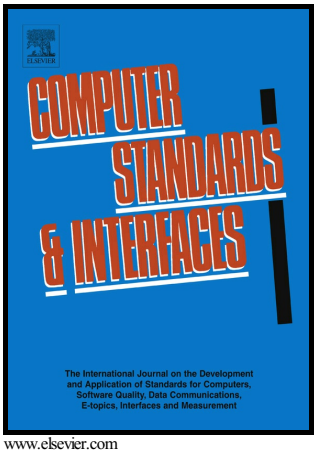
SeGoAC: A tree-based model for self-defined, proxy-enabled and group-oriented access control in mobile cloud computing

Wei Ren, Ran Liu, Min Lei, Kim-Kwang Raymond Choo



www.elsevier.com

PII: S0920-5489(16)30073-3
DOI: http://dx.doi.org/10.1016/j.csi.2016.09.001
Reference: CSI3133

To appear in: *Computer Standards & Interfaces*

Received date: 13 March 2016
Revised date: 31 August 2016
Accepted date: 4 September 2016

Cite this article as: Wei Ren, Ran Liu, Min Lei and Kim-Kwang Raymond Choo SeGoAC: A tree-based model for self-defined, proxy-enabled and group-oriented access control in mobile cloud computing, *Computer Standards & Interfaces* http://dx.doi.org/10.1016/j.csi.2016.09.001

SeGoAC: A Tree-based Model for Self-Defined, Proxy-Enabled and Group-Oriented Access Control in Mobile Cloud Computing

Wei Ren[a,b], Ran Liu[a], Min Lei[c], Kim-Kwang Raymond Choo[d,a]

[a]*School of Computer Science, China University of Geosciences, Wuhan, China*
[b]*Hubei Key Laboratory of Intelligent Geo-Information Processing (China University of Geosciences (Wuhan)), Wuhan, Hubei, China*
[c]*Information Security Center, Beijing University of Post and Telecommunications, Beijing, China*
[d]*Department of Information Systems and Cyber Security, University of Texas at San Antonio, San Antonio, U.S.A.*

Abstract

Designing an effective and secure group-oriented access control for mobile cloud storage services is an area of active research. For example, such schemes should provide user-friendly features that allow group members to be conveniently added or removed, privileges of group members to be assigned or revoked by authorized parties (e.g. group leaders), organizing of members into one or more sub-groups, forming of (multiple) hierarchical layers, etc. Specifically, privileges should be self-defined by group leaders, and access control can be carried out by group leaders as a proxy. In this paper, we propose a lightweight tree-based model designed to achieve self-defined, proxy-enabled and group-oriented access control (hereafter referred to as SeGoAC) for file storage access control in mobile cloud computing. SoGoAC is a flexible access control model that supports group access control, self-authorization and self-management iteratively, flexible self-defined accessing policies, user friendly features to grant and revoke privileges. We then demonstrate the utility of SeGoAC via extensive analysis.

Email addresses: weirencs@cug.edu.cn (Wei Ren), ranliu_cug@foxmail.com (Ran Liu), leimin@bupt.edu.cn (Min Lei), raymond.choo@fulbrightmail.org (Kim-Kwang Raymond Choo)

Keywords:
Access Control; Mobile Cloud Computing; Lightweight; Flexibility

1. Introduction

In mobile cloud computing, users can upload files into cloud storage servers for sharing, storage, etc. These files can then be accessed by authorized users remotely and collaboratively. For example, after user X has uploaded some files into the cloud storage services via a mobile device, other authorized users can access one or more of these files remotely. With the appropriate privileges, they can either view, edit, delete or copy the files via any digital device (e.g. Android and iOS devices) [1].

Providing an effective and secure access control mechanism in such a distributed environment remains a research challenge. Specifically, in a (mobile) cloud environment, the access control policies should be determined or self-defined by the users (rather than a central authority such as cloud server administrators). Generally, cloud service providers simply assume that all users have similar access privileges for files that have been uploaded for sharing (e.g. all users for a shared file have the same privileges, such as read or write). In other words, access control is not fine-grained and this may result in information leakage and other security threats.

Moreover, in a mobile business setting, files are usually shared and edited by one or more groups (e.g. users from marketing and other departments or different organizations). The group membership may also change regularly or an ad-hoc basis (e.g. due to change of status from draft tender proposal to ready for review by management). Also in most scenarios, a group leader may delegate access to the file(s) to one or more group members. Thus, the group leader needs to be able to assign access privileges for members in the group. The access control policies should be self-defined by group leaders, and access control self-managed for privilege granting and revocation. In such a setting, the role of the cloud servers is limited to executing self-defined policies. The group may be organized as multiple layers in that smaller sub-groups may be included within a larger group. The access control mechanism should also be flexible for normal mobile users as well as being lightweight to support a large number of mobile users. Existing access control schemes are not capable of supporting these features and this is the gap we seek to address in this paper.

The limitations of existing access control schemes can be summarized as follows:

1. Traditional access control schemes are not specifically designed for mobile cloud computing; thus, the access control is defined and managed by a central authority (e.g. cloud service administrator) rather than the users.
2. The need for flexible self-defined, proxy-enabled and group-oriented access control complicates the challenge of designing schemes to be lightweight and scalable.
3. Traditional access control design rationale generally uses a static concept model, which is inadequate for a dynamic group demanding dynamic privilege updates to dynamic files.

In this paper, we propose a tree-based model, SeGoAC, seeking to overcome the above limitations. Specifically,

1) SeGoAC supports self-defined privilege grant and revocation for group members, and multiple groups can be organized in hierarchical layers.

2) SeGoAC supports self-managed access control in which group leader performs as a proxy of upper layer in accessing tree, and the proxy structure can be iterative.

3) SeGoAC is lightweight as we only involve two tables, and accessing a tree can be constructed from tables for dynamically control management.

The rest of the paper is organized as follows. Section 2 presents an overview of related work. In Section 3, we present the basic assumption used in this paper. Section 4 details our proposed models and analysis. Finally, Section 5 concludes the paper.

2. Related Work

There are known security and privacy challenges for cloud computing [19, 2, 3, 20], and access control schemes are one viable solution. This, and the increasing popularity of mobile devices and cloud services, perhaps drive the increased interest of researchers in designing access control for mobile cloud computing. Tang et al. [4], for example, designed and implemented a secure overlay cloud storage system that provides fine-grained, policy-based access control and file deletion. Habiba et al. [5] presented a framework and different modules, as well as a multi agent-based system and an enhanced authorization scheme. More recently in 2014, Ruj et al. [6] presented a decentralized access control scheme to support anonymous authentication.

A number of existing schemes requires computational intensive operations. For example, Wan et al. [7] explained that since attribute-based encryption (ABE) is inflexible for implementing complex access control policies, they proposed a hierarchical attribute-set-based encryption (HASBE) by extending the ciphertext-policy attribute-set-based encryption (ASBE) with a hierarchical structure of users. Lv et al. [9] presented a modified CP-ABE algorithm to be used as a fine-grained access control method. In their approach, user revocation is achieved using the principle of secret sharing. Yang et al. [10] proposed delegating computationally intensive tasks (e.g. data re-encryption, key distribution and key derivation) to cloud servers. Their scheme requires the use of bilinear pairing and random padding. Jung et al. [12] proposed using fully anonymous ABE to control privilege and anonymity. In the same year, Wang et al.[13] proposed A constant-size ciphertext policy comparative ABE based on negative attributes and wildcards.

There have also been attempts to design lightweight and scalable access control schemes. For example, Yao et al. [11] proposed a lightweight ciphertext access control mechanism that is based on authorization certificates and secret sharing. Ortiz et al. [8] discussed the industrial application of extending traditional role-based access control to support secure and mobile collaboration among manufacturing enterprisers. Zhao et al. [14] proposed a security framework across distributed cloud data centers, and Tu et al. [15] proposed a new access architecture that introduces a layer between mobile devices and the underlying cloud infrastructure. This middle layer consists of cloudlets, which are deployed by cloud services providers. However, deployment of the approach is difficult and costly in practice.

A number of ABE-based schemes have also been published in the literature. In 2015, Jin et al. [16] presented a lightweight data access control scheme based on ciphertext-policy ABE (CP-ABE) designed to ensure the confidentiality of outsourced data and provide fine-grained data access control in mobile cloud computing. Lv et al. [17] proposed an attribute-based access control scheme for mobile cloud storage, using Key-Policy ABE (KP-ABE) scheme with outsourced key generation and decryption. More recently in 2016, Xie [18] proposed a hierarchical access control method using modified hierarchical attribute-based encryption (M-HABE) and a modified three-layer structure. As such schemes use ABE as the underlying structure, they may not be scalable.

3. Problem Formulation

3.1. Network Model

Generally, a file storage cloud has the following entities:

1. Cloud Servers: a storage service provider offering users the capability to store their files for later access.
2. File Creators: users who upload their files to cloud servers for storage, sharing and dissemination.
3. File Sharers: users who access files stored on cloud servers.

As access control is self-defined by users, cloud service authentication and file access authentication should be separated. For the authentication of cloud services, file creators and file sharers share the same user name and password to login to the cloud servers. For the authentication of file accessing, file creators and file sharers are distinguished by the tokens linked with different privileges for different files.

For example, file creator A uploads files to cloud server C. A (Creator) sets up B (Sharer)'s token and the associated privileges for the files. A gives the login account information (user name and password for cloud services, and token) to B. B logins to C (Server) using A's user name and password, and presents the token to C. C allows or denies B access to the files based on the privileges defined by A.

3.2. Design Goals

The design goals are to provide a secure and lightweight scheme which is self-defined, proxy-enabled, and group-oriented:

1. File sharers can be added to and removed from an authorized group easily and efficiently. File sharers can be organized into multiple groups as multiple hierarchical layers, which match file management styles in practice.
2. Access control policies for shared files should be self-defined and as flexible as possible. Accessing privilege assignment and revocation can be conducted by a proxy of file creators. The proxy mechanism can be iteratively granted.
3. The scheme needs to be lightweight and allows for separation of cloud authentication and file access authentication. In other words, cloud servers only perform the role of an executer of access control policies.

4. Proposed Scheme

4.1. Physical Model

In the scheme, a cloud server requires two tables, namely: ACL for access control and UCL for user authentication:

$$ACL = \langle Token, FileName, Privilege \rangle$$

$$UCL = \langle UserName, Token, Father \rangle$$

ACL has three fields, namely: $\langle Token, FileName, Privilege \rangle$ for file access control, where $Token$ denotes a ticket for access control authentication, $FileName$ denotes a file list for accessing files, and $Privilege$ denotes accessing privileges for those files. $Token$ is used to distinguish different file accessing users, even when they use the same username and password to login to the cloud servers. $FileName$ is usually a file list consisting of multiple file names for a given privilege. The privileges usually have five types, namely: $Read$, $Modify$, $Update$, $Authorize$, $Create$ (i.e. $Privilege = \{Create \| Authorize \| Update \| Modify \| Read\}$). Users who are assigned the $Read$ privilege can only read files, and users who are assigned $Modify$ can read and modify files, but cannot update files. Users who have been assigned $Update$ can read, modify and update the modified content into files. Users who have $Authorize$ privilege can read, modify, and update files, as well as assigning privileges to other users. Users who have $Create$ privilege are file creators who initialize and upload files to cloud servers. Thus, $Read \subset Modify \subset Update \subset Authorize \subset Create$. Only the user with the highest privilege level is required to assign for a file set.

UCL has three fields, namely: $\langle UserName, Token, Father \rangle$, where $UserName$ is a user name for logging into cloud servers, $Token$ is a password for file accessing user authentication, and $Father$ denotes the token who add this particular token. For example, if $Token_b$ is added by $Token_a$, $Token_b$'s $Father$ is $Token_a$. If a user is a file creator, then $Father$ of this user will be Null. $Father$ is used for maintaining a deriving relation (group leading relation) between accessing users. The users who are added to the accessing users by their father, and may be removed from accessing users by their father.

Although UCL can be merged into ACL, it will be more efficient when these two tables are separated. Having a separate UCL can reduce the length of ACL and fetching delay of fields (e.g. $Token$ and $Father$) upon successful login, which are critical for supporting a large scale of uses in cloud.

4.2. Process Model

File sharing with access control consists of the following steps:

(Step 1) (file creator logs in) A user ($UserName_a$) wishes to upload files to the cloud servers and share with other users. The user logins to the cloud servers with the registered user name, $UserName_a$, and password.

(Step 2) (cloud server fetches UCL) Upon the user's successful login, the cloud server will fetch $UserName_a$ in UCL.

If $UserName_a \notin UCL$, then the cloud server will label the user a file creator. The user will then be asked to set up a token by the cloud server via its user interface. The record $\langle UserName_a, Token_a, Father = Null\rangle$ will then be added to UCL, prior to skipping to (Step 3).

If $UserName_a \in UCL$, then the cloud server will label the user a file sharer and will prompt the user for his/her token. If the token is not $Token_a$, then skip to (Step 8). Otherwise, go to (Step 3).

(Step 3) (file creator uploads a file) Alice uploads a file to the cloud server.

(Step 4) (assigns one user privilege to the file) Once the user has successfully uploaded a file (e.g. $FileName_{a1}$) to the cloud server, the cloud server will add three records into ACL as follows: (Step 4.1) One record $\langle Token = Token_a, FileName_{a1}, Create\rangle$ is added to ACL. (Step 4.2) Alice will be asked to set up the token of the accessing user by the cloud server via its user interface. For example, Alice sets up $Token_b$ for an accessing user called Bob, then $\langle UserName_a, Token_b, Father = Token_a\rangle$ is appended to UCL. (Step 4.3) Alice will be asked to set up the access privilege of $FileName_{a1}$ by the cloud server via its user interface. For example, Alice specifies the *Read* privilege of $FileName_{a1}$ to $Token_b$, then this configuration will be stored in ACL. That is, the record $\langle Token_b, FileName_{a1}, Read\rangle$ is appended to ACL.

(Step 5) (assigns multiple users privilege to the file) Alice sets up more users and different privileges for this file via a user interface provided by the cloud server. The latter will append the relevant records in ACL accordingly. That is, (Step 4.2) and (Step 4.3) will be repeated for the users.

(Step 6) (uploads more files and assigns more users privileges to newly uploaded files) If Alice uploads additional files (e.g. $FileName_{a2}$ and $FileName_{a3}$), then (Step 4.1), (Step 4.2), and (Step 4.3) will be repeated for these newly uploaded files. In the context of our example, this will result in $\langle Token_a, FileName_{a2}, Create\rangle$ being added to ACL; $\langle UserName_a, Token_c, Father = Token_a\rangle$ added to UCL, and $\langle Token_b, FileName_{a2}, Modify\rangle$ and $\langle Token_c, FileName_{a3}, Authorize\rangle$ added to ACL.

(Step 7) (sends account information and tokens to accessing users) Alice provides her username ($UserName_a$) and password for the cloud server, together with the corresponding token to designated accessing user (e.g. $Token_b$ to Bob, and $Token_c$ to Carolyn, where $Token_b \neq Token_c$).

(Step 8) (file sharers login and present token) When a file sharer login to the cloud server, he/she will be asked for the token.

(Step 9) (retrieves UCL and ACL.) Upon receiving the token from the file sharer, the cloud server will retrieve UCL to verify the token. If token is valid, cloud servers list all files and corresponding the privileges by retrieving ACL according to the token. For example, after Bob presents his token, the cloud server will verify whether $\langle UserName_a, Token_b \rangle$ is in UCL to decide whether Bob can access Alice's uploaded files. In addition, the cloud server fetches all $FileName$ and $Privilege$ in ACL to monitor and check the accessing privileges. If $Token_a$ is presented, then it implies that the file creator login to the cloud servers and go to (Step 3).

4.3. Abstract Model

The abstract model for the above processes is presented in Tab. 1.

Remarks

1) In (Step 1), Alice login to the cloud server in order to load files. Upon successful login, UCL stores her token. In Step 8, Alice will be requested for her token. If the token being presented is Alice's, then she can upload additional files.

2) Once a file has been uploaded by a user, (Step 4.1) to (Step 4.3) will be conducted. Three records will be added to UCL and ACL at the cloud server.

3) In (Step 8), if a user login to the cloud server not using Alice's account (i.e. user name and password), cloud servers will deny access indicating that this user is not a file sharer authorized by Alice. If Bob login using Carolyn's account, instead of Alice's, then Bob will be regarded Carolyn's file sharer and be requested to present the token sent by Carolyn.

4.4. Logical Model

We propose a logical model, which is represented as a directed tree with linked leaf (or leaves). Recall that $ACL = \langle Token, FileName, Privilege \rangle$, $UCL = \langle UserName, Token, Father \rangle$. Thus, ACL is used to construct linked leaves on a node, and UCL for constructing a tree of nodes.

Using UCL, a tree can be constructed with nodes that present file sharers, and root that presents a file creator.

1) Users are organized in a tree structure. Each user is distinguished by the token and is presented as a node in the tree. The root of the tree is the file creator who uploads all current sharing files, namely, users who have *Create* privilege for those files. All users in the tree share the same login account information as those of the root, but they are distinguished by their tokens.

2) Each node excluding the root has one father. Anyone of these nodes has a directed edge pointed to its father. A user's father is the group leader who adds the user into the group. Only users who have *Authenticate* or *Create* privilege can add additional users. The father of a node can be fetched from UCL by looking up *Father* column.

3) Each node has two properties, namely, *Token* and *Father*. *Token* is assigned by node's father. *Father* is used to point to node's father.

Using ACL, each node at the tree can be linked to a leaf (or leaves) that is (are) specific to accessing files and corresponding privileges.

4) Each node links to one or multiple leaves, called *NodeLink*. Each link has two related properties. One is *AccessFileSet*; The other is *Privilege*. *AccessFileSet* property is a set consisting of one or more files; *Privilege* property is one privilege for all files in *AccessFileSet*. Each node may have more than one *AccessFileSet*, but each *AccessFileSet* has only one corresponding *Privilege* (namely, the highest privilege). Note that, a linked leaf is not a node: two properties of a linked leaf are *AccessFileSet* and *Privilege*; but two properties of a node are *Token* and *Father*.

We denote the proposed logical model as follows in Tab. 2.

Remarks

1) ACTree consists of *Root*, *NodeSet* and *Edges*.

2) *Root* has three properties, namely, *Token*, *Father*, and *RootLink*. Here, $Father = NULL$.

3) *RootLink* links *AccessFileSet* that specifies a file set including names of sharing files, and *Privilege* that specifies the accessing right, which is *Create*.

4) *NodeSet* is instantiated by nodes in the tree. Each node has three properties - *Token*, *Father*, and *NodeLink*. Similarly to *RootLink*, *NodeLink* consists of *AccessFileSet* and *Privilege* that specify accessing files and corresponding privileges.

Examples

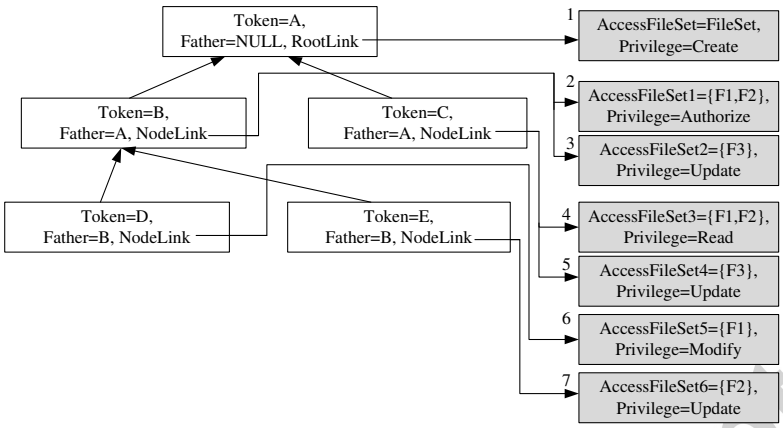


Figure 1: An Example for Illustrating the Logical Model. Grey parts are linked leaves.

We explain the logic in the following examples.

The explanation items correspond to sequence numbers listed in Fig. 1.

1) $FileSet = F1, F2, F3$. User A (e.g., file creator, manager) uploads three files $(F1, F2, F3)$ into cloud servers. A's privilege for all files are $Create$. A's login account information for cloud servers will be used for all other users's login, namely, B, C, D, E.

2) F1 and F2 are two files that will be edited by a group leaded by B (B is the leader of group 1, e.g., technical group in a company).

3) F3 can be edited and updated by B.

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

5) F3 can be edited and updated by C. Note that, here F3 can be edited and updated by B and C cooperatively.

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

7) B assigns E to edit F2 and E can update F2.

Fig. 2 depicts another illustrating example as follows:

1) $FileSet = \{F1, F2, F3, F4\}$. User A (e.g., file creator, manager) uploads four files (i.e., $F1, F2, F3, F4$) into cloud servers. A's privilege for all

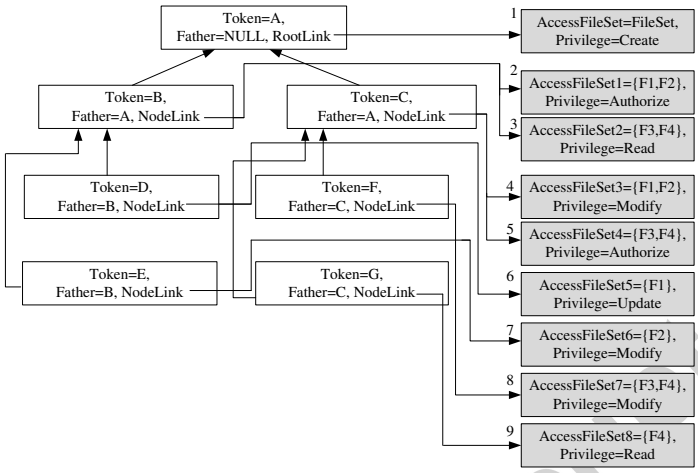


Figure 2: Another Example for Illustrating the Logical Model.

files are *Create*. A's login account information for cloud servers will be used for all other users's login, namely, B, C, D, E, F, G.

2) F1 and F2 are two files that will be edited by a group leaded by B.

3) F3 and F4 can be read by B, but B cannot modify F3 and F4.

4) F1 and F2 can be modified by C for comments, but C cannot update those modification. The modification can be reviewed and updated by B. (C is the leader of another group.)

5) F3 and F4 can be edited by a group leaded by C (e.g., F3 and F4 can be further handed out to others to edit).

6) D is in the group leaded by B. B adds two users into her group, namely, D and E. B assigns D to update F1.

7) B assigns E to modify F2 but not to update F2. The modification on F2 by E can be reviewed by B and updated by B.

8) F and G are added into the group leaded by C. C assigns F to edit F3 and F4, but F cannot update F3 and F4. The modification on F3 and F4 by F will be reviewed by C and updated by C.

9) C assigns G to read F4 and G cannot modify F4.

4.5. Analysis

Following two propositions state about the number of leaves. $|\cdot|$ returns the number of elements in a set.

proposition 4.1. $|Root.RootLink| = 1$.

Proof The number of leaves at *Root* is one. That is, $RootLink = \langle AccessFileSet = FileSet, Privilege = Create\rangle$. □

proposition 4.2. $|Node.NodeLink| \leq |Privilege| - 1 = 4$.

Proof The number of leaves ($NodeLink$) at a node is at most the number of $|Privilege| - 1$. As the leaves are organized by $Privilege$ for different $AccessFileSet$. $|\{Create \| Authorize \| Update \| Modify \| Read\}| = 5$. $Node.NodeLink.Privilege \neq Create$, thus the number of privileges at most is 4. □

Following four propositions state the relations between root and its child (children). The scale of the tree is also quantified for confirming its lightweightness.

proposition 4.3. $\bigcup Node_i.NodeLink.AccessFileSet \subset Root.RootLink.AcccessFileSet$, *where* $Node_i.Father = Root.Token$.

Proof Straightforward. The Root's file will be assigned to all users to access. The accessing files should be in the scope of files that are uploaded by root. In other words, the merging set of $AccessFileSet$ in leaves at a child or all children of *Root* is upper bounded by $FileSet$. □

proposition 4.4. $|Node_i.NodeLink.AccessFileSet| \leq |Root.RootLink.AccessFileSet|$, *where* $Node_i.Father = Root.Token$.

Proof Straightforward. The reason is due to Proposition 4.3. □

proposition 4.5. $|\bigcup Node_i.NodeLink| \leq \min(|Root.RootLink.AccessFileSet| * |Node_i|, 4 * |Node_i|)$ *where* $Node_i.Father = Root.Token$.

Proof Similarly to above Proposition 4.3. If $|AcceessFileSet| < 4$, there are at most $|AcceessFileSet|$ leaves in which each $F_i \in AccessFileSet$ has a privilege. If $|AcceessFileSet| \geq 4$, there are at most 4 leaves in which all 4 distinct privileges are assigned for $F_i \in AccessFileSet$. □

proposition 4.6. $|Node_i| = |\{Token|Node_i.Token \textit{ is added by Root.}\}|$ *where* $Node_i.Father = Root.Token$.

Proof The number of nodes whose father is *Root* equals the number of users that are added by *Root*. □

The number of children of *Root* (namely, the number of users added by file creator) has no limited. It depends on the editing logic of shared files, thus the number of members in a group could be large.

Similarly, following four propositions state the relations between a node and its child (children).

proposition 4.7. $\bigcup Node_i.NodeLink.AccessFileSet \subset \{Node_j.NodeLink.AccessFileSet|Node_j.NodeLink.Privilege = Authorize\}$, *where* $Node_i.Father = Node_j.Token$.

Proof The $Node_j$'s files that have "Authorize" privilege can be assigned to other users to access. In other words, the merging set of $AccessFileSet$ in leaves at a child or all children of a node is upper bounded by $AccessFileSet$ of this node. □

proposition 4.8. $|Node_i.NodeLink.AccessFileSet| \leq |Node_j.NodeLink.AccessFileSet|$, *where* $Node_i.Father = Node_j.Token$.

Proof Straightforward. The reason is due to Proposition 4.7. □

proposition 4.9. $|\bigcup Node_i.NodeLink| \leq \min(|Node_j.NodeLink.AccessFileSet| * |Node_i|, 4 * |Node_i|)$ *where* $Node_i.Father = Node_j.Token$.

Proof Similarly to Proposition 4.5. If $|AcceessFileSet| < 4$, there are at most $|AcceessFileSet|$ leaves in which each $F_i \in AccessFileSet$ has a privilege. If $|AcceessFileSet| \geq 4$, there are at most 4 leaves in which all 4 privileges are assigned for $F_i \in AccessFileSet$. □

proposition 4.10. $|Node_i| = |\{Token|Node_i.Token \textit{ is added by } Node_j.\}|$ *where* $Node_i.Father = Node_j.Token$.

Proof The proof and reason is similar to 4.6. □

Following propositions state relations between $Token$, $UserName$ and $Father$ in UCL. Some of them are straightforward, thus the proof is omitted.

proposition 4.11. *Given $UserName$, relation $UserName \times Token \in UCL$ is one-to-many.*

Proof One creator can create and add more tokens. □

proposition 4.12. *Given $Token$, only one $UserName \in UCL$ satisfies $\langle UserName, Token, * \rangle \in UCL$.*

Note that, here $Token$ is distinct in UCL. No matter in a tree or out of a tree in a forest, different nodes cannot have the same $Token$.

proposition 4.13. $|Node.Father| = 1$. *That is, given $UserName$, relation $Token \times Father \in UCL$ is many-to-one.*

Proof If a $Token$ is created and assigned, it will not be reused by other nodes for assigning. In other words, each node except for root has one father. Thus, the access structure is a tree. □

Similarly, following propositions state relations between $Token$, $FileName$ and $Privilege$ in ACL.

proposition 4.14. *Given $Token$, relation $FileName \times Privilege \in ACL$ is one-to-one.*

Note that, here $FileName$ is a file list in which all files have the same privilege.

proposition 4.15. *Given $FileName$, relation $Token \times Privilege \in ACL$ is many-to-one.*

Note that, here $Token$ is distinct in ACL. Different users cannot have the same $Token$. It can be done easily by let $Token$'s notation includes $UserName$.

```
Data: ACL , UCL, UserName
Result: ACTree
for i = 1; i ≤ |UCL|; i++ do
    if (UCL[i].UserName == UserName).and.(UCL[i].Father ==
     Null) then
        ACTree.Root.Token ⇐ UCL[i].Token;
        ACTree.Root.Father ⇐ Null;
        ACTree.Root.RootLink.Privilege ⇐ Create;
        ACTree.Root.RootLink.AccessFileSet ⇐
         RetrieveACL(ACTree.Root.Token, Create);
    end
    if (UCL[i].UserName == UserName) then
        ACTree.Node.Father ⇐ UCL[i].Father;
        ACTree.Node.Token ⇐ UCL[i].Token;
        for Privilege in {Authorize||Update||Modify||Read} do
            ACTree.Node.NodeLink.Privilege ⇐ Privilege;
            ACTree.Node.NodeLink.AccessFileSet ⇐
             RetrieveACL(ACTree.Node.Token, Privilege);
        end
    end
end
return ACTree;
```

Algorithm 1: Tree Construction $ACTreeCon()$. $|*|$ is the total number of rows in $*$. $RetrieveACL(Token, Privilege)$ returns $FileName$ in ACL with inputting $Token$ and $Privilege$.

4.6. Algorithms

Firstly, we propose an algorithm (called $ACTreeCon()$) to construct access control tree model by ACL and UCL.

proposition 4.16. $Time(ACTreeCon()) = O(|UCL| * |ACL|)$.

Proof In algorithm $ACTreeCon()$, there exists only one loop that has maximal length of $|UCL|$. For each execution in the loop, $RetrieveACL()$ is invoked once. Suppose $RetrieveACL()$ responds in time with $O(|ACL|)$. Thus, timing cost of the algorithm is $O(|UCL| * |ACL|)$. □

Next propositions state the completeness of the proposed model.

proposition 4.17. *$ACTreeCon()$ has minimal timing cost.*

Proof As each record in UCL will be checked at least once to find whether it is on the tree, the minimal timing cost is $|UCL|$. As each node in the tree requires to link to leaves, each record in ACL will be checked at least once to find whether it is the leaf of this node. Thus, total timing cost of $ACTreeCon()$ is at least $O(|UCL| * |ACL|)$. □

proposition 4.18. *ACL and UCL have minimal tuples (or total number of columns).*

Proof In $UCL = \langle UserName, Token, Father \rangle$, $UserName$ is used for distinguishing different trees. $Token$ is used for distinguishing nodes in a tree. Note that, $Token$ can be named distinctly in a tree and also in whole forest. $Father$ is used for constructing edges in a tree. Thus, above three items in UCL are the minimal set for a tree construction.

Furthermore, in $ACL = \langle Token, FileName, Privilege \rangle$, $Token$ is used for localizing a node in a tree. $FileName$ and $Privilege$ are the linked leaves of a node. Thus, above three items are the minimal set for leaf connection that used for access control.

In summary, UCL and ACL only contains minimal tuples for required functionality. □

Discussion

1)One can easily delete a file sharer, revoke the privilege of a user, or remove a user in the group, by simply deleting the corresponding node with

its linked leaves in the tree (i.e., records in UCL with this token and related records in ACL with this token).

2) The deletion of a file sharer can be conducted by this node's father, as the node's father adds this node (i.e., record in UCL with this token) and its leaves in the tree (i.e., grants the node's privileges for accessing files).

3) It can be conducted easily to add a file sharer, grant the privilege of a user, or permit a user join into the group, by adding a node (i.e., appending records into UCL with the corresponding token) with linked leaves (i.e., appending records into ACL with related $FileName$ and $Privilege$ to this token).

4) The addition of a file sharer can be conducted by a user with $Authorize$ or $Create$ privilege. If a user logs in with a token whose privilege of shared files is $Authorize$ or $Create$, the user can add more file sharers into the group leaded by her. That is, if $ACTree.Node.NodeLink.Privilege = Authorize$, the user with $ACTree.Node.Token$ can add more group members for accessing files in $ACTree.Node.NodeLink.AccessFileSet$.

5) If a node to be deleted is a node's father, the deletion will be conducted by replacing a new head of the group. That is, the group head's $Token$ will be deleted. Group members will change $Father$ to the $Token$ of the new group head. New group head may link to old group head's leaves.

6) The modification of user's privilege can be done by modifying the $Privilege$ on the linked leaves of the node. $Update$, $Modify$, and $Read$ can be changed each other among them. The modification may lead to changes of $AccessFileSet$. Besides, $Update$, $Modify$, and $Read$ can be changed into $Authorize$. If it happens, the node can become a node's father by adding more users.

7) If a user's (i.e., a node's linked leaves's) privilege is changed from $Authorize$ to $Update$, $Modify$, or $Read$, it may need to delete the whole subtree whose root is the node for the consistence of the tree and logics.

8) For each file, only one user can have $Create$ privilege. The user who has $Create$ privilege is the file creator who initially uploads files into cloud servers. All users in UCL will share the same account information for logging into cloud servers with this user who has $Create$ privilege.

9) The retrieval of ACL can be looked as the traverse in the tree. After a user logs into cloud servers and presents her token, the node will be determined in the tree. The linked leaves of this node can be fetched and then cloud servers list files that can be accessed and together with corresponding privileges. Also, the node's children can be listed for addition

and removal, if there exists *Authorize* privilege in this node's linked leaves ($Node.NodeLink.Privilege = Authorize$).

10) The accessing file set or files could be encrypted by corresponding token. The $Node(Root).NodeLink(RootLink).AccessFileSet$ links to files that are encrypted by $Node(Root).Token$.

5. Conclusion

In this paper, we proposed a lightweight, proxy-enabled and group-oriented access control scheme (SeGoAC) which supports file editing and sharing in a mobile cloud computing environment. SeGoAC can also support multiple new flexible functionalities, such as user-friendly addition and deletion of group users, self-defined and self-managed access control, iterative access control proxy, and separation of access authentication from system authentication. We described the physical model, abstract model, process model, logical model and key algorithms, as well as demonstrating the the soundness and completeness of SeGoAC via simulations.

Acknowledgement

The research was financially supported by the National Natural Science Foundation of China (61170217, 61310306028, and 61502440), the PAPD fund, and the CICAEET fund.

[1] D. Quick, B. Martini and K.-K. R. Choo, *Cloud Storage Forensics*. Syngress Publishing / Elsevier

[2] C. Perera, R. Ranjan, L. Wang, End-to-End Privacy for Open Big Data Markets. *IEEE Cloud Computing*, 2(4):44-53, 2015

[3] C. Perera, R. Ranjan, L. Wang, SU. Khan and AY. Zomaya, Big Data Privacy in the Internet of Things Era. *IT Professional*, 17(3):32-39, 2015

[4] 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*, 9(6): 903-916, Nov 2012.

[5] M. Habiba, M.R. Islam and A.B.M.S. Ali. Access Control Management for Cloud. *2013 12th IEEE International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom)*, 485-492, July 2013

[6] 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*, 25(2): 384-394, Feb 2014

[7] 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*, 7(2): 743-754, April 2012

[8] P. Ortiz, O. Lazaro, M. Uriarte and M. Carnerero. Enhanced multi-domain access control for secure mobile collaboration through Linked Data cloud in manufacturing. *2013 IEEE 14th International Symposium and Workshops on a World of Wireless, Mobile and Multimedia Networks (WoWMoM)*, 1-9, June 2013.

[9] Z. Lv, C. Hong, M. Zhang and D. Feng. A secure and efficient revocation scheme for fine-grained access control in cloud storage. *2012 IEEE 4th International Conference on Cloud Computing Technology and Science (CloudCom)*, 545-550, Dec 2012.

[10] R. Yang, C. Lin and Y. Jiang. Enforcing scalable and dynamic hierarchical access control in cloud computing. *2012 IEEE International Conference on Communications (ICC)*, 923-927, June 2012.

[11] X. Yao, X. Han and X. Du. A lightweight access control mechanism for mobile cloud computing. *2014 IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*, 380-385, April 2014.

[12] T. Jung, X. 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*, 10(1): 190-199, Jan 2015.

[13] Z. Wang, D. Huang, Y. Zhu, B. Li and C.J. Chung. Efficient Attribute-Based Comparable Data Access Control. *IEEE Transactions on Computers*, 64(12): 3430-3443, 2015

[14] J. Zhao, L. Wang, J. Tao, J. Chen, W. Sun, R. Ranjan, J. Kolodziej, A. Streit, D. Georgakopoulos, A Security Framework in G-Hadoop for Big Ddata Computing across Distributed Cloud Data Centres. *Journal of Computer and System Sciences*, 80(5): 994-1007, 2014

[15] S. Tu and Y. Huang, Towards efficient and secure access control system for mobile cloud computing, *China Communications*, 12(12):43-52, 2015

[16] Y. Jin and C. Tian and H. He and F. Wang, A Secure and Lightweight Data Access Control Scheme for Mobile Cloud Computing, *2015 IEEE Fifth International Conference on Big Data and Cloud Computing (BDCloud'15)*, 172-179, 2015

[17] Z. Lv and J. Chi and M. Zhang and D. Feng, Efficiently Attribute-Based Access Control for Mobile Cloud Storage System, *2014 IEEE 13th International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom'14)*, 292-299, 2014

[18] Y. Xie and H. Wen and B. Wu and Y. Jiang and J. Meng, A Modified Hierarchical Attribute-based Encryption Access Control Method for Mobile Cloud Computing, *IEEE Transactions on Cloud Computing*, PP, 99, 1-1, 2016

[19] Z. Fu, X. Sun, Q. Liu, L. Zhou and J. Shu, Achieving Efficient Cloud Search Services: Multi-keyword Ranked Search over Encrypted Cloud Data Supporting Parallel Computing, *IEICE Transactions on Communications*, vol. E98-B, no. 1, pp. 190-200, 2015.

[20] Y. Ren, J. Shen, J. Wang, J. Han and S. Lee, Mutual Verifiable Provable Data Auditing in Public Cloud Storage, *Journal of Internet Technology*, vol. 16, no. 2, pp. 317-323, 2015.

Table 1: Abstract Model

.

Step 1)	$Alice : Login(Cloud, UserName_a, Password_a)$.
Step 2)	$Cloud : FetchUCL(UserName_a)$,
	If $UserName_a \notin UCL$
	$Alice : SetupToken(\langle UserName_a, Token_a, Father = Null\rangle)$.
	$Cloud : SaveUCL(\langle UserName_a, Token_a, Father = Null\rangle)$.
	go to Step 3).
	If $UserName_a \in UCL$, $Cloud : Token_* \Leftarrow RequestToken()$
	If $Token_* = Token_a$, go to Step 3),
	else go to Step 8).
Step 3)	$Alice : Upload(File_{a1})$.
Step 4.1)	$Cloud : SaveACL(\langle Token_a, FileName_{a1}, Create\rangle)$.
Step 4.2)	$Alice : SetupToken(\langle UserName_a, Token_b, Father = Token_a\rangle)$,
	$Cloud : SaveUCL(\langle UserName_a, Token_b, Father = Token_a\rangle)$.
Step 4.3)	$Alice : SetupACL(\langle Token_b, FileName_{a1}, Read\rangle)$,
	$Cloud : SaveACL(\langle Token_b, FileName_{a1}, Read\rangle)$.
Step 5)	$Alice : SetupToken(\langle UserName_a, Token_c, Father = Token_a\rangle)$,
	$Cloud : SaveUCL(\langle UserName_a, Token_c, Father = Token_a\rangle)$,
	$Alice : SetupACL(\langle Token_c, FileName_{a1}, Modify\rangle)$,
	$Cloud : SaveACL(\langle Token_c, FileName_{a1}, Modify\rangle)$;
	$Alice : SetupToken(\langle UserName_a, Token_d, Father = Token_a\rangle)$,
	$Cloud : SaveUCL(\langle UserName_a, Token_d, Father = Token_a\rangle)$,
	$Alice : SetupACL(\langle Token_d, FileName_{a1}, Update\rangle)$,
	$Cloud : SaveACL(\langle Token_d, FileName_{a1}, Update\rangle)$.
Step 6)	$Alice : Upload(File_{a2})$,
	$Alice : Upload(File_{a3})$,
	Redo 4.1)-4.3), and 5).
Step 7)	Off-line Operations.
	$Alice \rightarrow Bob : \{UserName_a, Password, Token_b\}$,
	$Alice \rightarrow Carolyn : \{UserName_a, Password, Token_c\}$.
Step 8)	$Bob : Login(Cloud, UserName_a, Password_a)$,
	$Cloud : FetchUCL(UserName_a)$,
	If $UserName_a \in UCL$
	$Cloud : Token_* \Leftarrow RequestToken()$.
Step 9)	$Cloud : FetchUCL(UserName_a, Token_*)$,
	If $\{UserName_a, Token_*\} \in UCL$
	$Cloud : \{FileName, Privilige\} \Leftarrow RetrieveACL(Token_*)$.

Table 2: Logical Model

.

$ACTree ::= \langle Root, NodeSet, Edges \rangle$,
$Root ::= \langle Token, Father = NULL, RootLink \rangle$,
$RootLink ::= \langle AccessFileSet = FileSet, Privilege = Create \rangle$,
$FileSet ::= \{FileName_i \vert FileName_i$ are uploaded to CLOUD by Root. $i = 1, ..., n\}$,
$NodeSet ::= \langle Node \rangle \Vert \langle Node \rangle, ..., \langle Node \rangle$
$Node ::= \langle Token, Father, NodeLink \rangle$,
$NodeLink ::= \langle AccessFileSet, Privilege \rangle \Vert$
$\langle AccessFileSet, Privilege \rangle, ..., \langle AccessFileSet, Privilege \rangle$,
$AccessFileSet ::= \{FileName_i \in FileSet, i \in [1, ..., n], \vert FileName_i \vert = 1\} \Vert$
$\{FileName_i \subseteq FileSet, i \in [1, ..., n], \vert FileName_i \vert \geq 2, \}$
$Privilege ::= \{Create \Vert Authorize \Vert Update \Vert Modify \Vert Read\}$,
$Edges ::= \langle Edge \rangle, ..., \langle Edge \rangle$,
$Edge ::= \{\langle Node_1, Node_2 \rangle \vert Node_1.Father = Node_2,$
$Node_1, Node_2 \in NodeSet,\ Node_2.Privilege = \{Create \vert\vert Authorize\}\}$.