Hindawi
Mathematical Problems in Engineering
Volume 2018, Article ID 6091807, 13 pages
https://doi.org/10.1155/2018/6091807

Research Article

An Encrypted File Deduplication Scheme with Permission in Cloud Storage

Zuojie Deng,[1] **Xiaolan Tan**,[1] **and Shuhong Chen**[2]

[1]*School of Computer and Communication, Hunan Institute of Engineering, Xiangtan, Hunan 411104, China*
[2]*School of Computer Science and Educational Software, Guangzhou University, Guangzhou, Guangdong 510006, China*

Correspondence should be addressed to Zuojie Deng; zjdeng@hotmail.com

Received 24 January 2018; Revised 13 August 2018; Accepted 5 September 2018; Published 7 November 2018

Academic Editor: Emilio Insfran Pelozo

Encrypted file deduplication scheme (EFD) can improve its storage space utilization of cloud storage and protect the privacy of files in cloud storage. However, if an enterprise stores its files to cloud storage that has deployed an encrypted file deduplication scheme that does not support permission checking, this will destroy the permission of the enterprise files and bring some security problems. This seriously affects the practical value of EFD and prevents it from deploying in concrete cloud storage. To resolve this problem, we propose an encrypted file deduplication scheme with permission (EFDSP) and construct the EFDSP by using the hidden vector encryption (HVE). We have analyzed the security of EFDSP. The results have shown that EFDSP is secure and it can prevent the online deduplication oracle attack. We implement EFDSP and conduct the performance evaluation. The results show that the performance of EFDSP is little inferior to that of SADS, which is the only existing encrypted file deduplication scheme with permission, but the performance gap decreases with the increasing number of the authorized users and EFDSP has overcome the security weakness of SADS.

1. Introduction

1.1. Motivation. Recently, with the rapid development of network storage technology, cloud storage has become an important storage scheme. Owing to the rental cost lowness, outsourcing files of an enterprise to cloud storage can reduce its enterprise management costs and improve its competitiveness. To prevent files from information leakage, an enterprise user usually stores its files to cloud storage in an encrypted form. Encrypted file deduplication scheme can save its storage space and network bandwidth of cloud storage and improve its performance. However, in the enterprise application environment, different department employees have different permissions. Each employee can only access the files according to its permission. If an encrypted file deduplication scheme does not support permission checking, it will destroy the file permissions and bring some security problems. Li et al. proposed a secure authorized deduplication scheme based on a hybrid cloud (SADS) [1]. They introduce a private cloud in SADS to preserve the user permissions and generate a permission tag for a user when it uploads a file. When the cloud storage performs the deduplication checking for a user, it needs to check the deduplication permission for the user, and if the user does not have the deduplication permission, the user needs to upload the file even though there exists the same file in the cloud storage. Only when the user has the deduplication permission and there exists the same file in the cloud storage can the cloud storage perform file deduplication. The use of SADS can achieve the encrypted file deduplication, but there exist three shortcomings in SADS:

(i) Firstly, each permission is represented by a private key. If a user u has multiple permissions, it needs to store multiple private keys secretly which can cause a great deal of trouble in the user key management.

(ii) Secondly, when u uploads a file F or queries the duplication file of F, the scheme needs to use n permission keys to generate n encrypted file tags for F (If u has been assigned n permissions). So the scheme causes large network traffic.

(iii) Thirdly, there exists a security weakness in SADS. Assuming Mike is an enterprise manager who manages department A and department B. Mike has the permissions of department A and department B. At the same time, Mike is responsible for the financial department, so he also has the finance department permission. If a cloud storage uses SADS to deduplicate the files in the cloud storage, SADS uses the private keys of department A, department B, and the finance department to generate three encrypted file tags. As a result, the staffs in department A and department B have the permission to deduplicate their files with the payslip file. Suppose Mike has uploaded Alice's payslip file F to the cloud storage, if both Bob and Alice are employees of department B. Bob wants to get the salary information of Alice. He can use the following steps (called online deduplication oracle attack) to attack SADS to obtain the salary information of Alice:

(a) Bob first forges Alice's payslip file F. F is a kind of small entropy file and it has a fixed format. Bob knows the file format or he even has the kind of file, i.e., he has his own payslip. At the same time, he also knows that Alice's salary should be between 4000 and 4100 and he just does not know the concrete salary value of Alice. So Bob can set the salary value to $4000, 4001, \ldots 4100$, respectively, and generate 100 files $F_1, F_2, \ldots F_{100}$.

(b) Bob uploads $F_1, F_2, \ldots F_{100}$ to the cloud storage, respectively. If the cloud storage deduplicates the file when he uploads a file $F_i (1 \leq i \leq 100)$ to the cloud storage, Bob knows that the salary of Alice is the data in the uploaded file F_i.

Obviously, the success reason for the attack is the authorization precision of SADS which is rough. When Mike generates an encrypted file tag, it has assigned the file deduplication permission to Bob and causes the file permission checking bypass. At the same time, when the cloud storage checks the file deduplication, it only checks whether the encrypted file query tag of the upload file matches the encrypted file tags stored in the cloud storage owner and does not check the user's permission. Therefore, we want to design a securely encrypted file deduplication scheme with permission to improve the file deduplication permission check of the user and avoid the security issues of SADS.

1.2. Our Contributions. In this work, we study the problem on how to enable cloud storage to deduplicate a user encrypted file without destroying its file permission. We propose permission vector and permission relation, use permission vector to represent the user permissions, and use permission relation to compare the permission level between two users. We design an encrypted file deduplication scheme with permission, which has overcome the security weakness of SADS. In EFDSP, the file owner enables the cloud storage to perform deduplication when other users with the same or high permission level upload the duplication files to the cloud storage. Our contribution can be summarized as follows:

(i) Firstly, we discover a security weakness of SADS and propose an attack method against this scheme for small entropy files.

(ii) Secondly, we propose an encrypted file deduplication scheme with permission, which enables cloud storage to deduplicate the encrypted files without destroying the file permission. In EFDSP, a user with low permission level needs to upload the file even though there exists a duplication file in the cloud storage. EFDSP can prevent the online deduplication attack and overcome the security weakness of SADS.

(iii) Thirdly, we define permission vector and permission relation and use permission vector, permission relation, and hidden vector encryption to construct EFDSP.

(iv) Fourthly, we implement our scheme and conduct a performance evaluation, and the results demonstrate that our scheme is reasonable.

The paper is organized as follows. In Section 2, we present some preliminary knowledge. In Section 3, we describe and give the definition about the problem and define the encrypted file deduplication with permission. The permission vector and permission relation are defined in Section 4. In Section 5, we construct the encrypted file deduplication scheme with permission. In Section 6, we optimize EFDSP. In Section 7, we give some security analyses of EFDSP. In Section 8, we implement our scheme and conduct a performance evaluation, the evaluation results are presented here. In Section 9, we discuss related works. Finally, some conclusions are given in Section 10.

2. Preliminary

2.1. Bilinear Pairing

Definition 1. G_1, G_2, and G_T are three multiplicative cyclic groups with prime number order p, and g_1 and g_2 are the generators of G_1 and G_2, respectively. A bilinear pairing is a surjective function of the following properties:

(i) Bilinearity: for all $h_1, b_1 \in G_1$ and all $h_2, b_2 \in G_2$ we have $e(h_1 b_1, h_2) = e(h_1, h_2)e(b_1, h_2)$ and $e(h_1, h_2 b_2) = e(h_1, h_2)e(h_1, b_2)$.

(ii) Nondegeneration: $e(g_1, g_2) \neq 1$, where 1 is the identical element of G_T.

(iii) Computability: for all $h_1 \in G_1, h_2 \in G_2$, there exists an efficient algorithm that can compute $e(h_1, h_2)$.

If $G_1 \neq G_2$, we call $e : G_1 \times G_2 \longrightarrow G_T$ an asymmetric bilinear pairing; otherwise, if $G_1 = G_2$, we call $e : G_1 \times G_2 \longrightarrow G_T$ a symmetrical bilinear pairing. According to Definition 1, we can get Proposition 2 easily.

Proposition 2. *Let $h_1 \in G_1, h_2 \in G_2$ and $a, b \in Z_p$. Then $e(h_1^a, h_2^b) = e(h_1, h_2)^{ab}$.*

2.2. Hidden Vector Encryption. Hidden vector encryption (HVE) was first proposed by Doneh and Waters [2]. Subsequently, Katz [3] and Park [4] proposed some HVE schemes, respectively. HVE is a kind of predicate encryption, which has two attribute vectors associated with the ciphertext and the tag. Only when the two vectors are equal does the ciphertext match the tag. There are two character sets Σ and Σ_* in HVE, where $\Sigma_* = \Sigma \cup \{*\}$ and $*$ is a wildcard. If a vector of a component is $*$, it means that it does not participate in any of the attributes. HVE is mainly composed of four algorithms: key generation, data encryption, tag generation, and data query.

(i) In the key generation phase, the trusted authority (TA) assigns a public/private key pair (PK, SK) to a receiver.

(ii) In the data encryption phase, the user selects a vector $x = (x_1, x_2, \ldots, x_l) \in \Sigma^l$ to describe its data m and also uses the receiver's public key PK to encrypt the data m to obtain the ciphertext CT.

(iii) In the tag generation phase, the receiver first selects a vector $w = (w_1, w_2, \ldots, w_l) \in (\Sigma_*)^l$ to represent the query requirement and then uses its private key SK to generate a query tag T_W. Finally, the receiver sends T_W to the server.

(iv) In the data query phase, if x matches w, it outputs m, which is the plaintext of CT. The matching condition is defined as follows: let $s(w)$ be the subscript set that w_i is not $*$, where w_i is a vector component of $(w_1, w_2, \ldots, w_l)$. For two vectors x and w, let $P_w(x)$ be the equality predicate that satisfies (1).

$$P_w(x) = \begin{cases} 1 & \text{if } \forall i \in s(w) \ \ w_i = x_i, \\ 0 & \text{otherwise.} \end{cases} \tag{1}$$

3. Problem and Definition

3.1. The System Model. In order to facilitate the enterprise management, we need to introduce a permission server (PS) to manage the user permission. At the same time, we need to introduce a key generation server (KGS) to generate an encryption key for the upload file. After introducing PS and KGS, the system model of cloud storage is shown in Figure 1. It consists of four different kinds of entities: some users, a cloud storage, a permission server, and a key generation server. The permission server and the key generation server are deployed in the enterprise domain, which are absolutely secure. The cloud storage (CS) checks whether there exists a duplication file in the cloud storage and checks whether the user has the permission to deduplicate the file. If both conditions met, the user does not need to upload the file, and the cloud storage server provides it with a file pointer; otherwise, the user needs to upload this file.

When the system is initialized, the system administrator gives the user access permission according to its permission level. The system administrator can use the role-based method [5] to assign the permission to the user; that is, it assigns the permission to the user based on the role of the user. Suppose an IT company has only three types of employees: manager, project leader, and engineer; if a user A is assigned the permission of the manager, then A can access any file that its access role is the manager. Each file in the cloud storage has a file permission tag to describe its permission, only when other users with the same permission upload a duplication file can the cloud storage perform the deduplication.

Cloud storage provides its users with the data storage service. To reduce its storage costs, CS only stores one unique file by using cross-user file deduplication to eliminate the redundant files in its server. PS and KGS are deployed in the enterprise secure domain, which are absolutely secure. PS is responsible for the user permission management and the file permission query, and it assists CS to perform the file permission checking and the file deduplication. KGS is responsible for generating an encryption key for the user. When a user needs to store a file to CS, it needs to interact with KGS and gets an encryption key from KGS for the file.

3.2. Problem Formalization. In this work, we study the problem on how to enable the cloud storage to deduplicate the user encrypted file without destroying the file permissions. That is to say, we study the problem on how to enable the file owner to allow the cloud storage to perform deduplication when other users with the same or high permission level upload a duplication file to the cloud storage. We can formalize the problem as follows.

When a user, say u, wants to upload a file F to CS, it first interacts with KGS to get the encryption key K_F for F, then it interacts with PS. PS uses $H(F)$, the permission level of u PV_u and its private key K_{PS} to generate QFT_F for u, where QFT_F is a permission query tag of F. After receiving QFT_F, u sends QFT_F to CS to query whether there exists the encrypted file C_F in the cloud storage. If there exists C_F in the cloud storage, u does not need to upload F, and it only needs to store K_F; otherwise, u first encrypts F using K_F to get C_F, then uses PV_u and PK_{PS}, where PK_{PS} is the public key of PS, to generate the encrypted file tag FT_F for F. Finally, u sends FT_F and C_F to the cloud storage.

3.3. The EFDSP Scheme. In order to solve the problem that we have formalized in Section 3.2, we design an encrypted file duplication scheme with permission.

Definition 3 (EFDSP). An encrypted file duplication scheme with permission is a tuple of algorithms as follows:

(i) Setup(k) $\longrightarrow$ *params*: it takes the security parameter k as input and outputs the public parameter *params*.

(ii) KeyGeneration(*params*) $\longrightarrow$ $(PK_S, SK_S, SK_{PS}, PK_{PS}, Sig_{PS}, Ver_{PS})$: it takes *params* as input and outputs (PK_S, SK_S), which is an identity-based key pair of CS, (SK_{PS}, PK_{PS}), which is a private key/public key pair of PS, and (Sig_{PS}, Ver_{PS}), which is a signature/verification key pair of PS.

(iii) FileKeyGeneration($H(F)$) $\longrightarrow$ K_F: this algorithm is run by KGS, and it takes $H(F)$, which is the hash value

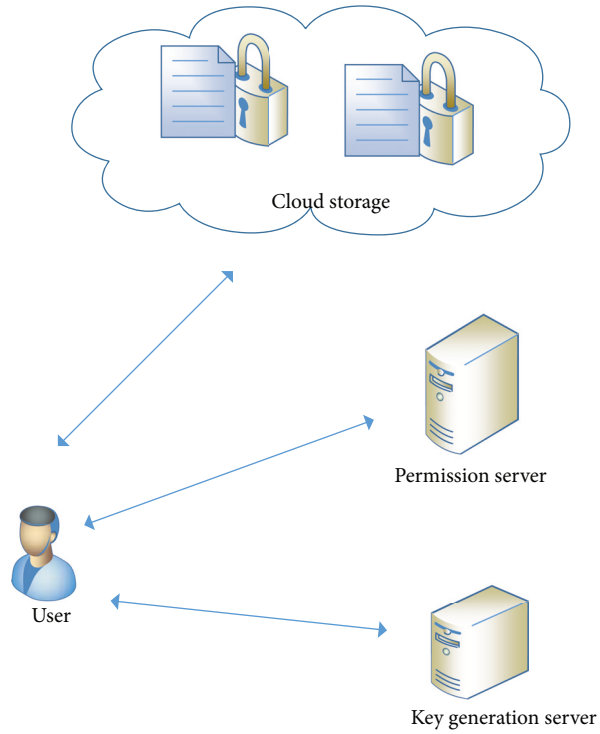

Figure 1: The system model.

of the user file F, as input and outputs a file encryption key K_F for F.

(iv) FileTagGeneration(F, PV, PK_{PS}) $\longrightarrow$ FT_F: this algorithm is run by u, and it takes F, PV, and PK_{PS} as input and generates an encrypted file tag FT_F as output. PV is the permission level of the user. PK_{PS} is the public key of PS.

(v) FileQueryTagGeneration($H(F)$, PV, SK_{PS}, Sig_{PS}) $\longrightarrow$ $(QFT_F, (QFT_F)_{Sig_{PS}})$: this algorithm is run by PS, and it takes $H(F)$, PV, SK_{PS}, and Sig_{PS} as input and outputs QFT_F and $(QFT_F)_{Sig_{PS}}$. $H(F)$ is the hash value of F, PV is the permission level of u, SK_{PK} is the private key of PS, and Sig_{PS} is the signature private key of PS.

(vi) FileQueryTagQueryAndFileDeduplication(ID_u, $H(F)$, QFT_F, $(QFT_F)Sig_{PK}$) $\longrightarrow$ ($LoadFlag$, FP_F): this algorithm is run by CS, and it takes $H(F)$, QFT_F, and $(QFT_F)Sig_{PK}$ as input and outputs $LoadFlag$ and FP_F. CS uses QFT_F to match some encrypted file C_F. If it matches, let $LoadFag = 1$ and let FP_F be the file pointer of C_F, and add ID_u to the ID_u to the file entry of C_F; otherwise, let $LoadFag = 0$ and assign NULL to FP_F.

(vii) FileUpload($H(F), C_F, FT_F$) $\longrightarrow$ (FP_F): this algorithm is run by CS, and it takes $H(F)$, C_F, and FT_F as input and outputs FP_F. $H(F)$ is the hash value of F, C_F is the encrypted file of F, and FT_F is the encrypted file tag of F.

(viii) Enc(F_k, F) $\longrightarrow$ C_F: this algorithm is run by u, and it uses F_K to encrypt F to generate C_F.

(ix) Dec(C_F, F_K) $\longrightarrow$ F: this algorithm is run by u, and it uses F_K to decrypt C_F to generate F.

(x) FileRetrieval(FP_F) $\longrightarrow$ C_F: this algorithm is run by CS, and it uses FP_F to search the encrypted files in CS and returns its corresponding encrypted file C_F.

The interaction process of EFDSP is described in Figure 2.

3.4. The Threat Model. Since PS is responsible for the user permission management and the file permission query and KGS is responsible for generating an encryption key for the user, we must assume that PS and KGS are absolutely secure and reliable. As CS performs the tasks assigned to it honestly and it is interested in the content of the user's files and tries to get some secret information from these files, we can regard it as an honest and curious adversary [6]. Some users try to access the files beyond their permissions. At the same time, we assume that all files stored in the cloud storage are confidential; if there is information disclosure, it will result in a very large loss to the user. According to this assumption, there are two kinds of adversaries in the system.

(1) External adversary: it tries to obtain secret information from the cloud storage or tries to access the file beyond its permission.

(2) Internal adversary: it can access the cloud storage easily and try to get some secret information from the encrypted file tags or the query tags.

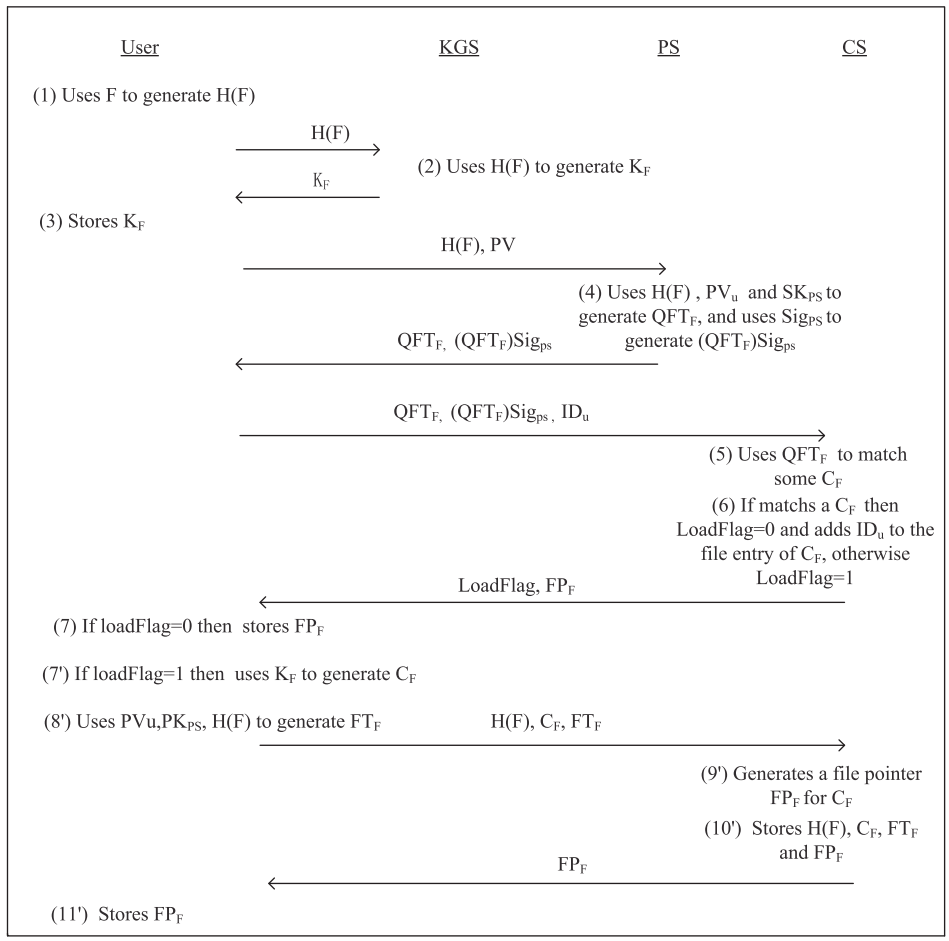

Figure 2: The interaction process of EFDSP.

3.5. The Security Requirements. According to the threat model described in Section 3.4, there exist four security requirements as follows:

(1) The confidentiality of the encrypted file tag: an unauthorized user, including the cloud storage server, cannot get the plaintext information from the encrypted file tags stored in the cloud storage server.

(2) The unforgeability of the encrypted file query tag: an unauthorized user should be prevented from getting or generating the encrypted file query tags because it has no appropriate permission. It is not allowed to collude with the cloud storage server to destroy the unforgeability of the query tags.

(3) The indistinguishability of the encrypted file query tag: a user cannot get any information from the query tags without querying the permission server, including the file content and the permissions.

(4) The confidentiality of the file: a user who does not own the files cannot obtain the plaintext from the files stored in the cloud storage server; that is, an adversary cannot retrieve and restore files that do not belong to it.

4. The Permission Vector and the Permission Relation

In order to effectively represent the user permission, we define permission vector in this section.

Definition 4 (permission vector). Let $T = (T_1, T_2, \ldots, T_N)$ be a collection of the system permission, 1 to N are the sequence numbers of the permissions in the system. Permission vector TV is a bit binary vector of n bits, which are numbered 1 to n from left to right. $TV(i)$ represents the permission T_i. If the value of $TV(i)$ is 0, it means that the permission T_i is valid, otherwise it means that the permission T_i is invalid.

Figure 3 is an example of role hierarchies given in [5]. It has four roles: programmer, test engineer, project member, and project supervisor. We can easily represent the permission of each role by using the permission vector. Let ITP={programmer, project engineer, project member, project supervisor} be the basic permission set of the system. Because there are only four basic permissions, we can use a 4-bit permission vector to represent the permission of each role;

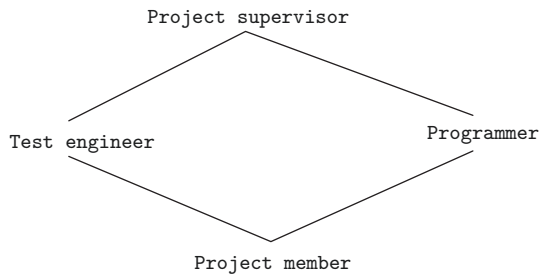

Figure 3: An example of role hierarchies.

the sequence number of the four basic permissions in the permission vector is 1, 2, 3, and 4, respectively. At the same time, the permission of these roles allows being inherited in [5]. From Figure 3, we can find that the project supervisor owns the project supervisor permission and inherits both permissions of the test engineer and the programmer. According to Definition 4, it is easy to get that the permission vector of the supervisor is 0010, the permission vector of the programmer is 0111, and the permission vector of the project member is 0001.

Definition 5 (permission relation). Let TV_{U_1} be the permission vector of user U_1, TV_{U_2} be the permission vector of user U_2; we can define permission relation as follows:

(1) If for each i $TV_{U_2}(i) = 0$ and $TV_{U_1}(i) = 0$, and there exist 0 or more j where $TV_{U_2}(j) = 1$ and $TV_{U_1}(j) = 0$, then we say the permission level of U_1 is higher than that of U_2. We use $TV_{U_1} \geq TV_{U_2}$ to denote it.

(2) If for each i $TV_{U_2}(i) = 0$ and $TV_{U_1}(i) = 0$, and there are 0 or more j where $TV_{U_2}(j) = 0$ and $TV_{U_1}(j) = 1$, then we say the permission level of U_1 is lower than that of U_2. We use $TV_{U_1} \leq TV_{U_2}$ to denote it.

(3) If there exists i where $TV_{U_1}(i) = 0$ and $TV_{U_1}(i) = 1$, and there exists j where $TV_{U_1}(j) = 1$ and $TV_{U_2}(j) = 0$, then we say the permission level of U_1 is not equal to that of U_2. We use $TV_{U_1} \neq TV_{U_2}$ to denote it.

(4) If for each i $TV_{U_1}(i) = TV_{U_2}(i)$, then we say the permission level of U_1 equals that of U_2. We use $TV_{U_1} = TV_{U_2}$ to denote it.

According to Figure 3, we can get the permission vectors of the project supervisor, the programmer, and project member which are 0010, 0111, and 0001, respectively. If both Alice and Bob are programmers, the permission vectors of Alice and Bob are 0111. According to Definition 5, we can get $TV_{projectsupervisor} > TV_{programmer}$, $TV_{programmer} < TV_{prjectmember}$, $TV_{projectsupervisor} \neq TV_{projectmember}$, and $TV_{Alice} = TV_{Bob}$.

With the definitions of permission vector and permission relation, we can define the permission equality predicate.

Definition 6 (permission equality predicate). Let TV_{U_1} be the permission vector of user U_1 and TV_{U_2} be the permission vector of user U_2. If $P_{TV_{U_1}}(TV_{U_2})$ satisfies (2), then we call $P_{TV_{U_1}}(TV_{U_2})$ the permission equality predicate of the users U_1 and U_2.

$$P_{TV_{U_1}}\left(TV_{U_2}\right) = \begin{cases} 1 & \text{if } TV_{U_1} = TV_{U_2}, \\ 0 & \text{otherwise.} \end{cases} \quad (2)$$

5. A Construction for EFDSP

We have defined EFDSP in Section 3.3. In this section, we use the efficient hidden vector encryption proposed by Park [4] and the permission vector defined in Section 4 to construct it. Let $H : \{0,1\}^* \longrightarrow G_2$ and $H_1 : \{0,1\}^* \longrightarrow G_1$ be two secure cryptography hash functions, which are modeled as random oracles. Let k be the security parameter, then our constructions for EFDSP are as follows:

(i) Setup(k) $\longrightarrow$ *params*: it takes k as input and outputs *params*, where *params* $= (p, g, G_1, G_2, e)$. g is a generator of the group G_1 and e is the map of $G_1 \times G_1 \longrightarrow G_2$.

(ii) KeyGeneration(*params*) $\longrightarrow$ (PK_S, SK_S, SK_{PS}, PK_{PS}, Sig_{PS}, Ver_{PS}): it takes *params* as input and outputs (PK_S, SK_S), SK_{PS}, PK_{PS}, and (Sig_{PS}, Ver_{PS}). (PK_S, SK_S) is an identity-based key pair of the cloud storage CS, and (SK_{PS}, PK_{PS}) is a private key/public key pair of PS. It randomly selects a master key $r \in Z_p$ and computes its corresponding public key g^r. It computes $PK_S = (H_1(ID_s)$ and $SK_S = H_1(ID_s)^r)$. It randomly selects $g_1, g_2, (h_1, u_1, \psi_1), \ldots, (h_n, u_n, \psi_n) \in G_1$ and $y_1, y_2, v_1, \ldots, v_n, t_1, t_2, \ldots, t_n \in Z_p$ and computes $Y_1 = g^{y_1}, Y_2 = g^{y_2}, V_k = g^{v_k}, T_k = g^{t_k} \in G_1, k \in \{1, 2, \ldots, n\}$. It also computes $\Gamma = e(g_1, Y_1)e(g_2, Y_2) \in G_2$. Let $SK_{PS} = (g_1, g_2, y_1, y_2, v_1, \ldots, v_n, t_1, t_2, \ldots, t_n)$, $PK_{PS} = (g, Y_1, Y_2, (h_1, u_1, \psi_1, V_1, T_1), (h_2, u_2, \psi_2, V_2, T_2), \ldots, (h_n, u_n, \psi_n, V_n, T_n))$. ($Sig_{PS}$, Ver_{PS}) is a signature/verification key pair of PS, and in our construction, we use DSA [7].

(iii) FileKeyGeneration($H(F)$) $\longrightarrow$ K_F: it takes $H(F)$ as input and performs the key generation algorithm based BLS signature [8] to generate K_F.

(iv) FileTagGeneration(F, PV, PK_{PS}) $\longrightarrow$ FT_F: it first uses the secure cryptography hash function H to compute the cryptography hash value of F, then it uses PV to generate the permission vector $x_u = (x_1, x_2, \ldots, x_n)$ according to Definition 4, and finally uses PK_{PS} which is the public key of PS to generate the encrypted file tag FT_F. The concrete steps are as follows.

(a) It gets $H(F)$ by using the secure cryptography hash function H on F.

(b) It uses its permission level PV to generate the permission vector $x_u = (x_1, x_2, \ldots, x_n)$ according to Definition 4. Let tx be the encrypted file permission index subscript set, then $tx = \{i \mid 1 \leq i \leq n\}$.

(c) It randomly selects two numbers r_1 and r_2 from Z_p and uses $x_u = (x_1, x_2, \ldots, x_n)$ and $H(F)$

to generate the encrypted file tag FT_F for F according to (3).

$$\begin{aligned} C_1 &= Y_1^{r_1}, \\ C_2 &= Y_2^{r_1}, \\ C_{3,1} &= \left(h_1 u_1^{x_1}\right)^{r_1} V_1^{r_2}, \ldots C_{3,n} = \left(h_n u_n^{x_n}\right)^{r_1} V_n^{r_2}, \\ C_{4,1} &= \psi_1^{r_1} T_1^{r_2}, \ldots, C_{4,n} = \psi_n^{r_1} T_n^{r_2}, \\ C_5 &= g^{r_2}, \\ C_6 &= \Gamma^{r_1} H(F), \\ FT_F &= \left(C_1, C_2, C_{3,1} \ldots, C_{3,n}, C_{4,1} \ldots, C_{4,n}, C_5, C_6\right) \end{aligned} \tag{3}$$

(v) FileQueryTagGeneration($H(F)$, PV, SK_{PS}, Sig_{PS}) $\longrightarrow$ (QFT_F, $(QFT_F)_{Sig_{PS}}$): PS first gets PV the permission level of u from its permission database and then according to Definition 4 generates a permission query vector for u, and finally it uses its own private key SK_{PS} to generate an encrypted file query tag QFT_F for u. The concrete steps are as follows.

(a) PS gets PV the permission level of u from its permission database and then generates the permission query vector $qv = (q_1, q_2, \ldots, q_n)$ according to Definition 4. Let tq be the permission query index set, then $tq = \{i \mid i \le i \le n\}$.

(b) PS randomly selects $\alpha, \beta \in Z_p$ and for each $i \in tq$ it generates $\lambda_i, \varphi_i, \gamma_i, \tau_i \in Z_p$ according to (4), and y_1, y_2 are the parts of SK_{PS} which is the private key of PS.

$$\begin{aligned} \lambda_i y_1 + \varphi_i y_2 &= \alpha \\ \gamma_i y_1 + \tau_i y_2 &= \beta \end{aligned} \tag{4}$$

(c) PS computes QFT_F according to (5).

$$\begin{aligned} T_1 &= g_1 \Pi_{i \in tq} \left(h_i u_i^{q_i}\right)^{\lambda_i} \psi_i^{\gamma_i}, \\ T_2 &= g_2 \Pi_{i \in tq} \left(h_i u_i^{q_i}\right)^{\varphi_i} \psi_i^{\tau_i}, \\ T_3 &= g^{\alpha}, \\ T_4 &= g^{\beta}, \\ T_5 &= g^{-\sum_{i \in tq} (v_i \alpha + t_i \beta)}, \\ QFT_F &= \left(\left(T_1, T_2, T_3, T_4, T_5\right), tq, H(F)\right) \end{aligned} \tag{5}$$

(d) PS uses its signature private key Sig_{PS} to sign QFT_F to generate $(QFT_F)_{sig_{PK}}$.

(vi) FileQueryTagQueryAndFileDeduplication(ID_u, $H(F)$, QFT_F, $(QFT_F)sigPK$) $\longrightarrow$ ($LoadFlag$, FP_F): CS first checks the format of QFT_F and $(QFT_F)_{sig_{PK}}$, and if there exists an error in its format, it stops. Otherwise it looks for the corresponding encrypted file tag according to $H(F)$, and if it does not find it, then it stops. Otherwise it computes FM according to

$$FM = \frac{D_1 D_2 e\left(T_5, C_5\right)}{e\left(T_1, C_1\right) e\left(T_2, C_2\right)} C_6 \tag{6}$$

where $D_1 = e(T_3, \Pi_{i \in tq} C_{3,i})$, $D_2 = e(T_4, \Pi_{i \in tq} C_{4,i})$.

If $FM = H(F)$, then it represents that F has been sent to CS and u has the deduplication permission for F. CS can perform deduplication for F and let $LoadFlag = 0$ and return the file pointer of F FP_F and add ID of u to the corresponding file entry of F, otherwise let $LoadFlag = 1$ and return $LoadFlag$.

(vii) FileUpload($H(F), C_F, FT_F$) $\longrightarrow$ (FP_F): when CS receives $H(F)$, C_F, and FT_F, it assigns a file pointer FP_F for C_F and creates a file entry to store $H(F)$, C_F, and FT_F.

(viii) Enc(F_k, F) $\longrightarrow$ C_F: it encrypts F by using AES [9]; that is, $C_F = E_{F_k}(F)$.

(ix) Dec(C_F, F_K) $\longrightarrow$ F: it decrypts C_F by using AES; that is, $F = D_{F_k}(C_F)$.

(x) FileRetrieval(FP_F) $\longrightarrow$ C_F: after receiving FP_F, it returns the corresponding encrypted file C_F of FP_F.

6. Optimization for EFDSP

Since EFDSP can only deduplicate files between users that have the same permissions, it has two shortcomings. Firstly, users with the high permission level can operate the files of users with the low permission level in the actual enterprise setting. However, EFDSP does not allow the cloud storage to perform deduplication between files of a user with high permission level and files of a user with low permission level, which violates the actual permission management in the enterprise setting, and it is not conducive to improving the deduplication efficiency. Secondly, during the generation of the encrypted file query tag in EFDSP, all the permission bits are involved in the computation which increases the computation cost.

In this section, we use the example in Figure 3 to illustrate how to optimize the permission query index subscript set to overcome the above shortcomings in EFDSP. The permission vector of the project supervisor $TV_{projectsupervisor}$ is 0010, and the permission vector of the programmer $TV_{programmer}$ is 0111. According to Definition 5, $TV_{projectsupervisor} \ge TV_{programmer}$. That is, the permission level of the project supervisor is higher than that of the programmer. Since 0 indicates the user has the permission and 1 indicates that the user does not have the permission, if EFDSP compares the permission level of u_1 with that of u_2, it only needs to consider these permissions that are not owned by u_1 whether are owned by u_2. If u_2 does not own these permissions that are not owned by u_1, then it means that the permission level of u_1 is higher than or equal to that of u_2. Otherwise if u_2 owns one permission that is not owned by u_1, then it means that the permission level of u_1 does not match that of u_2. (Either the permission level of u_1 is lower than that of u_2 or

the permission level of u_1 is unequal to that of u_2). So when EFDSP compares the permission level of u_1 with that of u_2, it only needs to consider the bits in the permission vector of u_1 which are 1. For example, if EFDSP wants to compare the permission level of project supervisor with that of programmer, as $TV_{projectsupervisor} = 0010$, $TV_{programmer} = 0111$, and all bits of $TV_{progjectsupervisor}$ are 0 except that bit 3 is 1, so EFDSP only needs to compare $TV_{projectsupervisor}[3]$ with $TV_{programmer}[3]$. Because $TV_{projectsupervisor}[3] = TV_{programmer}[3]$, it can derive that $TV_{projectsupervisor} \geq TV_{programmer}$ and can determine that the files of the project supervisors can be deduplicated with the files of the programmer that are stored in the cloud storage. If EFDSP wants to compare the permission level of the project supervisor with that of project member, as $TV_{projectsupervisor} = 0010$, $TV_{projectmember} = 0001$, and all bits of $TV_{projectsupervisor}$ are 0, except that bit 3 is 1, so it only needs to compare $TV_{projectsupervisor}[3]$ with $TV_{progammer}[3]$. Because $TV_{projectsupervisor}[3] \neq TV_{progammer}[3]$, the permission level of the project supervisor does not match that of project member, and EFDSP can determine the files of project supervisor which cannot deduplicate with the files of project member that are stored in the cloud storage.

That is to say, EFDSP only considers the bits of the permission vector of the query user that are 1. These vector bits form a set which is defined in (7). We call it permission query index subscript set and use etq to represent it. If we replace tq in the FileQueryTagGeneration algorithm with etq, then EFDSP can enable the cloud storage to perform deduplication between files of a user with high permission level and files of a user with low permission level, which can improve its efficiency. In addition, in order to prevent all bits of a permission vector from being 0, the bit number of the permission vector is required to be 2 more than the permission number, and EFDSP reserves the last two bits of the permission vector and codes them to be 1.

$$etq = \{i \mid q_i = 1\} \tag{7}$$

7. Security Analyses for EFDSP

In this section, we analyze EFDSP according to the security requirements discussed in Section 3.5. We analyze the correctness of EFDSP, the security of the encrypted file query tag which included unforgeability and indistinguishability, the confidentiality of the encrypted file tag, and the confidentiality of the encrypted file. Finally, we compare EFDSP with SADS [1].

7.1. The Correctness Analysis. To verify the correctness of EFDSP, we must verify the query process of the encrypted file query tag in EFDSP. In (6)

$$D_1 = e\left(g^{\alpha}, \prod_{i\in tq}\left(h_i u_i^{x_i}\right)^{r_i} g^{v_i r_2}\right),$$

$$D_2 = e\left(g^{\beta}, \prod_{i\in tq}\psi_i^{r_1} g^{t_i r_2}\right),$$

$$D_1 D_2 = e\left(g^{\alpha}, \prod_{i\in tq}\left(h_i u_i^{x_i}\right)^{r_i}\right) e\left(g^{\beta}, \prod_{i\in tq}\psi_i^{r_1}\right) \cdot e\left(g^{r_2}, \prod_{i\in tq} g^{(v_i\alpha + t_i\beta)}\right),$$

$$e(T_5, C_5) = e\left(g^{-\sum_{i\in tq}(v_i\alpha + t_i\beta)}, g^{r_2}\right)$$

$$\text{and } C_6 = \Gamma^{r_1} H(F). \tag{8}$$

We can get

$$D_1 D_2 e(T_5, C_5) C_6 = e\left(g^{\alpha}, \prod_{i\in tq}\left(h_i u_i^{x_i}\right)^{r_i}\right) e\left(g^{\beta}, \prod_{i\in tq}\psi_i^{r_1}\right)\Gamma^{r_1} H(F)$$

$$e(T_1, C_1) = e\left(g_1 \prod_{i\in tq}\left(h_i u_i^{q_i}\right)^{\lambda_i}, g^{y_1 r_1}\right)$$

$$e(T_2, C_2) = e\left(g_2 \prod_{i\in tq}\left(h_i u_i^{q_i}\right)^{\varphi_i}, g^{y_2 r_1}\right) \tag{9}$$

$$e(T_1, C_1)\, e(T_2, C_2) = \Gamma^{r_1}\prod_{i\in tq} e\left(\left(h_i u_i^{q_i}\right)^{r_1}, g^{\lambda_i y_1 + \psi_i y_2}\right)\prod_{i\in tq} e\left(\psi_i^{r_i}, g^{\gamma_i y_1 + \tau_i y_2}\right)$$

$$= \Gamma^{r_1} e\left(\prod_{i\in tq}\left(h_i u_i^{q_i}\right)^{r_1}, g^{\alpha}\right) e\left(\prod_{i\in tq} e\left(\psi_i^{r_i}, g^{\beta}\right)\right.$$

Let $S = \{i \mid i \in tq \text{ and } x_i \neq q_i\}$, and we can get $D_1 D_2 e(T_5, C_5) C_6 / e(T_1, C_1) e(T_2, C_2) = H(F) e(g^{\alpha}, \prod_{i\in S} u_i^{(x_i - q_i) r_i}) = H(F) e(g, g)^{ar\sum_{i\in S}(\log_g u_i)(x_i - q_i)}$.

Therefore, if $x = q$, then the above formula outputs $H(F)$; otherwise, it does not output $H(F)$.

7.2. The Security Analysis

(1) The Unforgeability of the Encrypted File Query Tag Analysis. In EFDSP the user passes the authentication of PS and sends $H(F)$ to PS. After receiving $H(F)$, PS first searches the permission database to find the permissions of the user and generates a permission query vector $q = (q_1, q_2, \ldots, q_n)$ in accordance with Definition 4 for the user, and then PS uses its own private key to generate the query tag, since the private key of PS is kept secret and we ensure the unforgeability of the encrypted file query tag.

(2) The Indistinguishability of the Encrypted File Query Tag Analysis. The encrypted file query tag $QFT(H(F)) = ((T_1, T_2, T_3, T_4, T_5), tq, H(F))$ is made up of four parts, where $T_1 = g_1 \prod_{i\in tq}(h_i u_i^{q_i})^{\lambda_i}\psi_i^{\gamma_i}$, $T_2 = g_2 \prod_{i\in tq}(h_i u_i^{q_i})^{\varphi_i}\psi_i^{\tau_i}$, $T_3 = g^{\alpha}$, $T_4 = g^{\beta}$, $T_5 = g^{-\sum_{i\in tq}(v_i\alpha + t_i\beta)}$, $tq = \{i \mid q_i = 1\}$. Since PS

randomly selects $\alpha, \beta \in Z_p$ when it generates the query tag, we can regard T_3 and T_4 as two random numbers. According to (4) we can get $\lambda_i y_1 + \varphi_i y_2 = \alpha$ and $\gamma_i y_1 + \tau_i y_2 = \beta$, where y_1 and y_2 are parts of the private key of PS, so that we can regard $\lambda_i, \varphi_i, \gamma_i$, and τ_i as random numbers, and then we can regard T_1, T_2, and T_5 as three random numbers. Because H is a secure cryptography hash function, we can also regard $H(F)$ as a random number. $tq = \{i \mid q_i = 1\}$ is publicly released, it is unconducive to help the probabilistic polynomial time (p.p.t) adversary to distinguish the encrypted file query tag with a random number; at the same time, there exist thousands of files with the same permission in the cloud storage, which make tq not useful to distinguish between the encrypted file query tag and a random number. Thus, we can ensure the indistinguishability of the encrypted file query tag.

(3) The Confidentiality of Encrypted File Tag Analysis. In EFDSP, when a user needs to generate an encrypted file tag FT_F for the encrypted file C_F, it first uses its own permission level to generate the permission vector $x = (x_1, x_2, \ldots, x_n)$ according to Definition 4 and finally uses PK_{PS} to generate an encrypted file tag FT_F, where PK_{PS} is the public key of PS. When it computes FT_F, it randomly selects two numbers r_1 and r_2 from Z_p. $FT_F = (C_1, C_2, C_{3,1}, \ldots, C_{3,n}, C_{4,1}, \ldots, C_{4,n}, C_5, C_6)$, and $C_1 = Y_1^{r_1}, C_2 = Y_2^{r_1}, C_{3,1} = (h_1 u_1^{x_1})^{r_1} V_1^{r_2}, \ldots, C_{3,n} = (h_n u_n^{x_n})^{r_1} V_n^{r_2}, C_{4,1} = \psi_1^{r_1} T_1^{r_2}, \ldots C_{4,n} = \psi_n^{r_1} T_n^{r_2}, C_5 = g^{r_2}, C_6 = \Gamma^{r_1} H(F)$. Since r_1 and r_2 are two random numbers, it is difficult for an p.p.t adversary to distinguish FT_F from a random number, thus it can ensure the confidentiality of FT_F.

(4) The Confidentiality of the File Analysis. In EFDSP, for any file F, $C_F = E_{K_F}(F)$. K_F is generated by the user performing a key generation protocol base on BLS signature [8] with KGS. Since the protocol is secure, that is, for any p.p.t adversary, if it does not own F, it cannot know K_F. At the same time, we use AES as the encryption algorithm E, which is a secure algorithm; therefore, C_F is secure. That is, for any p.p.t adversary who does not own F, it cannot get F from C_F.

7.3. The Online Deduplication Oracle Attack Analysis. In EFDSP, when a user u_i uploads a file F to CS, u_i uses its own permission vector and the public key of PS to generate an encrypted file tag. After that, only when a user that its permission level is equal to or higher than that of u_i upload the same file, can CS perform the file deduplication. Assuming an adversary A that its permission level is lower than that of u_i uses the file deduplication of CS to launch the file online deduplication oracle attack, it first needs to forge some files against F and then ask the PS to generate some encrypted file query tags for these files. PS uses its own private key, the permissions vector of A, and these forged files to generate some encrypted file query tags and gives these tags to A. A sends these query tags to CS and then observes whether CS performs file deduplication for the upload files to get information about F. Due to the fact that the permission level of A is lower than that of u_i, CS will not perform file deduplication for these upload files. It will ask A to upload these files. In the end, A cannot get any information about F from CS. So EFDSP can prevent adversary A from launching online deduplication oracle attack.

7.4. Comparison with SADS. Since SADS is the only existing encrypted file deduplication scheme with permission, we will compare EFDSP with SADS from the following aspects.

(i) In SADS [1], each permission is represented by a private key, and if a user has n permissions, it needs to keep n private keys secretly. However, in EFDSP, the user permissions are managed by a permission server, and the user only needs to store its own permission vector and the public key of the permission server.

(ii) In SADS, when a user uploads a file F or queries a duplication file, if the user is assigned n permissions, the system needs to use n private keys to generate n encrypted file tags for the file. So the space complexity of the network traffic of this scheme is $O(n)$. In EFDSP, the encrypted file tag of F is $FT_F = (C_1, C_2, C_{3,1} \ldots, C_{3,n}, C_{4,1} \ldots, C_{4,n}, C_5, C_6)$, so when a user uploads a file F, the space complexity of its network traffic is $O(n)$. However, the query tag of F is $QFT(H(F)) = ((T_1, T_2, T_3, T_4, T_5), tq, H(F))$, which has nothing to do with the number of permissions n. So when a user queries the duplication file of F, EFDSP requires the constant network traffic.

(iii) SADS has a security weakness, while EFDSP has overcome the security weakness. We use the example of the attack against SADS in Section 1 to show how EFDSP can prevent such attack. In EFDSP, it uses a 5-bit vector to represent a permission. The first bit of the vector represents the permission of department A, the second bit represents the permission of department B, the third bit represents the permission of financial management, and the fourth bit and the fifth bit are reserved and it codes them to be 1. Mike has permissions for department A and department B, and because Mike is also responsible for financial management and it has the permission of the finance department, so his permission vector is 00011. Bob is the employee of department B, his permission vector is 10111. If Mike uploads the payslip file of Alice to the cloud storage, Mike uses the permissions vector 00011 and the public key of the permission server to generate the encrypted file tag and upload the encrypted file tag and the encrypted file to the cloud storage server. Both Bob and Alice are employees of department B, Bob wants to know the salary of Alice. Since the payslip file has a fixed format and it is a kind of small entropy file, Bob knows the file format or may even have such a file format in his hand, i.e., Bob has his own payslip. He also knows that the salary of Alice should be between 4000 and 4100, he just does not know the exact salary data of Alice. Bob can set the salary item to $4000, 4001, \ldots 4100$, and forge 100 payslip files, then Bob uploads the 100 files to the cloud storage respectively to perform the file deduplication in the cloud storage to launch online deduplication oracle attack. However, due to the use

of EFDSP, when he needs to upload these files, it wants to get some query tags for these uploaded files and upload the query tag to the cloud storage server. According to EFDSP, since the permission level of Bob is lower than that of Mike, even if Bob uses the same file of Mike to generate the query tag, the cloud storage server does not perform file deduplication due to the permission level mismatch, Bob needs to upload all the 100 files to the cloud storage, so Bob does not know which file in his uploaded 100 files is the specific file; that is, Bob does not know the wage information of Alice.

8. Experiments

The experiment system is composed of four PCs, which simulate the client, the permission server, the key generation server, and the cloud storage server. All PCs are interconnected through a 100Mbps Ethernet network. The CPU in the PCs is Intel® Pentium® Dual E2160 1.68GHZ and the RAM is 4GB. The disk of these PCs is Western Digital Caviar Se hard drive that has a 320GB capacity, 7200 rpm with 8 MB cache. All experiments are performed in Fedora 8.0 with kernel 2.6.23.1. The cipher operation is implemented by invoking the OpenSSL cryptography library (0.98g) and the pairing-based cryptography (PBC) library. The key length of AES is 128 bits and the security parameter of the bilinear pairing is 80 bits.

We use txt, doc, and mp3, three kinds of files, as the test data set in the experiment, which is shown in Table 1. We test the computation costs of the encrypted file tag generation, query tag generation, file encryption, duplication file check, and file transmission in EFDSP. We conduct experiments on file size, file number, file duplication rate, and the user number with the same permission four aspects to analyze the performance in EFDSP, and all the experimental results are the average values of 10 experiments.

(1) The Performance Effect of File Size on EFDSP. As the file size will affect the encrypted file tag generation and file encryption in the deduplication scheme, we first test the performance effect of file size on EFDSP. We upload 10 files which have different sizes and then record the time spent. We upload 7 files of different sizes in the file set 1 and file set 2 and record the time spent in each step. As the seven files are different, CS does not perform deduplication; the results are shown in Figure 4. From the figure, we can see that file size has a great effect on the key generation, the encrypted file tag generation, and the encryption process, which are linear.

(2) The Performance Effect of the File Number on the EFDSP. We select 10 different files from the file set 3 to perform 10 groups of experiments; before each group of the experiment, we need to initialize the system to avoid encrypted file deduplication. In experiment group 1, we upload one file, and in the second group, we upload two files; thus, in the next experiment group, add one file per time, and in the experiment group 10, we upload all files. When each file group is uploaded, we record the time spent on each step. Figure 5 shows the effect of the file number on each step. Experiment

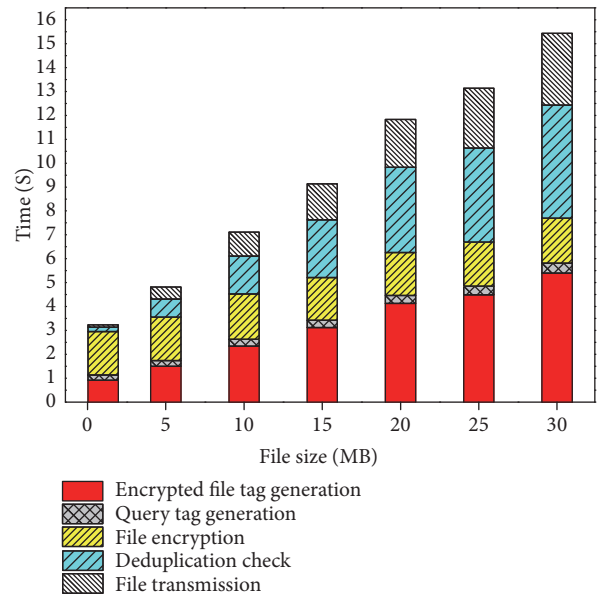

Figure 4: The performance effect of file size on EFDSP.

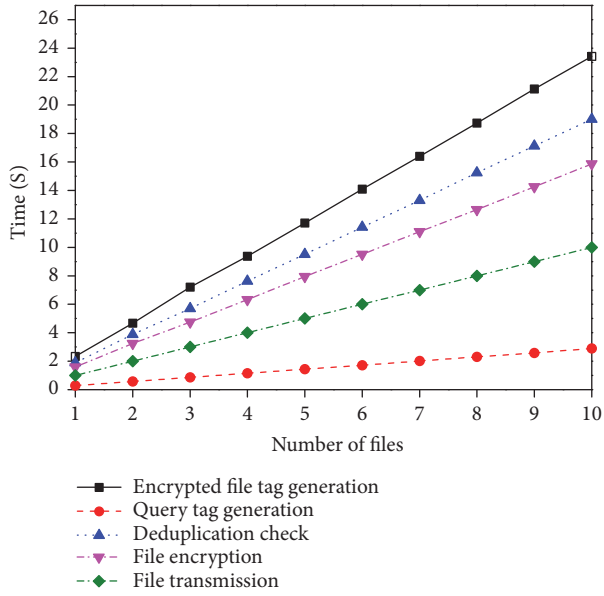

Figure 5: The performance effect of the file number on the EFDSP.

results show that the time spent in each step is in linear relationship with the file number.

(3) The Performance Effect of File Repetition Rate on EFDSP. In order to evaluate the performance effect of the file repetition rate, we divide the file set 3 into two different data test sets, each test set contains 10 files of 10MB. In each experiment, we uploaded all the files in the first data test set first. In the second file upload, we upload another 10 files, which are selected from the first data test set according to the given repetition rate, and the remaining files are selected from the second data test set, then we record the time spent on each step of the second upload. The experimental results are shown in

Table 1: The test data set.

File set	File type	explanation of the file set
File set 1	Txt	Extracts some files from Encron emails data set and package them into 15MB, 20MB, 25MB, and 30MB four files.
File set 2	Doc	Choose three files which sizes are 1MB, 5MB, and 10MB from the lab documentations.
File set 3	MP3	Choose 40 songs from 6 albums and generate 20 different 10MB files with MP3 Splitter&Joiner Pro.

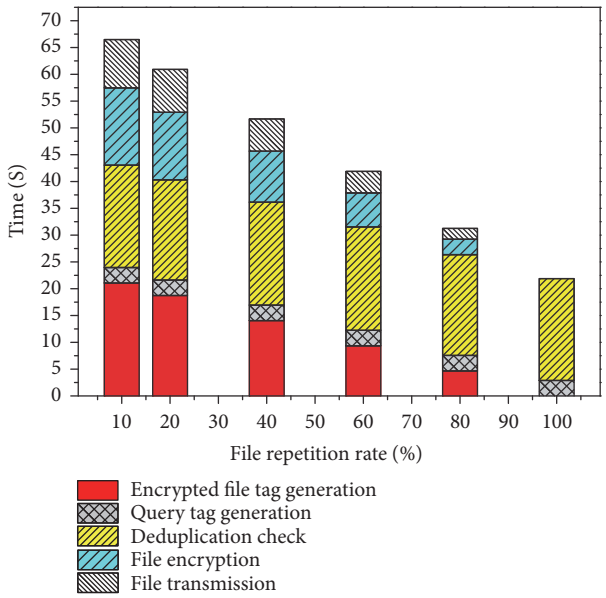

Figure 6: The performance effect of file repetition rate on EFDSP.

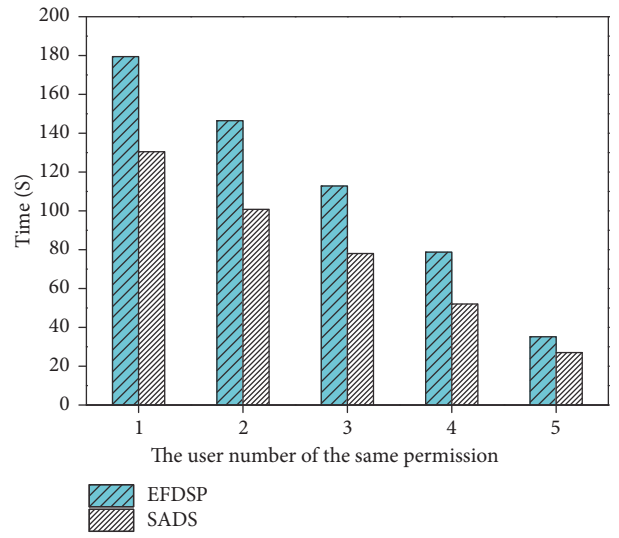

Figure 7: The performance comparison between EFDSP and SADS.

Figure 6. From the figure, we can see that the time spent by EFDSP decreases as the file repetition rate increases. When the file repetition rate reaches 100%, it is not necessary to encrypt and upload files. The time required to complete 10 files of 10*MB* is that of 32.98% when the repetition rate is 10%.

(4) The Performance Comparison between EFDSP and SADS. SADS is the only existing encrypted file deduplication with permission; to compare the performance between EFDSP and SADS, we perform the following experiment. We select 10 files of 10MB from the file set 3 as the data test set, and we set up 6 users in the experiment. We regard the first user as the upload file owner, which uploads the 10 files to the cloud storage server first, and the 10 files of the other five users are the same with the first user's file exactly. And then we configure the permission of these users on the permission server, respectively, so that one user, two users, three users, four users, and five users have the same permission with the first user, respectively, and we perform these experiments respectively. The experimental results are shown in Figure 7. The experiment results show that EFDSP is less efficient than SADS, but this gap decreases with the increasing number of authorized users; moreover, EFDSP has repaired the security weakness in SADS.

9. Related Works

Quinlan et al. proposed file deduplication to improve the storage space utilization in their document network storage system [10]. Using file deduplication to the cloud storage directly will bring some security issues to the files in the cloud storage. To demonstrate these security issues, Harnik et al. proposed three different kinds of attack methods [11]. To prevent these deduplication attacks, Halevi et al. proposed proof of ownership (POW) [12]. Some researchers have extended POW by improving its efficiency [13, 14]. However, these POWs cannot prevent attacks against small entropy files. Therefore, it is unrealistic to prevent all the deduplication attacks in the cloud storage by using the above POW.

As the same file encrypting with different keys will generate different encrypted files, the cloud storage server cannot deduplicate the encrypted file. So file encryption and file deduplication are incompatible to some extent. To solve this issue, Douceur et al. proposed convergence encryption [15]. The key of the convergence encryption is computed by using the hash function to the file that is encrypted. Different users that use the convergence encryption to encrypt the same file will generate the same encrypted file. Storer et al. proposed a block-level encrypted file deduplication scheme. The encryption key of the file block is determined by the contents of the file blocks [16], but their scheme is difficult to prevent the

brute-force attack. Bellare et al. proposed message lock encryption based on the convergence encryption [17] and designed a deduplication key generation protocol based on RSA signature [18], but the efficiency of their key generation protocol is low due to using RSA signature. Armknecht et al. designed a server-assisted key generation protocol using BLS signature that could overcome the shortcomings of the Bellare protocol [8].

Xu et al. designed a secure client encrypted file deduplication scheme for cloud storage [19]. Subsequently, Kaaniche, Stanek, and Puzio et al. proposed their encrypted file deduplication scheme for cloud storage [20–22]. To tackle the problem of encrypted file deduplication without relying on a trusted key generation server, Liu et al. [23] and Dang et al. [24] proposed their secure encrypted file deduplication scheme that does not require additional servers respectively. But their schemes do not support file permission. Li et al. proposed an encrypted file deduplication scheme that supports fuzzy search [25]. In all above-mentioned encrypted file deduplication schemes, the user participates in the encrypted file deduplication passively. Li et al. proposed an encrypted file deduplication scheme based on hybrid cloud server which supports deduplication authorization [1], but the user permission key management in their scheme is trouble; it wants relatively large storage space and network traffic, and at the same time its authorization precision is rough and there exists a security weakness.

10. Conclusions

An enterprise can reduce its business cost by storing its files to cloud storage. All files have permission in the enterprise application environment. If the cloud storage uses an encrypted file deduplication scheme without permission, it will destroy the enterprise file permission and give rise to some security issues. To solve the problem, Li et al. proposed a secure encrypted file deduplication with permission based on hybrid cloud, but its scheme has a security weakness. In this paper, we design an encrypted file deduplication model and construct an encrypted file deduplication scheme with permission (EFDSP) by using the permission vector and HVE and we optimize the performance of EFDSP. We analyze the security and the performance of EFDSP, and the results show that EFDSP satisfies the security requirements defined in Section 3.5. We implement EFDSP and conduct the performance evaluation. The experimental results show that the performance of EFDSP is slightly worse than that of SADS. However, with the increasing number of the authorized user, the performance gap decreases. At the same time, EFDSP has overcome the security weakness in SADS. Liu et al. [23] and Dang et al. [24] proposed their secure encrypted file deduplication scheme without relying on a trusted key generation server respectively, but their schemes do not support file permission in deduplication. We will introduce their technologies to our EFDSP in future work.

Data Availability

The data used to support the findings of this study are available from the corresponding author upon request.

Conflicts of Interest

The authors declare that they have no conflicts of interest related to this paper.

Acknowledgments

The research was partially funded by the Doctoral Program of Hunan Institute of Engineering (Grant No. 17RC028), the Hunan Province Natural Science Foundation (Grant No. 2016JJ3051), and the National Natural Science Foundation of China (Grant No. 61502163).

References

[1] J. Li, Y. K. Li, X. Chen, P. P. C. Lee, and W. Lou, "A Hybrid Cloud Approach for Secure Authorized Deduplication," *IEEE Transactions on Parallel and Distributed Systems*, vol. 26, no. 5, pp. 1206–1216, 2015.

[2] D. Boneh and B. Waters, "Conjunctive, subset, and range queries on encrypted data," in *Proceedings of the 4th Theory of cryptography Conference (TCC'07)*, vol. 4392, pp. 535–554, Springer, Berlin, Germany, 2007.

[3] J. Katz, A. Sahai, and B. Waters, "Predicate encryption supporting disjunctions, polynomial equations, and inner products," *Journal of Cryptology*, vol. 26, no. 2, pp. 191–224, 2013.

[4] J. H. Park, "Efficient hidden vector encryption for conjunctive queries on encrypted data," *IEEE Transactions on Knowledge and Data Engineering*, vol. 23, no. 10, pp. 1483–1497, 2011.

[5] R. S. Sandhu, E. J. Coyne, H. L. Feinstein, and C. E. Youman, "Role-based access control models," *The Computer Journal*, vol. 29, no. 2, pp. 38–47, 1996.

[6] O. Goldreich, *Foundations of cryptography: Basic applications*, Cambridge University Press, Cambridge, 2004.

[7] P. Gallagher and C. Kerry, "Fips pub 186-4: Digital signature standard (DSS)," in *National Institute of Standards and Technology (NIST)*, 2013.

[8] F. Armknecht, J. Bohli, G. O. Karame, and F. Youssef, "Transparent Data Deduplication in the Cloud," in *Proceedings of the the 22nd ACM SIGSAC Conference*, pp. 886–900, Denver, Colorado, USA, October 2015.

[9] V. Rijmen and J. Daemen, "Advanced encryption standard," in *Proceedings of Federal Information Processing Standards Publications*, pp. 19–22, National Institute of Standards and Technology, 2001.

[10] S. Quinlan and S. Dorward, "Venti: A new approach to archival storage," in *Proceedings of the 2th Conference on File and Storage Technologies (FAST'02)*, pp. 89–101, 2002.

[11] D. Harnik, B. Pinkas, and A. Shulman-Peleg, "Side channels in cloud services: Deduplication in cloud storage," *IEEE Security & Privacy*, vol. 8, no. 6, pp. 40–47, 2010.

[12] S. Halevi, D. Harnik, B. Pinkas, and A. Shulman-Peleg, "Proofs of ownership in remote storage systems," in *Proceedings of the 18th ACM Conference on Computer and Communications Security, CCS'11*, pp. 491–500, USA, October 2011.

[13] R. Di Pietro and A. Sorniotti, "Boosting efficiency and security in proof of ownership for deduplication," in *Proceedings of the 7th ACM Symposium on Information, Computer and Communications Security, ASIACCS 2012*, pp. 81-82, Republic of Korea, May 2012.

[14] J. Blasco, R. Di Pietro, A. Orfila, and A. Sorniotti, "A tunable proof of ownership scheme for deduplication using bloom filters," in *Proceedings of the IEEE Conference on Communications and Network Security (CNS '14)*, pp. 481–489, San Francisco, Calif, USA, October 2014.

[15] J. R. Douceur, A. Adya, W. J. Bolosky, D. Simon, and M. Theimer, "Reclaiming space from duplicate files in a serverless distributed file system," in *Proceedings of the 22nd International Conference on Distributed Systems*, pp. 617–624, IEEE, Austria, July 2002.

[16] M. W. Storer, K. Greenan, D. D. Long, and E. L. Miller, "Secure data deduplication," in *Proceedings of the the 4th ACM international workshop*, p. 1, Alexandria, Virginia, USA, October 2008.

[17] M. Bellare and S. Keelveedhi, "Message-locked encryption and secure deduplication," in *Advances in Cryptology (EUROCRYPT13)*, pp. 296–312, Springer, Heidelberg, 2013.

[18] M. Bellare, S. Keelveedhi, and T. Ristenpart, "Dupless: Server-aided encryption for deduplicated storage," in *Proceedings of the 22nd USENIX conference on security, USENIX Association*, pp. 179–194, 2013.

[19] J. Xu, E.-C. Chang, and J. Zhou, "Weak leakage-resilient client-side deduplication of encrypted data in cloud storage," in *Proceedings of the 8th ACM SIGSAC Symposium on Information, Computer and Communications Security, ASIA CCS 2013*, pp. 195–206, China, May 2013.

[20] N. Kaaniche and M. Laurent, "A secure client side deduplication scheme in cloud storage environments," in *Proceedings of the 2014 6th International Conference on New Technologies, Mobility and Security, NTMS 2014*, UAE, April 2014.

[21] J. Stanek, A. Sorniotti, E. Androulaki, and L. Kencl, "A Secure Data Deduplication Scheme for Cloud Storage," in *Financial Cryptography and Data Security*, vol. 8437 of *Lecture Notes in Computer Science*, pp. 99–118, Springer Berlin Heidelberg, Berlin, Heidelberg, 2014.

[22] P. Puzio, R. Molva, M. Onen, and S. Loureiro, "ClouDedup: Secure deduplication with encrypted data for cloud storage," in *Proceedings of the 5th IEEE International Conference on Cloud Computing Technology and Science, CloudCom 2013*, pp. 363–370, UK, December 2013.

[23] J. Liu, N. Asokan, and B. Pinkas, "Secure deduplication of encrypted data without additional independent servers," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, CCS 2015*, pp. 874–885, USA, October 2015.

[24] H. Dang and E.-C. Chang, "Privacy-Preserving Data Deduplication on Trusted Processors," in *Proceedings of the 10th IEEE International Conference on Cloud Computing, CLOUD 2017*, pp. 66–73, USA, June 2017.

[25] J. Li, X. Chen, F. Xhafa, and L. Barolli, "Secure deduplication storage systems supporting keyword search," *Journal of Computer and System Sciences*, vol. 81, no. 8, pp. 1532–1541, 2015.