Updatable Block-level Deduplication with Dynamic Ownership Management on Encrypted Data

Maozhen Liu[†] Chao Yang[†] Qi Jiang[†] Xiaofeng Chen[†] Jianfeng Ma[†] Jian Ren[‡]
[†]School of Cyber Engineering, Xidian University, Xi'an, Shaanxi, 710071.
Email: maozhen840@foxmail.com
{chaoyang, jiangqixdu, xfchen}@xidian.edu.cn
jfma@mail.xidian.edu.cn
[‡]Department of ECE, Michigan State University
East Lansing, MI 48824. Email: renjian@msu.edu

Abstract—**Deduplication is becoming increasingly important in that it can effectively reduce the storage space in the cloud server. Unfortunately the static file-level deduplication only supports limited data updatability and low deduplication ratio. In this paper, we show that by using updatable block-level deduplication (UBLDe) on encrypted data, all these issues can be addressed. In addition, this approach can also protect the user data privacy. However, updatable block-level dedeplication also faces several challenges. First, block-level deduplication should be achieved across different encrypted files. Second, an updatable authenticated data structure has to be designed for proof of file ownership. Finally, file ownership revocation has to be dealt with for forward secrecy. While the first challenge can be addressed by message-locked encryption, the last two challenges have not been solved yet. To address these two issues, we present a new UBLDe protocol on encrypted data with dynamic ownership management. Specifically, we design a new authenticated data structure for Proof of Ownership, named DBSL, to support update operations with low computation cost. We also propose a dynamic file ownership management scheme based on a novel lightweight MIX algorithm to protect forward secrecy. The security analysis and experimental results show that the proposed UBLDe protocol is secure and efficient.**

Index Terms—**block-level, deduplication, proof of ownership, dynamic ownership management, forward secrecy**

I. Introduction

With the rapid growth of cloud data volume, deduplication technology has become increasingly important to cloud storage. It can eliminate redundant copies of user-uploaded data to save storage space and management cost of cloud storage server. Many schemes [2]–[10] have been proposed for file-level deduplication on both encrypted and unencrypted files. However, the existing research has largely focused on file-level deduplication, which detects and eliminates duplicated files in the server. Block-level deduplication was first discussed in [11]. It showed that block-level deduplication can achieve much higher deduplication ratio. Recent research has extended file-level deduplication in cloud storage system to block-level setting [12]–[14]. In practice, we are more likely to encrypt files locally and then upload to the cloud servers in order to protect data privacy. We may also need to update our files, mostly just a few blocks. Therefore, only updatable block-level deduplication on encrypted data is needed. However, we face several challenges.

Challenge 1: How to achieve block-level deduplication?

In many cases, block keys are related to the file keys derived from the context of the files or generated randomly by the users. Therefore, the identical blocks belonging to different files could be encrypted to different ciphertexts, which makes block-level deduplication for different encrypted files virtually impossible. However, message-locked encryption [7] has been applied to address this problem in the deduplication field. It works by mapping each "message" into a key deterministically and then encrypting each "message" using such message-locked key. If the "message" is replaced with the file block, then block-level deduplication could be implemented across different encrypted files.

Challenge 2: How to design an updatable authenticated data structure for proof of file ownership?

In cloud storage system with block-level encrypted deduplication, an updatable authenticated data structure for proof of file ownership (POW) plays an important role. This structure is used not only in the proof of file ownership at client-side, but also in update of encrypted blocks stored in server. However, it is a big challenge to design an updatable authenticated data structure for POW. Chen et al. [12] first proposed a scheme to achieve block-level deduplication in ciphertext with message-locked encryption. Unfortunately, the scheme neither implements client-side deduplication nor supports encrypted block update. He et al. [10] presented a new updatable authenticated data structure based on a homomorphic authenticated tree (HAT) to support the update of encrypted blocks in file-level deduplication. However, HAT cannot maintain the balance of the structure after being updated since it can be easily transformed into a linear structure. As a result, the efficiency of the block update is extremely low. Zhao et al. [13] proposed an updatable block-level deduplication protocol (UMLE) based on Merkle Tree [5] to store the encrypted blocks as leaves nodes and the encrypted concatenation of

The General Program of The National Nature Science Foundation of China (No. 61672415, No.61672413, No.61671360), The Natural Science Foundation of Shaanxi Province (No.2017JM6054) and The 111 Project (No.B16037).

block keys as internal nodes. However, Merkle Tree only supports modification operation and cannot be used for the POW simultaneously. In addition, the storage cost for this structure doubles the size of the file, which is contrary to the purpose of deduplication. Thus, the updatable authenticated data structure for POW in block-level encrypted deduplication is still an interesting topic.

Challenge 3: How to deal with file ownership revocation in the update phase?

In an encrypted cloud storage system that supports data update, it is very important to manage file ownership dynamically. In practice, it is a difficult problem to prohibit the revoked file owners from accessing and decrypting the original file, which is called forward secrecy. Wen et al. [15] proposed a reliable key management scheme with dynamic updates. When file ownership revocation has happened, a file owner is required to download the ciphertext of file, re-encrypt it with a new key and upload the new ciphertext to server again. Hur et al. [16] presented a scheme for dynamic file ownership management in cloud storage. In this scheme, if a file owner is deleted from the file-sharing group, the server is required to decrypt the ciphertext of the original file with old group key and re-encrypt it with a new group key. Then the new group key is distributed to the remaining file owners. Both of these schemes are very inefficient in communication and computation for forward secrecy, especially for large files. Li et al. [14] introduced a rekeying scheme for encrypted deduplication storage. The scheme cannot protect forward secrecy because server only re-encrypts the 'stub' part without changing the ciphertext of blocks in the revocation phase. Thus, the revoked client can still decrypt the original file blocks with the help of old hash key. By now, no practical schemes existing that can securely and efficiently revoke the file ownership with forward secrecy.

To address these challenges, a new protocol updatable block-level deduplication (UBLDe) is proposed on encrypted data that can achieve dynamic management of file ownership with forward secrecy. In our scheme, a dynamic balanced skip list (DBSL) is designed as an updatable authenticated data structure for POW. DBSL is able to support three types of update operations with high efficiency. It combines the superiorities of a balanced binary tree structure and a skip list structure. Based on the concept of group key management, a new dynamic management scheme for file ownership is presented, which utilizes a lightweight MIX algorithm to protect forward secrecy. MIX algorithm uses a small-scale encryption and 'mix' operations to efficiently encrypt a series of blocks. MIX can also be used to encrypt the ciphertexts of blocks stored in server with a group key for forward secrecy. In addition, we also prove in detail the security of the DBSL-based POW protocol and MIX algorithm. Theoretical analysis and the experimental evaluation demonstrate that our scheme is efficient and practical and has much lower computation and communication overhead compared to that of the existing schemes.

II. System Model and Threat Model

A. System Model

Our system model includes three entities: Key Server (KS), Storage Server (SS) and Uploader, as shown in Fig. 1.

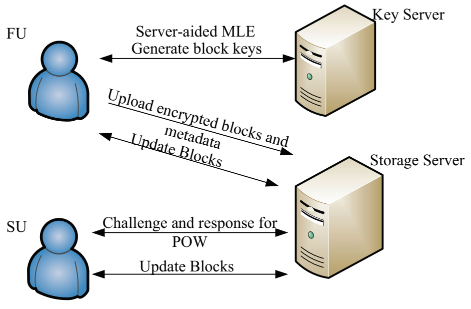

Fig. 1. The system model of UBLDe.

- Uploader: Uploaders are divided into two classes, first uploader (FU) and subsequent uploader (SU). For each uploaded file, FU denotes the user who uploaded the file to SS, while SU denotes the user who interacts with the storage server to prove the ownership of the file instead of uploading it. Only the file owner can update the encrypted blocks stored in the storage server.
- Key Server (KS): It runs RSA-OPRF algorithm [9] with first uploader to generate block keys.
- Storage Server (SS): It stores the encrypted blocks uploaded by users; interacts with subsequent uploader for Proof of Ownership; joins in data update phase; manages file ownership to protect forward secrecy.

It is worth reminding that the file ownership belongs to the user who uploaded the file as FU or has passed the proof of file ownership as SU.

B. Threat Model

UBLDe protocol is responsible for storage and updating of user-uploaded files with block-level deduplication, as well as managing file ownerships for forward secrecy. We assume that the first uploader is honest and will upload uncorrupted files. We also assume that key server can protect its private keys from being stolen by any attackers and the storage server can preserve integrity of user-uploaded files. There are two kind of attacks that we consider.

First, attackers may cheat the cloud server in order to obtain the ownership of file that has been uploaded. Second, attackers may illegally access the file stored in the cloud server after the file ownership has been revoked.

III. Our Proposed Building Locks

In this section, we propose a novel structure of DBSL and a new lightweight MIX algorithm.

A. Dynamic Balanced Skip List

1) Overview: To implement an efficient and updatable deduplication at client side, we design a novel authenticated data structure called dynamic balanced skip list (DBSL). DBSL is a special skip list consisting of internal nodes and basic-level nodes. Its most prominent characteristics are updatablity and balanceablity. A DBSL structure with four nodes in the basic level is shown in Fig.2.

Here are some important definitions (Node E denotes an internal node and Node C denotes a basic-level node in DBSL).

- The subnode of the node E: the node which is in the sub linked list pointed by node E's *below* pointer. Thus, node E is named as the father node of its subnodes.
- The father-class node of node C: the node which can reach the node C by following its *below* pointer.
- The sibling node of the node E/C: the node which is located in the same linked list with node E/C.

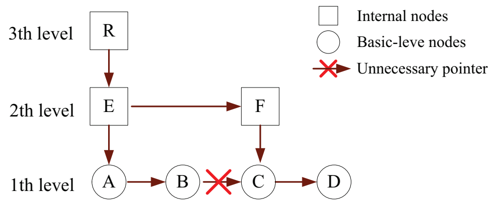

Fig. 2. Building DBSL with nodes A, B, C and D in basic level.

For example, the node C in the basic level is shown in Fig. 2, of which the father node is F, the father-class nodes is the set of {R, F} and the sibling node is D.

In Fig. 2, each node consists of a five-tuple vector $V_i = (\nu, l, s, after, below)$, ν denotes the hash value of the node, l denotes the number of the reachable nodes in basic level by the *below* pointer, s denotes the number of subnodes. $after$ and $below$ denote the pointer which points to the next sibling node, and the pointer which points to the first subnode counted from left side of its sub linked list, respectively.

The calculation formulas of building DBSL from file blocks are listed as follows.

For the i-th node in basic level: $\nu_i = h(m_i\|h(m_i))$, $l_i = 1, s_i = 0$, where m_i denotes the i-th file block. For each internal node: $\nu = h(\nu_{sb_1}\|...\|\nu_{sb_s})$, $l = \sum_{j=1}^{s} l_{sb_j}$, $s = 2$ *or* $s = 3$ where sb_j denotes the j-th subnode counted from left side of its sub linked list. ($h(\cdot)$ denotes a hash function, such as SHA256.)

2) Building DBSL: The building DBSL algorithm(alg) is defined as $DBSL_F \leftarrow building\,DBSL(n, \{\nu_i\})$.

a) Build a single linked list as the basic level with nodes in the sequence of the corresponding blocks. In basic level, one node corresponds to one block in the file.

b) For each upper level, generate one father node by every two nodes in the sub level sequentially from left to right. The last father node is generated by the last three nodes in the sub level, if the number of nodes in the sub level is odd.

c) Delete all of the unnecessary $after$ pointers in DBSL which will never be accessed in the searching process.

3) Father-class Nodes and Sibling Nodes Search: The related searching algorithms are defined as $Fa \leftarrow fathers(i, DBSL)$, $\varphi \leftarrow siblings(Fa, DBSL)$.

a) Searching starts from the root node and follows its *below* pointer down to sub level.

b) In each level except the basic level, judge whether the current node is a father-class node satisfying $l \geq i$, if it is, then search down to the sub level by following its *below* pointer; otherwise, update $i \leftarrow i - l$ and follow its $after$ pointer to judge the next sibling node under the same condition.

c) Continue to search till the target node in the basic level, and all father-class nodes Fa have been found out.

d) Find out all sibling nodes φ of the father-class nodes Fa in DBSL.

4) Node Update: The update operations of DBSL include modification, insertion and deletion for the node in the basic level. We assume there is a target node $TN_\imath$ with the index $\imath$ in the basic level. The update algorithm is defined as $DBSL' \leftarrow UpdateNode(\imath, TN_\imath, DBSL, OP)$.

Modifying the target node (OP=modify):

a) Run alg. $Fa \leftarrow fathers(\imath, DBSL)$ to search the father-class nodes.

b) Update the values of the target node and the values of each father-class node from button to top with the help of all its subnodes.

For example, to modify the node C, as shown in Fig. 2, we first find out the father-class nodes $\{R, F\}$, then update node F with node C and D, update node R with node E and F.

Inserting a new node in front of the target node (OP=insert):

a) Run alg. $Fa \leftarrow fathers(\imath, DBSL)$ to search the father-class nodes.

b) Insert the new node in front of the target node in the same linked list, so they have the same father-class nodes now and the number of subnodes s_f belonging to their father node is increased by 1, $s_f \leftarrow s_f + 1$.

c) If $s_f = 3$, then update the values of each father-class node in upper levels with the help of all its subnodes.

d) Else $s_f = 4$, then the last two subnodes are divided from the linked list to generate a new father node inserted behind the original father node. Update the values of the original father node with the remaining subnodes. So the number of subnodes s_{gf} for the father node of the original father node is increased by 1, $s_{gf} \leftarrow s_{gf} + 1$.

e) Update the remaining father-class nodes in upper levels as step c) or step d).

For example, inserting node G in front of the node C is shown in Fig3.(a), and inserting node H in front of the node D is shown in Fig.3(b).

Deleting the target node (OP=delete):

a) Run alg. $Fa \leftarrow fathers(\imath, DBSL)$ to search the father-class nodes.

b) Delete the target node from the linked list and the number of sub nodes s_f for its father node is decreased by 1, $s_f \leftarrow s_f - 1$.

c) If $s_f = 2$, then update the values of each father-class node in upper levels with the help of all its subnodes.

d) Else $s_f = 1$, then execute the insertion operation steps b), c), d) and e) to insert the remaining single subnode into the next sibling linked list or the front sibling linked list, if the next sibling linked list does not exist. The sequence of nodes will not be changed. Delete the father node which has lost all subnodes.

For example, delete node G is shown Fig.4(a), and delete node C is shown in Fig.4(b).

After update operation, each internal node owns two or three subnodes which keeps DBSL balanced.

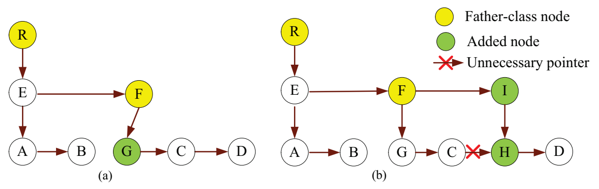

Fig. 3. Insertion operation.

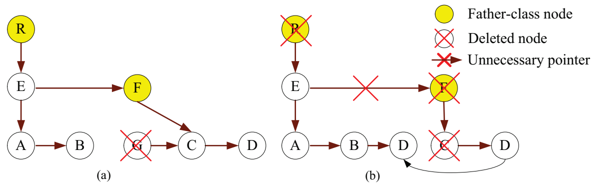

Fig. 4. Deletion operation.

B. Mixes Algorithm

1) Intuition: In order to prohibit the revoked file owners from decrypting the ciphertexts of original file blocks correctly, the storage server is required to encrypt these ciphertexts with a group key, which is owned only by legal file owners. Therefore, the revoked file owners will not be able to decrypt it. Therefore, forward secrecy is guaranteed. A novel MIX algorithm is proposed to encrypt these ciphertexts efficiently. Unlike other encryption algorithms, such as DES and AES, MIX is a lightweight algorithm, which executes small-scale encryption and mix operations (insert some random bits in random positions and run exclusive-or (XOR) operation) with a group key and a public block to encrypt a series of encrypted blocks belonging to a file. Thus, it is much more efficient than that of the existing encryption algorithms.

2) MEA and MDA: MIX includes two sub algorithms $MEA(\cdot)$ and $MDA(\cdot)$. The execution steps are shown in Fig. 5.

$MEA(k_G, S, \{r_i\}, \{C_i\}) \rightarrow \{C_i''\}$:	$MDA(k_G, S, \{C_i''\}) \rightarrow (\{r_i\}, \{C_i\})$:
$S' \leftarrow En(k_G, S)$	$S' \leftarrow En(k_G, S)$
For the i-th block:	For the i-th block:
$\alpha_i \leftarrow H(k_G \| i)$	$C_i' \leftarrow C_i'' \oplus S'$
$C_i' \leftarrow Embed(\alpha_i, r_i, C_i)$	$\alpha_i \leftarrow H(k_G \| i)$
$C_i'' \leftarrow S' \oplus C_i'$	$(C_i, r_i) \leftarrow Extract(\alpha_i, C_i')$

Fig. 5. MEA and MDA algorithms.

In MEA/MDA algorithm, S denotes a public block that can be accessed by each user; k_G denotes a group key; $\{r_i\}$ denotes the set of random strings; S' denotes the ciphertext of S; $\{C_i\}$ denotes the set of encrypted blocks stored in the server; $\{C_i'\}$ denotes the set of bit-embedding blocks; $\{C_i''\}$ denotes the set of mixed blocks.

$H(\cdot)$: $\{0,1\}^* \rightarrow \mathbb{Z}_N$ is a hash function [9]. It is used to hash a message to an element of $\mathbb{Z}_N$. In $Embed(\cdot)/Extract(\cdot)$, $|r_i|$ integers are generated by a random seed α_i, representing random positions used to embed/extract each bit of the random string r_i into/from C_i/C_i'. $En(\cdot)$ denotes an encryption algorithm, such as AES, and $\oplus$ denotes XOR operation.

3) Security Analysis for MEA: Assume there is a probability polynomial time (PPT) attacker A^*, who knows $S, \{r_i\}, C_1, \{C_i''\}$ and block keys $\{k_i\}$. $\{k_i\}$ denotes the set of block keys. We think that $\{C_i'\}$ is not secure, because A^* can separate r_i from C_i' to obtain C_i by statistical methods. Then, A^* decrypts $\{C_i\}$ with $\{k_i\}$ to regain file F. Therefore, the probability of A^* winning $\Pr[A^*win1] = \Pr[A^*getsC_i{}']$, where $\Pr[A^*getsC_i{}'] = \Pr[A^*getsS']$ in MEA. If A^* can correctly guess C_1' or the group key k_G, then A^* can obtain S', for $En(k_G, S) = S' = C_1' \oplus C_i''$. Thus, we analyse the probability of guessing C_1' correctly in Game1.

Game1: There are two binary strings k and b. Each bit in k is embedded into $|k|$ random positions in b to obtain a new binary string c. The length of string c is $|c| = |k| + |b|$. If string c is identical to the given string c^*, then A^* wins; otherwise, A^* fails. The probability of A^* winning is $\Pr[A^*win1] = \frac{1}{C_{|b|}^{|k|}} \leq \frac{1}{(\frac{|b|}{|k|})^{|k|}}$. $\Pr[A^*getS'] = max\{\Pr[A^*getsk_G], \Pr[A^*win1]\} = max\{\frac{1}{2^{256}}, \frac{1}{2^{1024}}\} \leq \varepsilon(\lambda)$, where $|k| = |k_G| = 256bits$, $|b| = 4kbits$, $\varepsilon(\lambda)$ denotes a negligible function. Thus, MEA is secure.

IV. The UBLDe Scheme

Base on DBSL structure and MIX algorithm, we present a new protocol UBLDe for updatable block-level deduplication on encrypted data with dynamic ownership management. It consists of five phases: pre-process, upload, deduplication, download and update.

A. The Pre-process Phase

The client runs the initialization algorithm to obtain file tag T_F and file key k_F. i.e.$(T_F, k_F) \leftarrow Init(F, 1^k)$:$k_F \leftarrow h(F), T_F \leftarrow h(k_F)$, where 1^k denotes a security parameter.

Then, the client announces that it has a certain file via T_F. If the file does not exist in server, the client goes into the upload phase. Otherwise, the client goes into the deduplication phase.

B. The Upload Phase

The first uploader FU runs the encoding algorithm$(\{C_i\}, \{\nu_i\}, C_k) \leftarrow Encode(k_F, F)$.

FU executes:

a) Run $RSA - OPRF$ [9] algorithm with key server to generate block keys$\{k_i\}$;

b) Compute authenticated value of each block $\nu_i = h(m_i||h(m_i))$, $1 \leq i \leq n$, n denotes the number of blocks;

c) Compute the ciphertext of each block $C_i = En(k_i, m_i)$ and the ciphertext of the concatenation of block keys $C_k = En(k_F, k_1||...||k_n)$, upload $\{C_i\}$, $\{\nu_i\}$, C_k to storage server SS.

Then SS executes:

a) Compute the tag of each block $T_i = hash(C_i)$, detect and delete duplicate blocks with $\{T_i\}$;

b) Build $DBSL_F \leftarrow buildingDBSL(n, \{\nu_i\})$;

c) Build file-sharing group G_F and add the FU to G_F;

d) Generate a group key $k_G \leftarrow \{0,1\}^k$ to manage the ownership of file F, computer the public key and master secret key $(pk, mk) \leftarrow Setup(1^k)$ [17], build an access control tree for file F and encrypt group key $C_{k_G} = CPABE.En(k_G, ACT_F, pk)$ [17].

C. The Deduplication Phase

The subsequent uploader SU runs the deduplication protocol $res \in \{0,1\} \leftarrow Dedup < SU(F), SS(n, DBSL_F) >$ to prove the ownership of the file F.

SS executes:

a) Randomly generate two positive integers (a, b) as challenge Q and send it to SU;

b) Compute the challenged index $I_i = Random(a)$, $1 \leq i \leq b$, the sibling nodes $\varphi = siblings(fathers(\{I_i\}))$, and the hash value of root node ν_R in $DBSL_F$.(Random($\cdot$) denotes a pseudo-random generator).

SU computes the challenged indexes as SS does, and send back the set of authenticated values for each challenged block $\{\nu_{I_i}\}$, $\nu_{I_i} = h(m_{I_i}||h(m_{I_i}))$.

SS recomputes the hash value of root $\nu_{R'}$ with $\{\nu_{I_i}\}$ and φ. If $\nu_{R'} = \nu_R$ then outputs 1 , adds SU to group G_F and update group key k_G (It is described detailedly in the update phase), otherwise outputs 0.

D. The Download Phase

Storage server SS encrypts the ciphertext of blocks with $MEA(\cdot)$ algorithm and then file owner FO runs the decryption algorithm $F \leftarrow Decrypt(k_F, sk, C_{k_G}, \{C_i{}''\})$ to recover file F.

SS executes:

a) Divide C_k into n strings $\{C_{k_i}\}$ with equal length, encrypt the ciphertext of blocks by alg. $\{C_i{}''\} \leftarrow MEA(k_G, S, \{C_{k_i}\}, \{C_i\})$;

b) Download the processed blocks $\{C_i{}''\}$ and the encrypted group key C_{k_G} to client.

Then FO executes:

a) Request SS for the newest master secret key mk and recompute secret key $sk \leftarrow keygen(\omega, mk)$ [17] with his attributes ω, if mk has been updated;

b) Decrypt C_{k_G} to get $k_G \leftarrow CPABE.De(C_{k_G}, sk)$, decrypt $\{C_i{}''\}$ to get $(\{C_{k_i}\}, \{C_i\}) \leftarrow MDA(k_G, S, \{C_i{}''\})$, recover $C_k = C_{k_1}\| \ldots \|C_{k_n}$, decrypt C_k to get $k_1\| \ldots \|k_n = De(k_F, C_k)$ (k_F is stored locally), decrypt C_i to get block $m_i = De(k_i, C_i)$, recover file $F = m_1\| \ldots \|m_n$.

E. The Update Phase

The file owner FO can arbitrarily update the file F. For each new block, FO firstly generates the new block key with the help of key server as the upload phase, and then runs update protocol with storage server SS $res \in \{< DBSL_{F'}, C_k{}', C_i{}' >, \perp\} \leftarrow Update < FO(k_F, i, m_i, k_i, k_F{}', OP), SS(C_k, DBSL_F) >$. SS downloads the encrypted concatenation of block keys C_k to FO.

Then FO executes:

a) Decrypt C_k with file key k_F, updates it with new block key k_i, and encrypt it with the new file key $k_F{}'$ to get $C_k{}'$;

b) Compute the authenticated value ν_i of the new block and the ciphertext of new block $C_i{}' = En(k_i, m_i)$, then upload the index of the updated block i, $C_k{}'$, ν_i, $C_i{}'$ and the update operation OP to SS.

SS executes:

a) Compute the tag of new block, detect and delete duplicate blocks with its tag;

b) Generate a new node TN_i with ν_i and run $DBSL_{F'} \leftarrow UpdateNode(i, TN_i, DBSL_F, OP)$;

c) Remove the ownership of FO from the G_F to the new file-sharing group $G_{F'}$. For each updated group $\{G_F, G_{F'}\}$, the group key is re-generated. The ACT, pk and mk are also updated. Finally, the new group key should be encrypted by CPABE algorithm as described in the upload phase.

V. Security Analysis

In this section, we analyze the security of UBLDe. As far as we known, server-aided MLE has provided semantic security [8], holding even for predictable messages, which protects blocks with low entropy from violent attacks in uploading phase. Hence, we mainly argue about the security of DBSL-based POW protocol.

A. Security Model Game

In the game G$_A$(t,s), participants includes user U, storage server SS and PPT adversary A. There are some system parameters t, s, λ, where λ denotes a security parameter, t denotes the min entropy of file F, and s denotes a constant parameter, $t > s > \lambda$, $\frac{s}{t} > \frac{1}{2}$. Assume that adversary A can at most get $(t - s)$ bits information of the file F from game G$_A$(t,s). Without loss of generality, it is assumed that the file F is extracted from any distribution of $\{0,1\}^M$, and the minimum entropy of file F is not less than t. M denotes the length of file F, and $M \geq t$.

The game is divided into two phases, the learning phase and the verification phase.

The learning phase

a) O^F runs a probability coding algorithm based on file F. A queries Random Oracle O^F and gets four parameters (T_F, C, a, b). T_F denotes the tag of file that can identify file F, C denotes the set of all encrypted file blocks and (a, b) denote two random numbers.

b) A runs as SS, takes two integers (a, b) as input and interacts with a user U in the DBSL-based POW phase. Then, A gets the response V_{q1} from U and the internal state y. y indicates whether the server SS accepts U's response for file ownership, if $y = 1$ then SS accepts U as a file owner, otherwise SS rejects U.

c) A runs as U, inputs(a, b), V_{q1} and C. Then A interacts with SS in the DBSL-based POW phase. A also queries O^F for response V_{q2}.

The verification phase

A can at most get $t - s$ bits information of file F in the learning phase. Hence, A constructs a file F^* based on the $t - s$ bits information, runs DBSL-based POW protocol with SS.

Definition 1. DBSL-based POW is security, if for any PPT adversary A and any positive integer λ, the probability of A winning $Pr[A_{wins}] =\leq \varepsilon(\lambda)$, $\frac{s}{t} > \frac{1}{2}$, where $\varepsilon(\lambda)$ is a negligible function.

B. Security Proof

Theorem 1: Let $hash(\cdot)$ be a collision resistant hash function, the symmetric encryption algorithm $En(\cdot)$ and the server-aided MLE provide semantic security, DBSL-based POW protocol is security.

Proof: The adversary A does not have whole file F. Hence, there are three possible ways in which A may win the game $G_A(t,s)$.

a) A obtains at most $t - s$ bits information of file F in learning phase. If the challenged blocks all cover the A's known fields in F, then A can pass the verification phase. The probability of this case is $Pr[A_{win1}] = (\frac{t-s}{t})^b = (1-c)^b < \frac{1}{2^b}$, where $c = \frac{s}{t} > \frac{1}{2}$, b denotes the number of the challenged blocks. If $b \geq 256$, $Pr[A_{win1}]$ is negligible.

b) A constructs x_i to imitate the challenged block b_i, st $hash(x_i||hash(x_i)) = hash(b_i||hash(b_i))$, $i \in I$, where I denotes the set of the challenged indexes. If $x_i \neq b_i$, then it contradicts with that $hash(\cdot)$ is collision resistant. If $x_i = b_i$, then the probability of A winning is $Pr[A_{win2}] = \frac{1}{2^{|b_i|}}$, where $|b_i|$ denotes the size of the challenged block. When $|b_i| \geq 4kb$, $Pr[A_{win2}]$ is negligible.

c) A may intercept the encrypted block c_i from network. If A accurately guesses the key k_i to decrypt c_i, then A can compute the authenticated value ν_i with the context of this block. The probability of A winning is $Pr[A_{win3}] = \frac{1}{2^{|k_i|}}$, where $|k_i|$ denotes the size of block key. When $|k_i| \geq 256$, $Pr[A_{win3}]$ is negligible.

$Pr[A_{wins}] = Pr[A_{win1}] + Pr[A_{win2}] + Pr[A_{win3}] < \varepsilon(\lambda)$, where $\varepsilon(\cdot)$ is a negligible function. Hence the DBSL-based POW scheme is proved to be security. ■

VI. Performance Evaluation

In this section, we evaluate the performance of UBLDe theoretically and experimentally.

A. Theoretical Comparison

Table 1 summarizes the performance of our scheme UBLDe in comparison with related schemes, where n denotes the number of blocks, b denotes the number of challenged blocks, $|m|$ denotes the size of one block, and $|h|$ denotes the size of one hash value. From the table, we observe that both UBLDe and DeyPos [10] support update operations. Furthermore, the asymptotic performance of UBLDe is better than other schemes expect [6], which only provides weak security guarantee.

TABLE I
COMPARISON WITH RELATED SCHEMES

Related Schemes	**Deduplication on the client-side**			**Is Updatable**
	Client	***Server***	***Comm***	
MK [5]	$O(n^2)$	$O(blogn)$	$O(blogn) + b\|m\|$	No
Xu [6]	$O(n)$	$O(1)$	$O(1)$	No
DeyPos [10]	$O(n)$	$O(blogn)$	$O(blogn) + \|m\|$	Yes
UBLDe	$O(n)$	$O(blogn)$	$b\|h\|$	Yes

B. Experimental Comparison

1) Experiment Environment: We implemented our scheme by C language on a computer with an Intel 3.2GHz CPU and 8GB DDR4 memory. Each data point is the average of twenty experiment results. The experiment includes two phases, the upload phase and the deduplication phase. The cost in update phase is similar to the cost in deduplication phase, hence, we don't present the cost in update phase for the limitation of space. We choose the size of key is 256bits, SHA256 as hash function, and AES256 as encryption algorithm.

Because the HAT structure in DeyPos [10] is essentially a binary tree, like Merkle Tree (MK) [5], except that HAT is able to update its leaf node with homomorphic algorithm. Therefore, we compare our scheme with the Merkle Tree based schemes in upload phase and deduplication phase.

2) Result Analysis:

The upload phase: We first evaluate the cost of building structures on the server side. Fig.7 presents the time to construct the Merkle Tree and the DBSL with different file sizes and blocks. The construction time of DBSL is less than that of the Merkle Tree in same file size, especially when the size of file is larger than 256MB. Because the time consumption grows logarithmically with the number of blocks, the different block sizes of a file makes little influence on the construction of both MK and DBSL.

The deduplication phase: Next, we evaluate the communication cost of 4KB block in deduplication phase as shown in Fig.8, when the number of challenged blocks are 30 and 120, respectively. With the same number of challenged blocks and the same file size, the communicational cost is much less for DBSL-based POW, in which client only needs to send back the authenticated values of the challenged blocks. For Merkle Tree-based POW, client needs to send back the challenged blocks and the sibling nodes of the challenged nodes in the Merkle Tree, which leads to more cost in communication.

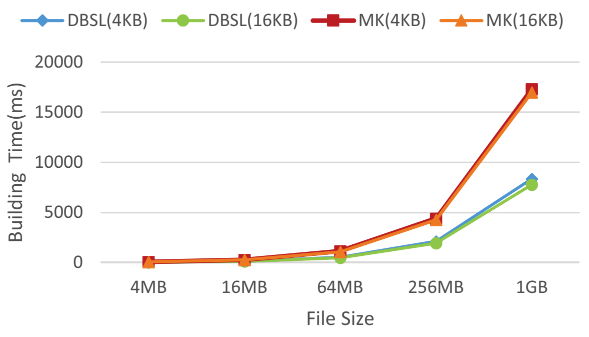

Fig. 6. Building time in different file sizes.

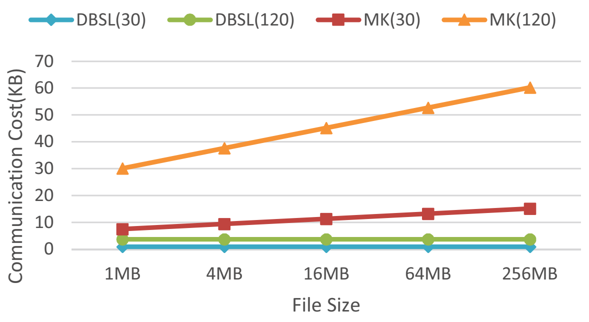

Fig. 7. Communication cost of 4KB block in the deduplication phase, when the number of challenged blocks are 30 and 120, respectively.

VII. Conclusion

In this paper, we proposed a novel DBSL structure and a lightweight MIX algorithm to implement update operations of encrypted blocks stored in the server and deal with ownership revocation for forward secrecy, respectively. Based on DBSL and MIX, we present the first updatable block-level deduplication protocol on encrypted data with dynamic ownership management and prove its security in detail. The theoretical analysis and experimental results demonstrate that our proposed scheme is efficient, especially when the file size and the number of challenged blocks are large.

References

[1] 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.
[2] R. D. Pietro and A. Sorniotti, "Boosting efficiency and security in proof of ownership for deduplication," in *ACM Symposium on Information, Computer and Communications Security*, 2012, pp. 81–82.
[3] Q. Zheng and S. Xu, "Secure and efficient proof of storage with deduplication," in *ACM Conference on Data and Application Security and Privacy*, 2012, pp. 1–12.
[4] C. Yang, J. Ren, and J. Ma, "Provable ownership of files in deduplication cloud storage," *Security & Communication Networks*, vol. 8, no. 14, pp. 2457–2468, 2015.
[5] S. Halevi, D. Harnik, B. Pinkas, and A. Shulman-Peleg, "Proofs of ownership in remote storage systems," in *ACM Conference on Computer and Communications Security*, 2011, pp. 491–500.
[6] J. Xu, E. C. Chang, and J. Zhou, "Weak leakage-resilient client-side deduplication of encrypted data in cloud storage," in *ACM Sigsac Symposium on Information, Computer and Communications Security*, 2013, pp. 195–206.
[7] M. Bellare, S. Keelveedhi, and T. Ristenpart, "Message-locked encryption and secure deduplication," in *International Conference on the Theory and Applications of Cryptographic Techniques*, 2013, pp. 296–312.
[8] W. K. Ng, Y. Wen, and H. Zhu, "Private data deduplication protocols in cloud storage," in *Acm Symposium on Applied Computing*, 2012, pp. 441–446.
[9] M. Bellare, S. Keelveedhi, and T. Ristenpart, "Dupless: server-aided encryption for deduplicated storage," in *Usenix Conference on Security*, 2013, pp. 179–194.
[10] K. He, J. Chen, R. Du, Q. Wu, G. Xue, and X. Zhang, "Deypos: Deduplicatable dynamic proof of storage for multi-user environments," *IEEE Transactions on Computers*, vol. 65, no. 12, pp. 3631–3645, 2016.
[11] D. T. Meyer and W. J. Bolosky, "A study of practical deduplication," in *Usenix Conference on File and Stroage Technologies*, 2011, pp. 1–1.
[12] R. Chen, Y. Mu, G. Yang, and F. Guo, "Bl-mle: Block-level message-locked encryption for secure large file deduplication," *IEEE Transactions on Information Forensics & Security*, vol. 10, no. 12, pp. 2643–2652, 2015.
[13] Y. Zhao and S. S. M. Chow, "Updatable block-level message-locked encryption," in *ACM Asia Conference on Computer and Communications Security*, 2017.
[14] J. Li, C. Qin, P. P. C. Lee, and J. Li, "Rekeying for encrypted deduplication storage," in *Ieee/ifip International Conference on Dependable Systems and Networks*, 2016, pp. 618–629.
[15] M. Wen, K. Ota, H. Li, J. Lei, C. Gu, and Z. Su, "Secure data deduplication with reliable key management for dynamic updates in cpss," *IEEE Transactions on Computational Social Systems*, vol. 2, no. 4, pp. 137–147, 2016.
[16] J. Hur, D. Koo, Y. Shin, and K. Kang, "Secure data deduplication with dynamic ownership management in cloud storage," *IEEE Transactions on Knowledge & Data Engineering*, vol. 28, no. 11, pp. 3113–3125, 2016.
[17] L. Ibraimi, Q. Tang, P. Hartel, and W. Jonker, "Efficient and provable secure ciphertext-policy attribute-based encryption schemes," in *Information Security Practice and Experience, International Conference, Ispec 2009, Xi'an, China, April 13-15, 2009, Proceedings*, 2008, pp. 1–12.
[18] J. Xiong, Y. Zhang, L. Lin, J. Shen, X. Li, and M. Lin, "msposw: A multiserver aided proof of shared ownership scheme for secure deduplication in cloud," *Concurrency & Computation Practice & Experience*, no. 5, 2017.
[19] J. Li, X. Chen, M. Li, J. Li, P. P. C. Lee, and W. Lou, "Secure deduplication with efficient and reliable convergent key management," *IEEE Transactions on Parallel & Distributed Systems*, vol. 25, no. 6, pp. 1615–1625, 2014.
[20] Z. Yan, W. Ding, X. Yu, H. Zhu, and R. H. Deng, "Deduplication on encrypted big data in cloud," *IEEE Transactions on Big Data*, vol. 2, no. 2, pp. 138–150, 2016.