrivilege, ExtraConditions\rangle$ if required.

4.7. Further Extension for Lightweight. As stated in aforementioned Principle 1, linked leaves of a node may be too many. We propose several algorithms to merge leaves in the tree structure or merge records in *ACL*. We take *ACL* as an example to describe our algorithms. The algorithm for tree structure can be done accordingly.

ACL has four tuples $\langle File_*, User_*, P, C\rangle$. We change it into $\langle User_*, File_*, P \parallel C\rangle$ for matching with tree model. Next, we will propose algorithms to accelerate the retrieval delay in *ACL* (and also in the tree).

$\langle User_*, File_*\rangle$ is cartesian product (simply speaking, many-to-many). $\langle File_*, P\rangle$ is many-to-one. $\langle User_* \times File_*, P\rangle$ is many-to-one. Thus, we can combine files that have the same privilege for a single user as one record to decrease the number of records in *ACL* (and retrieval delay). It is the so-called $\langle AccessFileSet\rangle$ in the tree. For each user (except for original user), there are at most four file sets, as the maximal number of privileges are four (namely, *Authorize*, *Update*, *Modify*, and *Read*). Thus, the linked leaves for each node will be at most four.

If we look *AccessFileSet* as a single item in the field $File_*$, the number of records in *ACL* can further be decreased by merging users. $\langle User_*, File_* \times P\rangle$ is cartesian product. As the intersection of *AccessFileSet* is empty set, $File_* \times P$ may be the same for different users. In this situation, those $User_*$ fields can be combined into one item. This can be looked as the merging of nodes that have the same linked leaves in the tree.

The retrieval of *ACL* can be considered as the traverse in the tree. After a user logs in and presents her token, the node is determined in the tree. The linked leaves can be fast fetched for listing (or looking up) files that can be accessed together with corresponding privileges. Also, the node's children can be listed for addition and removal, if applicable. Thus, the proposed tree-based model provides critical support for algorithms (and functions) in the access control mechanism.

5. Conclusions

In this paper, we proposed a lightweight, fine-grained, and flexible scheme, called F2AC, for access control in multiple-party file editing and sharing in mobile cloud computing. F2AC can support dynamically adding and deleting users in an ad hoc group, privilege self-defining as a creator's proxy or team leader, transitively authorizing privileges for members in subteams, transitively revoking privileges, and separating of access authentication from system authentication. The directed tree with linked leaf (leaves) model is proposed for lightweight implementation and verification. The leaf merging and node merging are described for lightweight storage and fast retrieval of privileges. The future work could be the evaluation of linked leaf model in some mainstream cloud services such as Apple iCloud, Baidu Cloud, and Alibaba Cloud.

Conflict of Interests

The authors declare that there is no conflict of interests regarding the publication of this paper.

Acknowledgments

The research was financially supported by the National Natural Science Foundation of China under Grant nos. 61170217, 61502440, 61301166, and 61363069, the Fundamental Research Funds for the Central Universities, and China University of Geosciences (Wuhan) (Grants nos. CUGL150831 and CUGL150416).

References

[1] X. Li and X. Zhao, "Survey on access control model in cloud computing environment," in *Proceedings of the International Conference on Cloud Computing and Big Data (CloudCom-Asia '13)*, pp. 340–345, IEEE, Fuzhou, China, December 2013.

[2] X. Yao, X. Han, and X. Du, "A lightweight access control mechanism for mobile cloud computing," in *Proceedings of the IEEE Conference on Computer Communications Workshops (INFOCOM '14)*, pp. 380–385, April 2014.

[3] T. Jung, X.-Y. Li, Z. Wan, and M. Wan, "Control cloud data access privilege and anonymity with fully anonymous attribute-based encryption," *IEEE Transactions on Information Forensics and Security*, vol. 10, no. 1, pp. 190–199, 2015.

[4] S. Ruj, M. Stojmenovic, and A. Nayak, "Decentralized access control with anonymous authentication of data stored in clouds," *IEEE Transactions on Parallel and Distributed Systems*, vol. 25, no. 2, pp. 384–394, 2014.

[5] A. Ghafoor, M. Irum, and M. Qaisar, "User centric access control policy management framework for cloud applications," in *Proceedings of the 2nd National Conference on Information Assurance (NCIA '13)*, pp. 135–140, IEEE, Rawalpindi, Pakistan, December 2013.

[6] Y. Tang, P. P. C. Lee, J. C. S. Lui, and R. Perlman, "Secure overlay cloud storage with access control and assured deletion," *IEEE Transactions on Dependable and Secure Computing*, vol. 9, no. 6, pp. 903–916, 2012.

[7] M. Habiba, M. R. Islam, and A. B. M. S. Ali, "Access control management for cloud," in *Proceedings of the 12th IEEE International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom '13)*, pp. 485–492, IEEE, Melbourne, VIC, Australia, July 2013.

[8] Z. Wan, J. Liu, and R. H. Deng, "HASBE: a hierarchical attribute-based solution for flexible and scalable access control in cloud computing," *IEEE Transactions on Information Forensics and Security*, vol. 7, no. 2, pp. 743–754, 2012.

[9] M. Hajivali, M. T. Alrashdan, F. Fatemi Moghaddam, and A. Z. M. Alothmani, "Applying an agent-based user authentication and access control model for cloud servers," in *Proceedings of the International Conference on ICT Convergence (ICTC '13)*, pp. 807–812, IEEE, Jeju, South Korea, October 2013.

[10] P. Ortiz, O. Lazaro, M. Uriarte, and M. Carnerero, "Enhanced multi-domain access control for secure mobile collaboration through linked data cloud in manufacturing," in *Proceedings of the IEEE 14th International Symposium and Workshops on a World of Wireless, Mobile and Multimedia Networks (WoWMoM '13)*, pp. 1–9, IEEE, Madrid, Spain, June 2013.

[11] Z. Lv, C. Hong, M. Zhang, and D. Feng, "A secure and efficient revocation scheme for fine-grained access control in cloud storage," in *Proceedings of the 4th IEEE International Conference on Cloud Computing Technology and Science (CloudCom '12)*, pp. 545–550, Taipei, Taiwan, December 2012.

[12] R. Yang, C. Lin, and Y. Jiang, "Enforcing scalable and dynamic hierarchical access control in cloud computing," in *Proceedings of the IEEE International Conference on Communications (ICC '12)*, pp. 923–927, IEEE, Ottawa, Canada, June 2012.

[13] Z. Shen, J. Shu, and W. Xue, "Keyword search with access control over encrypted data in cloud computing," in *Proceedings of the 22nd IEEE International Symposium of Quality of Service (IWQoS '14)*, pp. 87–92, Hong Kong, May 2014.