A Secure Fine-Grained Access Control Mechanism for Networked Storage Systems

Hsiao-Ying Lin, John Kubiatowicz† and Wen-Guey Tzeng*
Intelligent Information and Communications Research Center, National Chiao Tung University
hsiaoying.lin@gmail.com
†Department of Computer Science, University of California Berkeley
kubitron@cs.berkeley.edu
*Department of Computer Science, National Chiao Tung University
wgtzeng@cs.nctu.edu.tw

***Abstract*—Networked storage systems provide storage services for users over networks. Secure networked storage systems store encrypted data to guarantee data confidentiality. However, using encryption schemes somehow restricts the access control function over stored data. We address the access control function for a secure networked storage system by proposing a fine-grained access control mechanism. In our mechanism, a user cannot only read or write data but also grant the reading permissions of a single file or a whole directory of files to others with low cost. Moreover, these functions are supported in a confidential way against honest-but-curious storage servers. Our technical contribution is to propose a hybrid encryption scheme for a typical structure of a file system by integrating a hierarchical proxy re-encryption scheme and a hierarchical key assignment scheme. We measure the computation overhead for reading, writing, and granting operations by experiments. Our experimental results show that getting a finer access control mechanism does not cost much.**

***Index Terms*—Networked storage system; access control mechanism; proxy re-encryption; hybrid encryption;**

I. Introduction

Networked storage systems, such as OceanStore [1], PAST [2], Plutus [3] and Tahoe [4], provide storage services via the Internet. Users can store their data in a networked storage system and access them whenever they need. The stored data are often protected by some encryption scheme for data confidentiality. Data encryption somewhat limits the functionalities of the networked storage system. For example, the owner of the encrypted file cannot forward it to another user without retrieving and decrypting first. Access control is one of fundamental functions of a networked storage system. In this paper, we focus on this particular functionality and investigate efficient ways for a user to grant reading permissions to others in a secure networked storage system.

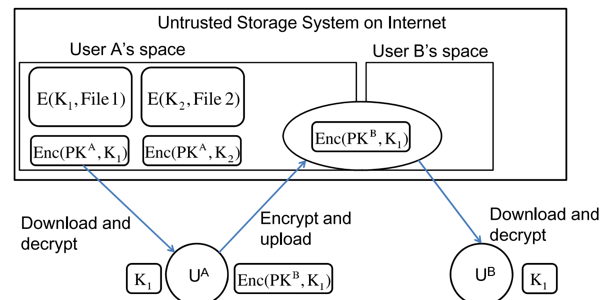

Fig. 1. This figure shows a straightforward solution to a secure access control mechanism in a networked storage system. There are two users U^{A} and U^{B}. File 1 is encrypted by the content key K_1 and the content key is encrypted by the public key PK_A of user U^{A}, where E is a symmetric encryption algorithm and Enc is an asymmetric encryption algorithm. The owner U^{A} wants to grant the reading permission of File 1 to another user U^{B}. U^{A} first downloads the encrypted content key, decrypts it, and then encrypts it again for U^{B}. Afterward, U^{B} can download and decrypt the file.

A straightforward solution is shown in Fig. 1. File 1 is first symmetrically encrypted by a content key K_1. The content key K_1 is then asymmetrically encrypted by the public key PK^{A} of user U^{A}. Because a symmetric encryption scheme is more efficient than an asymmetric one while an asymmetric encryption scheme often is provable secure, a hybrid encryption approach is a common practice for achieving strong security and efficiency. The entire granting process requires three communication rounds and expensive computation costs of user U^{A} (the data owner).

A better approach is to let the underlying encryption scheme be a proxy re-encryption scheme. Each user in the scheme has a key pair of public key and secret key. Given a proxy key from U^{A} to U^{B}, a proxy server

978-0-7695-4742-8/12 $26.00 © 2012 IEEE
DOI 10.1109/SERE.2012.35

IEEE computer society

can transfer a ciphertext under user U^{A}'s public key to another ciphertext under user U^{B}'s public key. In the networked storage system, the storage server plays the role of the proxy server. Therefore, user U^{A} can grant the reading permission to another user U^{B} by giving the proxy key to the storage server while the writing permission is granted by publishing the public key. The process requires only two communication rounds and the computation cost of U^{A} is reduced. However, the granting pattern is in an "all or nothing" manner. For example, once the proxy server has the proxy key from U^{A} to U^{B}, all U^{A}'s ciphertexts can be re-encrypted to ciphertexts of U^{B}. To improve the granularity of the granting pattern, treating a file as an independent entity and assigning independent key pair to each file is a possible solution. To grant a specific file, the owner generates the proxy key from the file to the granted user and sends the key to the server. But, in this method, the amount of public keys and secret keys of a user would be extremely huge. One more problem in this solution is that when a user wants to grant t files, he needs to generate and send t proxy keys to the server.

It is challenging to efficiently support fine-grained access control in a networked storage system with low storage overhead when stored data are encrypted. To address this issue, we customize the "hierarchical proxy re-encryption (HIPRE)" scheme and integrate it with a conventional hierarchical key assignment scheme to construct an access control mechanism. In our access control mechanism, a user only stores a constant number of keys (a secret key, a master key, and a root content key) and he can grant the reading permission of a selected single file or a selected directory of files by a single proxy key. Our access control mechanism balances the storage overhead and the granularity of the granting pattern.

In our scenario, we use the hybrid encryption approach in the networked storage system. Our key idea for reducing the storage overhead for keys of a user is to systematically generate a key pair for each file from the keys that the user already has. Since each file has a key pair, the granularity of the granting pattern is obtained. To support the granting pattern "a directory of files", we use the hierarchical key assignment scheme to generate content keys for files and directories such that the root content key of the user can derive all content keys and a content key of a directory can derive the content key of a file in the directory.

To evaluate the performance of our access control mechanism, we implement our hierarchical proxy re-encryption scheme and conduct experiments. We compare our mechanism and another mechanism using a proxy re-encryption scheme in terms of reading and writing speeds. The experimental results show that our access control mechanism only causes a small constant computation overhead for a writing operation when comparing with a plain proxy re-encryption solution. The overhead is independent of the size of the file.

II. Related Work

At the early years, the data confidentiality issue is addressed to defend against external attackers. The storage servers are fully trusted by users. In the storage systems such as Blaze's CFS [5], TCFS [6], and NCryptfs [7], storage servers encrypt files when receiving them from users and decrypt files before sending them to users. As a result, storage servers handle the access control and the access control may depend on additional authentication schemes.

Many distributed networked storage systems, such as Oceanstore [1], Plutus [3], and Tahoe [4], decrease the trust on the storage servers. A user has to encrypt his file by a content key before uploading to storage servers. To grant the reading permission to other user, the owner has to send the content key to the granted user by himself. Thus, the owner has fully control over his own data and has to manage the granting processes.

Ateniese et al. [8] proposed proxy re-encryption schemes and showed their application on the networked storage system. By using proxy re-encryption schemes, a user can easily grant reading permission of his encrypted files to other users. Identity-based proxy re-encryption schemes [9], [10] use identities of users as their pubic keys. However, as we discussed in Section I, the granting pattern is "all or nothing". Tang [11] proposed a type-based proxy re-encryption scheme, which supports type-based granting patterns. Attribute-based proxy re-encryption schemes support attribute-based granting patterns [12], [13]. Types or attributes associated with data are determined when the data are encrypted. Moreover, the length of ciphertexts or proxy keys depends on the number of associated attributes.

III. System Model and Security Requirements

A. System Model

In the networked storage system, we assume that each user has his own space to store files. Files and directories in the space are organized as a hierarchy.

A *hierarchy* Γ consists of classes and is modeled as a directed acyclic graph, where classes are vertices. A

class C_i is said to be higher than another class C_j if and only if there is a path from C_i to C_j in the graph. We consider tree-like hierarchies, i.e., each class in the hierarchy has at most one parent class. Each file or an empty directory is represented as a leaf-class and each non-empty directory is represented as an internal class in the hierarchy. When a directory contains a file, there is an edge from the class of the directory to the class of the file. Define the level of a class in the hierarchy as the number of edges from the root class to the class. The level of the hierarchy is defined as the maximum level among all classes in the hierarchy. Fig. 3 contains an example of 3 levels: level 0, level 1, and level 2.

A user U^{A} has a public key PK^{A}, a secret key SK^{A} and a master key MK^{A} of a proxy re-encryption scheme. U^{A} also has a root content key K_0. For creating a directory or a file, U^{A} needs to generate a key pair, encryption key and decryption key, from SK^{A}. U^{A} also needs to generate a content key for the new directory or file from the content key of the parent directory. The keys of the root class of A's hierarchy are the encryption key $\mathsf{EK}_0^{\mathsf{A}}$, the decryption key $\mathsf{DK}_0^{\mathsf{A}}$ and the root content key K_0, where $\mathsf{DK}_0^{\mathsf{A}} = \mathsf{SK}^{\mathsf{A}}$.

Denote the encryption and decryption algorithms of a symmetric encryption scheme as $\{\mathsf{E}, \mathsf{D}\}$. When user U^{A} wants to store a file F into the system, U^{A} divides file F into blocks $f_1, f_2, \cdots, f_n$ with the defined block size, where n is the number of those blocks. Let the file F be represented as a class C_i in the hierarchy. Then U^{A} generates an encryption key $\mathsf{EK}_i^{\mathsf{A}}$ and a decryption key $\mathsf{DK}_i^{\mathsf{A}}$, from SK^{A}. U^{A} also generates a content key K_i from the content key K_0. U^{A} symmetrically encrypt each file block by K_i. The ciphertexts are denoted as $\mathsf{E}(K_i, f_1), \mathsf{E}(K_i, f_2), \cdots, \mathsf{E}(K_i, f_n)$ [1]. U^{A} encrypts the content K_i by using the encryption key $\mathsf{EK}_i^{\mathsf{A}}$ of the file. The file F is stored as follows

$$\mathsf{Enc}(\mathsf{EK}_i^{\mathsf{A}}, K_i), \mathsf{E}(K_i, f_1), \mathsf{E}(K_i, f_2), \cdots, \mathsf{E}(K_i, f_n)$$

Here $\mathsf{Enc}(\mathsf{EK}_i^{\mathsf{A}}, K_i)$ denotes an encryption of K_i by the encryption key $\mathsf{EK}_i^{\mathsf{A}}$. Note that each file is encrypted by a different content key.

Illustrating in Fig. 2, the proxy re-encryption operation supports the functionality of granting reading permissions in the system. To grant the reading permission of file F to another user U^{B}, user U^{A} generates the proxy key $\mathsf{PRK}_i^{A \to B}$ and gives it to the storage server. Here we assume that the communication channel between the user and the storage server is secure such that no one can get the proxy key by eavesdropping. The storage server uses the proxy key to re-encrypt the ciphertext $\mathsf{Enc}(\mathsf{EK}_i^{\mathsf{A}}, K_i)$ of U^{A} to a ciphertext $\mathsf{Enc}(\mathsf{EK}_0^{\mathsf{B}}, K_i)$ of U^{B}. Thus, the user U^{B} can decrypt for the content key K_i and read the file. Similarly, U^{A} can grant a directory of files to U^{B}. Let K_j be the content key of the directory. Once U^{B} gets K_j, he can derive content keys of files under the directory.

[1]For example, the symmetric encryption scheme may be 3DES or AES.

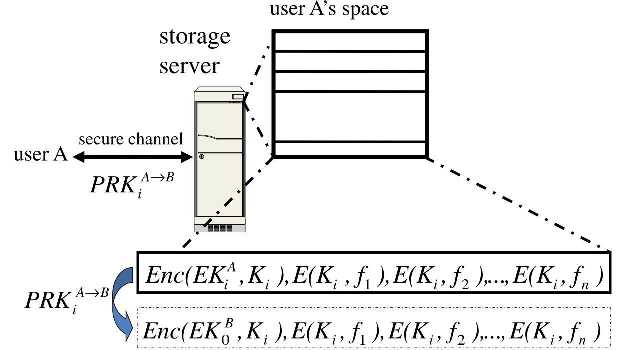

Fig. 2. *Storage System Model*. The storage server can re-encrypt $\mathsf{Enc}(\mathsf{EK}_i^A, K_i)$ by a proxy key $\mathsf{PRK}^{(A \to B)_i}$ to $\mathsf{Enc}(\mathsf{EK}_0^B, K_i)$ then U^{B} can get the content key K_i.

B. Security Requirements

The security requirements of the networked storage system are the confidentiality of the stored files and the security of the access control mechanism. For the confidentiality requirement, a malicious user cannot read another user's file without the reading permission. The malicious user may have many reading permissions of other files of the same user. We expect that the malicious user cannot get the content key of the target file. For the security of the access control mechanism, the storage server cannot grant any reading permission without the owner's authorization even if the storage server colludes with other users. We capture these two security requirements in the security games $\mathcal{G}_1$ and $\mathcal{G}_2$.

In the security game $\mathcal{G}_1$ of the confidentiality, we model the scenario that the malicious user chooses a target file of a target user U^t and has all reading permissions of files that are not located in the path from the root to the target file. As a result, the malicious user has all content keys of those files but none of those content key can derive the content key of the target file. We also allow the malicious user accessing the encryption of the target content key stored in the storage server. The goal of the malicious user is to compute the

content key of the target file. The security game $\mathcal{G}_1$ is described as follows:

- *Setup.* The challenger $\mathcal{C}$ generates the public parameters and gives them to the adversary $\mathcal{A}$. The adversary $\mathcal{A}$ acting as another legal user $U^{\mathcal{A}}$ sets his keys $\mathsf{MK}^{\mathcal{A}}$, $\mathsf{SK}^{\mathcal{A}}$ and $\mathsf{PK}^{\mathcal{A}}$ and sends $\mathsf{PK}^{\mathcal{A}}$ to $\mathcal{C}$. $\mathcal{C}$ generates all keys MK, SK and PK for each user except $\mathcal{A}$. $\mathcal{C}$ then sends all PK^i to $\mathcal{A}$.
- *Challenge.* $\mathcal{A}$ chooses a target user U^t, a hierarchy Γ^t for the user U^t, and a target class $C_{t'} \in \Gamma^t$.
- *Response.* For the indicated hierarchy Γ^t, $\mathcal{C}$ generates the content keys. Let the content key of the target file be K. $\mathcal{C}$ considers a subset of the content keys that cannot derive the content key K and encrypts those content keys under the encryption key of the root class of $\mathcal{A}$ (in the format of $(\mathbb{G}_2, \mathbb{G}_2)$). $\mathcal{C}$ also encrypts the target content key under the encryption key $\mathsf{EK}^t_{t'}$. $\mathcal{C}$ sends all those encrypted content keys to $\mathcal{A}$.
- *Output.* Finally, $\mathcal{A}$ outputs K^* for guessing the content key K of the target file.

$\mathcal{A}$ wins the game if $K^* = K$ and the advantage of $\mathcal{A}$ is defined as $\Pr[K^* = K]$.

In the security game $\mathcal{G}_2$ of the security of the access control mechanism, we model the scenario that the storage server chooses a target file of a target user U^t and tries to generate a proxy key from this target file to another user U without the permission of U^t. Moreover, the storage server colludes with the user U. Consider the path from the root to the target file in the hierarchy. The storage server has all proxy keys excluding the proxy keys belongs to the path. Finally, $\mathcal{A}$ outputs a value PRK^* as a guessing for the proxy key of the target file. $\mathcal{A}$ wins if he correctly guesses the proxy key and the advantage of $\mathcal{A}$ is defined as the probability that $\mathcal{A}$ wins.

The security game $\mathcal{G}_2$ is described as follows:

- *Setup* is the same as *Setup* in $\mathcal{G}_1$.
- *Challenge.* $\mathcal{A}$ chooses the target user U^t, a hierarchy Γ^t for the user U^t, and the target class $C_{t'} \in \Gamma^t$.
- *Response.* For the indicated hierarchy Γ^t, $\mathcal{C}$ generates the proxy keys $\mathsf{PRK}_j^{t \to \mathcal{A}}$ for all classes C_j in Γ^t except those in the path from the root to $C_{t'}$. Let the target proxy key be $\mathsf{PRK}_{t'}^{t \to \mathcal{A}}$.
- *Output.* Finally, $\mathcal{A}$ outputs PRK^* for guessing the target proxy key $\mathsf{PRK}_{t'}^{t \to i}$.

$\mathcal{A}$ wins the game if $\mathsf{PRK}^* = \mathsf{PRK}_{t'}^{t \to i}$ and the advantage of $\mathcal{A}$ is defined as $\Pr[\mathcal{A} \text{ wins}]$.

Definition 1. Secure Networked Storage System
A networked storage system is secure if no polynomial-time algorithm wins $\mathcal{G}_1$ and $\mathcal{G}_2$ with non-negligible advantage.

IV. Building Blocks

We introduce the hierarchical key assignment scheme and present our hierarchical proxy re-encryption scheme.

A *hierarchical key assignment scheme* generates a key for each class in a given hierarchy [14]–[17]. The keys assigned to those classes have the property that the key of a class C_i can derive the keys of all classes lower than C_i. The security requirement of the hierarchical key assignment scheme is that for any set S of classes, they cannot compute the key of any class not lower than any class in S. We use a basic construction proposed in [14]. It uses a one-way hash function $H : \{0,1\}^z \times \{0,1\}^l \to \{0,1\}^l$ to derive the keys from top to bottom in the hierarchy. A key K_0 is randomly selected from $\{0,1\}^l$ and set as the key of the root class. For the direct children class C_i of the root class, the key of the class C_i is generated by $K_i = H(C_i, K_0)$. Recursively, all classes have their own keys. Fig. 3 shows an example of the key assignment scheme. This key assignment scheme is secure assuming that the hash function is one-way.

A. Hierarchical Proxy Re-Encryption

We briefly describe the algebraic notations and describe our HIPRE construction in details.

1) Algebraic Setting: Let $x \in_R X$ denote that x is randomly chosen from the set X.

Bilinear Map. Let $\mathbb{G}_1$ and $\mathbb{G}_2$ be multiplicative cyclic groups of prime order q. Let $g \in \mathbb{G}_1$ be a generator. A map $\tilde{e} : \mathbb{G}_1 \times \mathbb{G}_1 \to \mathbb{G}_2$ is a bilinear map if it has bilinearity and non-degeneracy properties, that is:

- $\forall a, b \in Z_q$, $\forall h \in \mathbb{G}_1$, $\tilde{e}(g^a, h^b) = \tilde{e}(g, h)^{ab}$.
- $\tilde{e}(g, g)$ is not the identity element in $\mathbb{G}_2$

Discrete Logarithm Assumption. For a multiplicative cyclic group with prime order q and a generator g, it is assumed infeasible to find a from given g^a. Here the discrete log assumption holds for both $\mathbb{G}_1$ and $\mathbb{G}_2$.

Decisional Bilinear Diffie-Hellman Assumption. Following the above parameters, given g, g^x, g^y, g^z where x, y, $z \in_R \mathbb{Z}_q$, the decisional bilinear Diffie-Hellman problem is to distinguish $\tilde{e}(g, g)^{xyz}$ from a random element from $\mathbb{G}_2$. Formally, for any probabilistic polynomial time algorithm $\mathcal{A}$, the following is negligible:

$$|\Pr[\mathcal{A}(g, g^x, g^y, g^z, \mathbb{Q}_b) = b : x, y, z, r \in_R \mathbb{Z}_p; \quad \mathbb{Q}_0 = \tilde{e}(g, g)^{xyz}; \mathbb{Q}_1 = \tilde{e}(g, g)^r; b \in_R \{0, 1\}] - 1/2|.$$

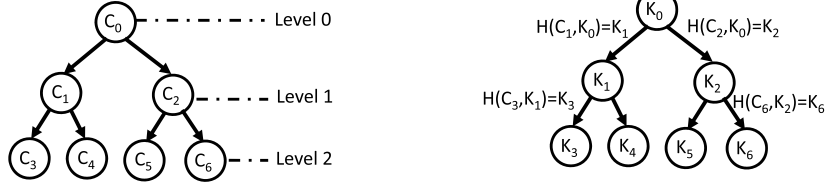

Fig. 3. The left graph is a hierarchy with 3 levels. The right graph is an example of a key assignment scheme for the hierarchy. Each class C_i has a key K_i. The root class's key K_0 can derive all keys in the hierarchy via the hash function H.

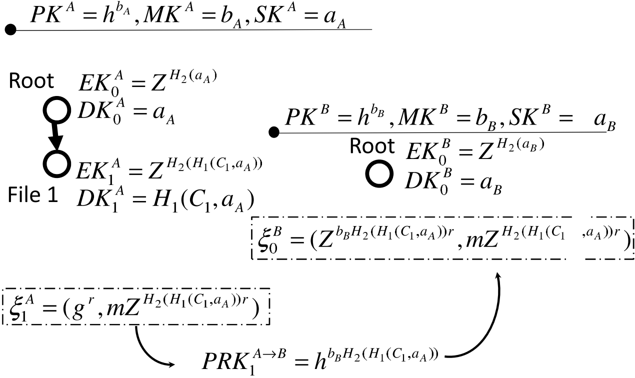

Fig. 4. *Construction example*. User U^{A} has keys $(\mathsf{PK}^A, \mathsf{MK}^A, \mathsf{SK}^A)$ and the directory. With SK^A, the encryption keys and decryption keys are generated. With the proxy key $\mathsf{PRK}_1^{A\to B}$, the content key m of File 1 of U^{A} is re-encrypted to user U^{B}.

2) Our HIPRE Construction: Denote the HIPRE scheme as $\Pi = \{$Setup, KeyGen, HieKeyGen, ProxyKeyGen, Enc, ReEnc, Dec$\}$. In addition, illustrated in Fig. 4, an example is shown for a re-encryption operation from the content key m of user U^{A}'s File 1 to user U^{B}.

- $\mathsf{SetUp}(1^\tau)$ generates system parameters $\mu = \{\tilde{e}, \mathbb{G}_1, \mathbb{G}_2, q, g, h, Z\}$, where $\tilde{e} : \mathbb{G}_1 \times \mathbb{G}_1 \to \mathbb{G}_2$ is a bilinear map, and $Z = \tilde{e}(g, h)$.
- $\mathsf{KeyGen}(\mu)$ generates a public key PK^i, a master key MK^i, and a secret key SK^i for each user U^i.
$$\mathsf{PK}^i = h^{b_i}, \mathsf{MK}^i = b_i, \mathsf{SK}^i = a_i,$$
where $a_i \in \{0,1\}^l, b_i \in Z_q$
- $\mathsf{HieKeyGen}(\mathsf{SK}^i = a_i, \Gamma^i)$ generates a key pair for each class in the hierarchy Γ^i via two collision-resistant hash functions H_1 and H_2, where $H_1 : \{0,1\}^z \times \{0,1\}^l \to \{0,1\}^l$ and $H_2 : \{0,1\}^l \to Z_q$. The key pair of the root class C_0 is
$$\mathsf{DK}_0^i = a_i, \mathsf{EK}_0^i = Z^{H_2(a_i)}.$$
For a child class C_j of the root class,
$$\mathsf{DK}_j^i = H_1(C_j, a_i), \mathsf{EK}_j^i = Z^{H_2(H_1(C_j, a_i))}.$$
For a child class C_j of any class C_w, $w \neq 0$, with $\mathsf{DK}_w^i = v$, $\mathsf{DK}_j^i = H_1(C_j^i, v)$ and $\mathsf{EK}_j^i = Z^{H_2(H_1(C_j^i, v))}$. The key generation continues recursively until every class has a key pair.
- $\mathsf{ProxyKeyGen}(\mathsf{DK}_j^i, \mathsf{PK}^k)$ generates the proxy key from the class C_j of the user U^i to user U^k. Let the input $\mathsf{DK}_j^i = v$ and $\mathsf{PK}^k = u$. The proxy key is computed as $\mathsf{PRK}_j^{i\to k} = u^{H_2(v)}$.
- $\mathsf{Enc}(\mathsf{EK}_j^i, m)$ generates a ciphertext ζ_j^i for the class C_j of user U^i. Let $r \in_R Z_q$ and $\mathsf{EK}_j^i = u$. The ciphertext is computed as $\zeta_j^i = (g^r, mu^r)$.
- $\mathsf{ReEnc}(\zeta_j^i, \mathsf{PRK}_j^{i\to k})$ re-encrypts the ciphertext ζ_j^i to another ciphertext under user U^k's key via the proxy key $\mathsf{PRK}_j^{i\to k}$. Let ζ_j^i be decomposed to (c_1, c_2). The new ciphertext ζ_0^k is computed in the following: $\zeta_0^k = (\tilde{e}(c_1, \mathsf{PRK}_j^{i\to k}), c_2)$.
- $\mathsf{Dec}_1(\zeta_j^i, \mathsf{DK}_j^i)$ decrypts a ciphertext ζ_j^i via the decryption key DK_j^i. Let ζ_j^i be decomposed to (c_1, c_2) and $\mathsf{DK}_j^i = v_1$. The message $m = c_1/\tilde{e}(c_1, h)^{H_2(v_1)}$ by $\mathsf{DK}_j^i = v_1$.
- $\mathsf{Dec}_2(\zeta_0^i, \mathsf{MK}^i)$ decrypts a re-encrypted ciphetext ζ_0^i via the master key MK^i. Let ζ_0^i be decomposed to (c_1, c_2) and $\mathsf{MK}^i = v_2$. The message $m = c_2/c_1^{1/v_2}$.

Remark. In our HIPRE, the ciphertexts have two formats. The first format $(c_1, c_2) \in (\mathbb{G}_1, \mathbb{G}_2)$ can be re-encrypted while the other one $(c_1, c_2) \in (\mathbb{G}_2, \mathbb{G}_2)$ cannot. A user can encrypt a message m in either formats via the encryption key.

The *correctness* of HIPRE is twofold. One is that a ciphertext can be correctly decrypted, i.e.,
$$\mathsf{Dec}_1(\mathsf{Enc}(\mathsf{EK}_j^i, m), \mathsf{DK}_j^i) = m.$$
The other one is that a re-encrypted ciphertext can be correctly decrypted, i.e.,
$$\mathsf{Dec}_2(\mathsf{ReEnc}(\mathsf{Enc}(\mathsf{EK}_j^i, m), \mathsf{PRK}_j^{i\to k}), \mathsf{MK}^k) = m.$$

The correctness can be verified as follows. For each class C_j of each U^i with $\mathsf{EK}^i_j = u$ and $\mathsf{DK}^i_j = v$, the ciphertext $\zeta^i_j = (c_1, c_2)$ can be correctly decrypted:

$$\begin{aligned}\mathsf{Dec}_1(\zeta^i_j, \mathsf{DK}^i_j) &= c_2/\tilde{e}(c_1, h)^{H_2(v)} = m(u)^r/\tilde{e}(g^r, h)^{H_2(v)} \\ &= (mZ^{rH_2(v)})/\tilde{e}(g^r, h)^{H_2(v)} = m\end{aligned}$$

Let $\mathsf{EK}^k_0 = u_2$, $\mathsf{DK}^k_0 = v_1$, $\mathsf{MK}^k = v_2$. The re-encrypted ciphertext $\zeta^k_0 = (c'_1, c_2)$ from ζ^i_j can be correctly decrypted by the masker key MK^k of U^k :

$$\begin{aligned}\mathsf{Dec}_2(\zeta^k_0, \mathsf{MK}^k) &= c_2/{c'}_1^{1/v_2} = m(u)^r/\tilde{e}(g^r, u_2^{H_2(v)})^{1/v_2} \\ &= mZ^{H_2(v)r}/(Z^{rv_2H_2(v)})^{1/v_2} = m\end{aligned}$$

User storage. In this construction, each user U^i needs only to store the master key MK^i and the secret key SK^i. The key pairs of classes in the hierarchy, and the public key PK^i can be derived from the master key MK^i and the secret key SK^i.

V. Access Control over Networked Storage System

We provide an access control mechanism for the networked storage system via our HIPRE construction. Each user's file subsystem is modeled as a hierarchy where the identity of each class is the corresponding file name or directory name.

A. The Networked Storage System

Described as follows, a networked storage system is proposed by using our HIPRE Π, one symmetric encryption (E, D), and the hierarchical key assignment scheme described in Section IV.

System Setup. The system runs $\mathsf{Setup}(1^\tau)$ to generate public parameters and KeyGen to generate the keys $(\mathsf{MK}^i, \mathsf{SK}^i)$ and PK^i for each user U^i.

File Storage. A user U^i's file subsystem is modeled as a hierarchy Γ^i. Each class in the hierarchy represents a file or a directory. For instance, a class C_j represents a directory F and a class C_k represents a file f. Moreover, C_j is the parent class of C_k while the file f is in the directory F.

U^i runs $\mathsf{HieKeyGen}(\mathsf{SK}^i, \Gamma^i)$ for the hierarchy Γ^i. As a result, each file or each directory in the hierarchy Γ^i has its corresponding key pairs. U^i selects a root content key K_0 for the root in Γ^i and chooses a hash function H. U^i uses the hierarchical key assignment scheme to generate the rest content keys via the chosen hash function H. Those content keys form a hierarchy parallel to the key pair hierarchy generated by $\mathsf{HieKeyGen}$. The example in Fig. 5 shows the decryption key structure and the hierarchy of content keys.

- A directory F is stored as $\mathsf{Enc}(\mathsf{EK}^i_j, k_F)$, where k_F is the content key.
- A file f in a directory F is stored as $\mathsf{Enc}(\mathsf{EK}^i_k, k_f)$, $\mathsf{E}(k_f, f_1)$, $\mathsf{E}(k_f, f_2)$, $\cdots$, $\mathsf{E}(k_f, f_n)$, where $(f_1, \cdots, f_n)$ are the file blocks of the file f.

Following the example in Fig. 5, the directory of user U^i is stored as the table in Fig. 6.

B. Access Control Mechanism

After U^i stores his file subsystem into the storage system, he can read files or continue writing new files or directories into the system. Moreover he can grant reading permissions of a file or all files in a directory to other users. These operations are described as follows.

Granting Reading Permission. When U^i wants to grant the reading permission to another user U^x, the granting pattern can be a file or a directory of files by a single proxy key. In the example in Fig. 6, U^i grants the reading permission of File 3 and Directory 2 to U^y and U^x, respectively.

- To grant the reading permission of File 3 to U^y, U^i computes and sends the proxy key $\mathsf{PRK}^{(i\to y)}_3$ to the storage server.
- Similarly, U^i can grant the reading permission of files in Directory 2 to U^x by generating the proxy key $\mathsf{PRK}^{(i\to x)}_2$.

File Reading. To read a file f, the content key k_f is required. There three cases for file reading. First, the owner reads his file f. Second, a user U^y has the reading permission of the file f. Third, a user U^x has the reading permission of a directory F that contains the file f. In all cases, the content key k_f can be derived.

- The owner U^i derives the content key k_f from the root content key K_0 by using the hash function H.
- When a user U^y has the reading permission of the file f, there would be a ciphertext $\mathsf{Enc}(\mathsf{EK}^y_0, k_f)$ in the system. U^y can decrypt the ciphterext $\mathsf{Enc}(\mathsf{EK}^y_0, k_f)$ and get k_f. Note that the storage server computes $\mathsf{Enc}(\mathsf{EK}^y_0, k_f)$ from $\mathsf{Enc}(\mathsf{EK}^i_j, k_f)$ and $\mathsf{PRK}^{(i\to y)}_j$, where C_j is the identity of the file f.
- When a user U^x has the reading permission of a directory F containing the file f, there would be a ciphterext $\mathsf{Enc}(\mathsf{EK}^x_0, k_F)$ in the system. U^x can get k_F from $\mathsf{Enc}(\mathsf{EK}^x_0, k_F)$. To read the file f in the directory F, U^x derives the content key k_f by k_F and the hash function H. In the example in Fig. 6,

U^x can get K_2 and derive $K_3 = H(C_3, K_2)$ and $K_4 = H(C_4, K_2)$ to read File 3 and File 4.

C. Supporting Writing Permission Granting

While the granularity of granting reading permissions is flexible in our access control mechanism, the pattern of granting writing permissions is restricted. A user can only grant the writing permission of files in a directory to others, not including files in any sub-directory. Also, the stored structure of a file which is written by a granted user is different from a file written by the owner.

Granting Writing Permission. U^i can grant the writing permission of a single directory F to another user U^z by giving the encryption key EK^i_j of the directory, where C_j is the identity of the directory F. After U^z gets the encryption key EK^i_j of the directory, he can add a new file f' in the directory F. To do so, U^z randomly chooses a content key $k_{f'}$ and encrypts f' by $k_{f'}$. Then U^z encrypts the content key $k_{f'}$ by EK^i_j. The new file f' is stored as $\mathsf{Enc}(\mathsf{EK}^i_j, k_{f'}), E(k_{f'}, f')$. Consider the example in Fig.6 and the granted directory is Directory 2. The new entry of the new file f' is illustrated in Fig.7.

To read the new file f', U^i decrypts $\mathsf{Enc}(\mathsf{EK}^i_j, k_{f'})$ by using DK^i_j and then decrypts $E(k_{f'}, f')$ by using $k_{f'}$. Another user U^x with the read permission of the directory F can also read the files in the directory, since the storage server can transfer $\mathsf{Enc}(\mathsf{EK}^i_j, k_{f'})$ to $\mathsf{Enc}(\mathsf{EK}^x_0, k_{f'})$ by the existent proxy key $\mathsf{PRK}^{(i\to x)}_j$. Thus, U^x can get the content key $k_{f'}$ as well.

Since the granted user with an encryption key cannot derive any other encryption keys of subdirectories, he can only add files directly in the directory. To make this pattern more flexible, the encryption key structure of the HIPRE must have the derivable property, i.e., the EK^i_j can derive EK^i_k if C_j is higher than C_k in Γ^i.

VI. Security Analysis

We show that our storage system meets the two security requirements by Lemma 1 and Lemma 2, respectively.

Lemma 1. *Assume that the decisional bilinear Diffie-Hellman problem is hard over the bilinear map and the hierarchical key assignment scheme is secure. Our networked storage system meets the confidentiality requirement.*

Proof: We prove by contradiction. If there exists an adversary $\mathcal{A}$ winning the game $\mathcal{G}_1$ with advantage ϵ_0 and there exists an adversary $\mathcal{B}$ breaking the security of the used hierarchical key assignment scheme with probability ϵ_1, there exists an adversary $\mathcal{A}'$ solving the decisional bilinear Diffie-Hellman problem with advantage $(\epsilon_0 - \epsilon_1)/4$.

The input of $\mathcal{A}'$ is $(\tilde{e}, \mathbb{G}_1, \mathbb{G}_2, q, g, g^x, g^y, g^z, \mathbb{Q})$, where $g \in \mathbb{G}_1$ is a generator of $\mathbb{G}_1$ with prime order q. The goal of $\mathcal{A}'$ is to distinguish whether $\mathbb{Q} = \tilde{e}(g, g)^{xyz}$. $\mathcal{A}'$ simulates the environment of $\mathcal{A}$ and uses $\mathcal{A}$ as a subroutine.

- *Setup.* $\mathcal{A}'$ sets the system parameter $\mu = \{\tilde{e}, \mathbb{G}_1, \mathbb{G}_2, q, g, h, Z\}$, where $h = g^x$ and $Z = \tilde{e}(g, g)^x$. $\mathcal{A}$ generates $\mathsf{PK}^{\mathcal{A}}$ and sends to $\mathcal{A}'$. $\mathcal{A}'$ sets $\mathsf{PK}^i = g^{xb_i}$, where $b_i \in_R Z_q$. $\mathcal{A}'$ then sends all PK^i to $\mathcal{A}$.
- *Challenge.* $\mathcal{A}$ chooses a target user U^t, a hierarchy Γ^t for the user, and the target class $C_{t'} \in \Gamma^t$.
- *Response.* $\mathcal{A}'$ first generates all content keys for the hierarchy Γ^t. Let the target content key be K. Let $\mathsf{PK}^t = h^{b_t}, \mathsf{MK}^t = b_t$. $\mathcal{A}'$ considers the subset of content keys that cannot derive K and encrypts those content keys in the re-encrypted form by using $\mathsf{PK}^{\mathcal{A}}$ and maintaining a hash list for H_2. Let K' be a content key for a class C_i. The encryption of the content key is $(\tilde{e}(g, \mathsf{PK}^{\mathcal{A}})^{rv}, K' Z^{rv})$, where $r, v \in_R Z_q$. $\mathcal{A}'$ records v as $H_2(\mathsf{DK}^t_i)$, where DK^t_i is implicitly set. $\mathcal{A}'$ implicitly sets the decryption key $\mathsf{DK}^t_{t'} = H_2^{-1}(y)$ by setting the encryption key $\mathsf{EK}^t_{t'} = \tilde{e}(g^x, g^y) = Z^y$. Then $\mathcal{A}'$ sets the ciphertext ζ^* of the target content key K as $\zeta^* = (g^{zr'}, K\mathbb{Q}^{r'})$. $\mathcal{A}'$ sends all those encrypted content keys to $\mathcal{A}$.
- *Output.* $\mathcal{A}$ outputs K^*. If $K^* = K$, $\mathcal{A}'$ considers $\mathbb{Q} = \tilde{e}(g, g)^{xyz}$ and outputs 0. Otherwise, $\mathcal{A}'$ outputs an uniformly random bit b.

Observe that when $\mathbb{Q} = \tilde{e}(g, g)^{xyz}$, ζ^* is a valid ciphertext for K. In this case, $\mathcal{A}$ has an advantage ϵ to correctly output K. When $\mathbb{Q} = \tilde{e}(g, g)^r$, the distribution of ζ^* is uniformly random over $\{(u, v) | u \in \mathbb{G}_1, v \in \mathbb{G}_2\}$. If $K^* = K$, it means that $\mathcal{A}$ breaks the security of the hierarchical key assignment scheme. The probability of this event is at most ϵ_1. Thus we can compute the

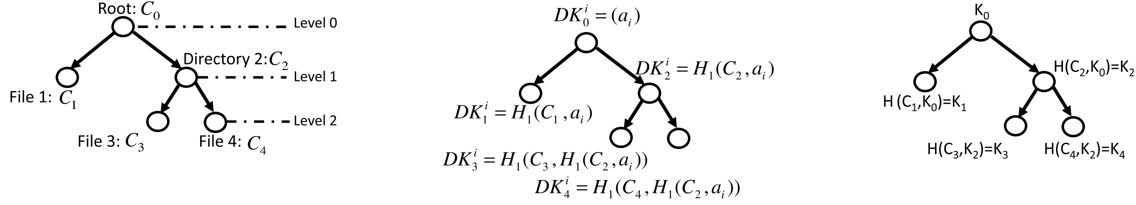

Fig. 5. The networked storage system Ω. The left one is the directory structure. The middle one is the hierarchical decryption key structure for the directory. The right one is the content key structure for the directory.

Content	Stored data	File blocks	Proxy keys
Root	$\mathsf{Enc}(\mathsf{EK}_0^i, K_0)$	None	None
File 1	$\mathsf{Enc}(\mathsf{EK}_1^i, K_1)$	$E(K_1,\text{File1})$	None
Directory 2	$\mathsf{Enc}(\mathsf{EK}_2^i, K_2)$	None	$\mathsf{PRK}_2^{(i\to x)}$
File 3	$\mathsf{Enc}(\mathsf{EK}_3^i, K_3)$	$E(K_3,\text{File3})$	$\mathsf{PRK}_3^{(i\to y)}$
File 4	$\mathsf{Enc}(\mathsf{EK}_4^i, K_4)$	$E(K_4,\text{File4})$	None

Fig. 6. The networked storage system Ω. The figure shows that how U^i's directories and files are stored. Directory 2 is granted to another user U^x by storing a proxy key $\mathsf{PRK}_2^{(i\to x)}$. File 3 is granted to another user U^y by storing a proxy key $\mathsf{PRK}_3^{(i\to y)}$.

Content	Stored data	File blocks	Proxy keys
the file f'	$\mathsf{Enc}(\mathsf{EK}_2^i, k_{f'})$	$E(k_{f'}, f')$	$\mathsf{PRK}_2^{(i\to x)}$

Fig. 7. The new entry for the new file f' in Directory 2 in the networked storage system Ω.

probability of the event $\mathcal{A}'$ wins the game as follows:

$$\begin{aligned}
&\Pr[0 \leftarrow \mathcal{A}'|\mathbb{Q} = \mathbb{Q}_0]\Pr[\mathbb{Q} = \mathbb{Q}_0] \\
&\quad + \Pr[1 \leftarrow \mathcal{A}'|\mathbb{Q} = \mathbb{Q}_1]\Pr[\mathbb{Q} = \mathbb{Q}_1] \\
&= (\Pr[0 \leftarrow \mathcal{A}'|K^* = K, \mathbb{Q} = \mathbb{Q}_0]\Pr[K^* = K|\mathbb{Q} = \mathbb{Q}_0] \\
&\quad + \Pr[0 \leftarrow \mathcal{A}'|K^* \neq K, \mathbb{Q} = \mathbb{Q}_0]\Pr[K^* \neq K|\mathbb{Q} = \mathbb{Q}_0])\frac{1}{2} \\
&\quad + (\Pr[1 \leftarrow \mathcal{A}'|K^* = K, \mathbb{Q} = \mathbb{Q}_1]\Pr[K^* = K, \mathbb{Q} = \mathbb{Q}_1] \\
&\quad + \Pr[1 \leftarrow \mathcal{A}'|K^* \neq K, \mathbb{Q} = \mathbb{Q}_1]\Pr[K^* \neq K, \mathbb{Q} = \mathbb{Q}_1])\frac{1}{2} \\
&\leq (1 \times \epsilon_0 + \frac{1}{2}(1 - \epsilon_0))\frac{1}{2} + (0 \times \epsilon_1 + \frac{1}{2}(1 - \epsilon_1))\frac{1}{2} \\
&= \frac{1}{2} + \frac{\epsilon_0 - \epsilon_1}{4}
\end{aligned}$$

Either ϵ_0 or ϵ_1 being non-negligible makes a contradiction to the decisional bilinear Diffie-Hellman assumption. ■

Lemma 2. *Assume that the discrete log problem is hard over the group $\mathbb{G}_1$ and the hash functions H_1, H_2 are random oracles. Our networked storage system meet the security requirement of the access control mechanism.*

Proof: We prove by contradiction. If there exists an adversary $\mathcal{A}$ winning the game $\mathcal{G}_1$ with non-negligible advantage ϵ, there exists an adversary $\mathcal{A}'$ solving the discrete log problem with non-negligible advantage.

The input of $\mathcal{A}'$ is (g, g^x), where $g \in \mathbb{G}_1$ is a group generator of $\mathbb{G}_1$ with prime order q. The goal of $\mathcal{A}'$ is to output the discrete logarithm value x. $\mathcal{A}'$ simulates the environment of $\mathcal{A}$ and uses $\mathcal{A}$ as a sub-routine.

- *Setup.* First, $\mathcal{A}'$ runs SetUp, passes the public parameter to $\mathcal{A}$. $\mathcal{A}$ generates the keys $\mathsf{PK}^{\mathcal{A}}, \mathsf{MK}^{\mathcal{A}}$ and $\mathsf{SK}^{\mathcal{A}}$ and sends $\mathsf{PK}^{\mathcal{A}}$ to $\mathcal{A}'$. $\mathcal{A}'$ then generates the keys for other users and sends the keys PK to $\mathcal{A}$.
- *Challenge.* $\mathcal{A}$ chooses a target user U^t and a target hierarchy Γ^t with a target class $C_{t'}$.
- *Response.* To give $\mathcal{A}$ all non-trivial proxy keys, $\mathcal{A}'$ implicitly maintains two hash lists for H_1 and H_2, and generates proxy keys as follows:
  - Set the x value as the decryption key of the target class. Randomly choose a value $\alpha \in Z_q$ as $H_2(x)$ and record it in the H_2-list. For each child class with identity C_i of the target class, randomly choose a value y_i as $H_1(C_i, x)$ and record it in the H_1-list. Also randomly assign values $\alpha_i = H_2(y_i)$ and record those values in H_2-list.
  - For the parent class C_j of the target class, randomly choose a value $y_j \in Z_q$ and set the

first part of the decryption key for C_j as y_j, i.e. set $H_1(C_{t'}, y_j) = x$ in the H_1-list. Then randomly choose a value $\alpha_j \in Z_q$ as $H_2(y_j)$ and record it in the H_2-list. Recursively repeat this step until the decryption key of the root class is set.

  - For the rest classes, randomly assign the decryption keys with their H_1 values and record those values in the H_1-list accordingly. Then assign the H_2-values randomly and record those random values in the H_2-list as well.
  - Generate the proxy keys $\mathsf{PRK}_j^{t \to \mathcal{A}}$ for all classes C_j in Γ^t except those in the path from $C_{t'}$ to the root by using $\mathsf{PK}^{\mathcal{A}}$ and the decryption keys.
  - *Output.* Finally, $\mathcal{A}$ outputs PRK^* for guessing the target proxy key $\mathsf{PRK}_{t'}^{t \to i}$.
    those proxy keys by using $\mathsf{PK}^{\mathcal{A}'}$ and those hash values in the H_2-list.

  Note that $\mathcal{A}'$ does not really know what x is, he only marks those values in these lists.
  $\mathcal{A}'$ simulates as follows. When $\mathcal{A}$ queries (s_1, s_2) to the H_1 oracle, $\mathcal{A}'$ aborts if $(s_1, s_2) = (C_{t'}, y_j)$ where $H_1(C_{t'}, y_j)$ is marked as x. Otherwise, $\mathcal{A}'$ checks whether s_2 equals x by comparing g^x and g^{s_2}. If $s_2 = x$, $\mathcal{A}'$ can output s_2 as the answer to the discrete log problem. If $s_2 \neq x$, $\mathcal{A}'$ returns $H_1(s_1, s_2)$ when (s_1, s_2) exists in the H_1-list, or $\mathcal{A}'$ randomly sets a value for $H_1(s_1, s_2)$ and records in the H_1-list. When $\mathcal{A}$ queries s_3 to the H_2 oracle, $\mathcal{A}'$ checks whether $s_3 = x$. If yes, $\mathcal{A}'$ outputs s_3 as the answer to the discrete log problem. Otherwise, $\mathcal{A}'$ returns $H_2(s_3)$ when s_3 exists in the H_2-list, or $\mathcal{A}'$ randomly sets a value for $H_2(s_3)$ and records in the H_2-list.
- *Output.* $\mathcal{A}$ outputs $\mathsf{PRK}_{t'}^{t \to \mathcal{A}}$.

When $\mathcal{A}'$ simulates successfully, $\mathcal{A}$ has the non-negligible advantage ϵ answering the correct proxy key. $\mathcal{A}'$ can solve the discrete log problem in this case. Hence we bound the probability that $\mathcal{A}'$ fails the simulation. $\mathcal{A}'$ fails the simulation, when $\mathcal{A}$ queries $(C_{t'}, y_j)$ to the H_1 oracle. Define the event as FAIL and let n_1 be the number of queries to H_1. Since $\mathcal{A}$ succeeds in querying $(s_1, s_2) = (C_{t'}, y_j)$ by guessing only, we have

$$\Pr[\mathsf{FAIL}] = n_1/q.$$

$\mathcal{A}$ has the advantage ϵ of answering the correct proxy key when $\mathcal{A}'$ simulates successfully. Define the event Succ as that $\mathcal{A}'$ solves the discrete log problem. When $\mathcal{A}$ can answer the correct proxy key, with a high probability $\mathcal{A}$ queries (s_1, x) to the H_1 oracle or x to the H_2 oracle. Then $\mathcal{A}'$ can use these queries to solve the discrete log problem. Define the event Q as that $\mathcal{A}$ queries (s_1, x) to H_1 oracle or x to H_2 oracle. Thus,

$$\Pr[\mathsf{Succ}] \geq \Pr[Q].$$

Define GetP as the event that $\mathcal{A}$ outputs the correct proxy key $\mathsf{PRK}_{t'}^{t \to \mathcal{A}}$, we have $\Pr[\mathsf{GetP}] \geq \epsilon(1 - n_1/q)$. The probability of the event Q then can be expressed as:

$$\Pr[\mathsf{Q}] = (\Pr[\mathsf{GetP}] - \Pr[\mathsf{GetP}|\bar{\mathsf{Q}}]\Pr[\bar{\mathsf{Q}}]) / \Pr[\mathsf{GetP}|\mathsf{Q}].$$

If the event Q does not happen, $H_2(x)$ could be any value over Z_q to $\mathcal{A}$. Let n_2 be the number of queries to the H_2 oracle. The probability of $\mathcal{A}$ giving the correct $\mathsf{PRK}_{t'}^{t \to \mathcal{A}} = g^{b_{\mathcal{A}} H_2(x)}$ is bounded by n_2/q, where $g^{b_{\mathcal{A}}}$ is $\mathsf{PK}^{\mathcal{A}}$, i.e. $\Pr[\mathsf{GetP}|\bar{\mathsf{Q}}] \leq n_2/q$. Thus the probability of the event Q can be bounded:

$$\Pr[\mathsf{Q}] \geq \epsilon(1 - n_1/q) - n_2/q.$$

It implies that $\Pr[\mathsf{Succ}] \geq \epsilon(1 - n_1/q) - n_2/q$. $\mathcal{A}'$ has non-negligible advantage of solving the discrete log problem. It makes a contradiction. ■

From Lemma 1 and Lemma 2, we conclude that our networked storage system is secure.

VII. Performance Evaluation

To evaluate the performance of our networked storage system, we implement our HIPRE construction and the key assignment scheme. We conduct experiments for the reading, writing, and granting operations. The experiments measure the time of writing a file, reading a file, sharing a file, and reading a file from another user in our networked storage system. To investigate the computation overhead of our system, we compare our system with the networked storage system where a (non-hierarchical) proxy re-encryption scheme is applied [8].

Implementation. We implement algorithms in our HIPRE construction. The proxy re-encryption scheme in [8] has been implemented as the JHU-MIT Proxy Re-cryptography Library. We implement the pairing operation and the proxy re-encryption operation by using the JHU-MIT Proxy Re-cryptography Library. The hash function h and the symmetric encryption scheme are implemented as the SHA1 hash function and the Advanced Encryption Standard (AES) encryption by using the Crypto++ Library. The key length of AES as well as the key length of content keys is set to 128-bit.

Experiments. All experiments are conducted on a 1.6GHz Intel CPU and 2GB RAM. We measure the time

TABLE I
EXPERIMENTAL RESULTS ON A SINGLE FILE WITH SIZE 256 MB IN SECONDS. OVERHEAD IS THE EXTRA TIME NEEDED BY HIPRE COMPARED WITH PRE.

	Read	Write	Share	Read from others
PRE	22.373	28.842	0.035	22.35
HIPRE	22.373	28.874	0.035	22.35
Overhead	0	0.032	0	0

of accessing a sample file with 256MB in our system. The results are listed in the third row in Table I. We also measure the time of accessing the same sample file in a networked storage system where only a proxy re-encryption scheme is applied. The results are listed in the second row in Table I. Compared with a proxy re-encryption scheme, HIPRE only takes extra 32 ms to gain the flexibility of the access patterns. The extra cost is produced by the hierarchical key generation process, and it is a constant respect to the file size. We conclude that HIPRE can protect the confidentiality of stored files and provide fine-grained access control with low computation overhead.

VIII. CONCLUSION

In this paper, we propose an access control mechanism by integrating our hierarchical proxy re-encryption scheme and the hierarchical key assignment scheme. In this storage system, one user can grant the reading permission of any subset of his files to another user. It provides a finer access pattern. Meanwhile, our networked storage system meets the security requirements. The users without the reading permissions cannot read those files, and the proxy server cannot illegally generate the proxy keys. It provides a strong security guarantee.

Acknowledgement. The research was supported in part by projects ICTL-101-Q707, ATU-101-W958, NSC 101-2218-E-009-003-, NSC 98-2221-E-009-068-MY3, NSC 100-2218-E-009-003, and NSC 96-3114-P-001-002-Y(the iCAST project).

REFERENCES

[1] J. Kubiatowicz, D. Bindel, Y. Chen, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, W. Weimer, C. Wells, and B. Zhao, "Oceanstore: An architecture for global-scale persistent storage," in *Proceedings of the Ninth international Conference on Architectural Support for Programming Languages and Operating Systems - ASPLOS'00*. ACM, 2000.

[2] P. Druschel and A. Rowstron, "PAST: A large-scale, persistent peer-to-peer storage utility," in *The 8th Workshop on Hot Topics in Operating System - HotOS VIII*. USENIX, 2001, pp. 75–80.

[3] M. Kallahalla, E. Riedel, R. Swaminathan, Q. Wang, and K. Fu, "Plutus: Scalable secure file sharing on untrusted storage," in *Proceeding of the 2nd Conference on File and Storage Technologies - FAST'03*. USENIX, 2003.

[4] Z. Wilcox-O'Hearn and B. Warner, "Tahoe: the least-authority filesystem," in *Proceedings of the 4th ACM international workshop on Storage security and survivability - StorageSS'08*. ACM, 2008, pp. 21–26.

[5] M. Blaze, "A cryptographic file system for unix," in *Proceedings of the ACM Conference on Computer and Communications Security - CCS'93*. ACM, 1993, pp. 9–16.

[6] G. Cattaneo, L. Catuogno, A. D. Sorbo, and P. Persiano, "The design and implementation of a transparent cryptographic file system for unix," in *Proceedings of the USENIX Annual Technical Conference*. USENIX Association, 2001, pp. 199–212.

[7] C. P. Wright, M. C. Martino, and E. Zadok, "Ncryptfs: A secure and convenient cryptographic file system," in *Proceedings of the USENIX Annual Technical Conference*. USENIX Association, 2003, pp. 197–210.

[8] G. Ateniese, K. Fu, M. Green, and S. Hohenberger, "Improved proxy re-encryption schemes wth applications to secure distributed storage," in *Proceeding of the Network and Distributed System Security Symposium - NDSS'05*. The Internet Society, 2005.

[9] M. Green and G. Ateniese, "Identity-based proxy re-encryption," in *Applied Cryptography and Network Security, 5th International Conference - ACNS'07*, ser. Lecture Notes in Computer Science, vol. 4521. Springer, 2007, pp. 288–306.

[10] C.-K. Chu and W.-G. Tzeng, "Identity-based proxy re-encryption without random oracles," in *Proceedings of the 10th International Conference on Information Security - ISC'07*, ser. Lecture Notes in Computer Science, vol. 4779. Springer, 2007, pp. 189–202.

[11] Q. Tang, "Type-based proxy re-encryption and its construction," in *Proceedings of the 9th International Conference on Cryptology in India - Indocrypt'08*, ser. Lecture Notes in Computer Science. Springer, 2008, pp. 130–144.

[12] X. Liang, Z. Cao, H. Lin, and J. Shao, "Attribute based proxy re-encryption with delegating capabilities," in *Proceedings of the 2009 ACM Symposium on Information - ASIACCS'09*. ACM, 2009, pp. 276–286.

[13] J. Zhao, D. Feng, and Z. Zhang, "Attribute-based conditional proxy re-encryption with chosen-ciphertext security," in *Proceedings of the Global Communications Conference - GLOBECOM'10*. IEEE, 2010, pp. 1–6.

[14] R. S. Sandhu, "On some cryptographic solutions for access control in a tree hierarchy," in *Proceedings of the 1987 Fall Joint Computer Conference on Exploring technology: today and tomorrow*. IEEE Computer Society Press, 1987.

[15] W.-G. Tzeng, "A time-bound cryptographic key assignment scheme for access control in a hierarchy," *IEEE Transactions on Knowledge and Data Engineering.*, vol. 14, no. 1, pp. 182–188, 2002.

[16] H.-Y. Chien and J. ke Jan, "New hierarchical assignment without public key cryptography," *Computers & Security*, vol. 22, no. 6, pp. 523–526, 2003.

[17] A. D. Santis, A. L. Ferrara, and B. Masucci, "Efficient provably-secure hierarchical key assignment schemes," in *Proceedings of the 32nd International Symposium on Mathematical Foundations of Computer Science*, ser. Lecture Notes in Computer Science, vol. 4708. Springer, 2007, pp. 371–382.