ISSN:2231-0711

Optimal Data Deduplication in Cloud with Homomorphic Authenticated Tree

V.Manothini [#1], K.Karthick [*2]

[#]*PG Scholar- CSE dept, Kongunadu College Of Engineering &Technology, Trichy, Tamilnadu,India*

[*]*Assistant professor-CSE dept,Kongunadu College Of Engineering &Technology, Trichy, Tamilnadu,India*

Abstract— **Cloud computing has become a new platform for personal computing. Cloud computing provides high performance computing resources and mass storage resources.Data Deduplication involves finding and removing duplication within data without compromising its integrity. A practical multi-user cloud storage system needs the secure client-side cross-user Deduplication technique, which allows a user to skip the uploading process and obtain the ownership of the files immediately, when other owners of the same files have uploaded them to the cloud server. To the best of our knowledge, none of the existing dynamic PoSs can support this technique. In this paper, we introduce the concept of deduplicatable dynamic proof of storage and propose an efficient construction called DeyPoS, to achieve dynamic PoS and secure cross-user Deduplication, simultaneously. Considering the challenges of structure diversity and private tag generation, we exploit a novel tool called Homomorphic Authenticated Tree (HAT). We prove the security of our construction, and the theoretical analysis and experimental results show that our construction is efficient in practice.**

Keywords— **Structure Diversity, Cloud Storage, Deduplication, Dynamic Proof of Storage.**

I. Introduction

Cloud storage is a model of networked enterprise storage where data is stored in virtualized pools of storage which are generally hosted by third parties. Cloud storage provides customers with benefits, ranging from cost saving and simplified convenience, to mobility opportunities and scalable service. These great features attract more and more customers to utilize and storage their personal data to the cloud storage: according to the analysis report, the volume of data in cloud is expected to achieve 40 trillion gigabytes in 2020.

Even though cloud storage system has been widely adopted, it fails to accommodate some important emerging needs such as the abilities of auditing integrity of cloud files by cloud clients and detecting duplicated files by cloud servers. We illustrate both problems below.

The first problem is integrity auditing. The cloud server is able to relieve clients from the heavy burden of storage management and maintenance. The most difference of cloud storage from traditional in-house storage is that the data is transferred via Internet and stored in an uncertain domain, not under control of the clients at all, which inevitably raises clients great concerns on the integrity of their data. These concerns originate from the fact that the cloud storage is susceptible to security threats from both outside and inside of the cloud in [1], and the uncontrolled cloud servers may passively hide some data loss incidents from the clients to maintain their reputation. What is more serious is that for saving money and space, the cloud servers might even actively and deliberately discard rarely accessed data files belonging to an ordinary client. Considering the large size of the outsourced data files and the clients' constrained resource capabilities, the first problem is generalized as how can the client efficiently perform periodical integrity verifications even without the local copy of data files.

However, dynamic PoS remains to be improved in a multi-user environment, due to the requirement of cross-user deduplication on the client-side in [15]. This indicates that users can skip the uploading process and obtain the ownership of files immediately, as long as the uploaded files already exist in the cloud server. This technique can reduce storage space for the cloud server in [10], and save transmission bandwidth for users. To the best of our knowledge, there is no dynamic PoS that can support secure cross-user deduplication.

There are two challenges in order to solve this problem. On one hand, the authenticated structures used in dynamic PoSs, such as skip list in [8] and Merkle tree in [14], are not suitable for deduplication. We call this challenge structure diversity, which means the authenticated structure of a file in dynamic PoS may have some conflicts. For instance, the authenticated structure of a file F is shown in Fig. 1a.When the file is updated to F′ , the authenticated structure stored on the server-side may turn into the structure in Fig. 1b. However, an owner who intends to upload F′ usually generates a structure as shown in Fig. 1c, which is different from the structure stored in the cloud server. Thus, the cloud server synchronize the authenticated structure. On the other hand, even if cross-user deduplication is achieved (for example, the cloud server sends the entire authenticated structure to the owner), private tag generation is still a challenge for dynamic operations. In most of the existing dynamic PoSs, a tag used for integrity verification is generated by the secret key of the uploader. Thus, other owners who have the ownership of the file but have not uploaded it due to the cross-user deduplication on the client-side, cannot generate a new tag when they update the file. In this situation, the dynamic PoSs would fail.

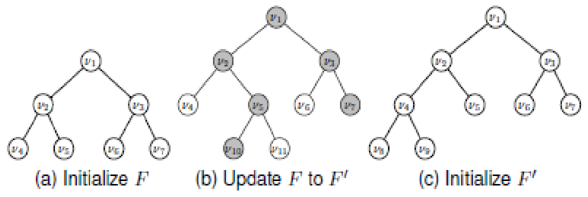

Fig. 1 An Overview of Tree-based Authenticated Structures

If we take dynamic PoS and cross-user deduplication on the client-side as orthogonal issues, we may simply combine the existing dynamic PoS schemes and deduplication techniques. Then, structure diversity is solved via deduplication scheme. For solving private tag generation, each owner can generate its own authenticated structure and upload the structure to the cloud server, which means that the cloud server stores multiple authenticated structures for each file. Also, when a file is updated by a user, the cloud server has to update the corresponding authenticated structure in dynamic PoS, and construct a new authenticated structure for deduplication. As a result, this trivial combination introduces introduces unnecessary computation and storage cost to the cloud server. Taking the combination of in [10] and in [15] as example, [10] is a dynamic PoS scheme which employs Merkle tree as its authenticated structure, and in [15] is a crossuser deduplication scheme which also employs Merkle tree as its authenticated structure. Suppose Alice and Bob independently own a file F, a Merkle tree TF is generated and stored by the cloud server for deduplication, and two Merkle trees TA and TB are generated by Alice and Bob respectively, and stored in the cloud server for PoS. When Alice updates F to F′ , the cloud server updates TA to T′ A for PoS and generates a new Merkle tree TF′ for deduplication. Thus, the number of Merkle trees grows with the version numbers and the number of owners, which is 4 (TF , T′ A , TB, and TF′) in the above example. Also, the cloud server has to generate two Merkle trees in the above example which is more time-consuming than update the Merkle trees. As a summary, existing dynamic PoSs cannot be extended to the multi-user environment.

II. Related Work

The main contributions of this paper are as follows.

1) To the best of our knowledge, this is the first work to introduce a primitive called deduplicatable dynamic Proof of Storage (deduplicatable dynamic PoS), which solves the structure diversity and private tag generation challenges.
2) In contrast to the existing authenticated structures,such as skip list in [8] and Merkle tree in [14], we design a novel authenticated structure called Homomorphic Authenticated Tree (HAT), to reduce the communication cost in both the proof of storage phase and the deduplication phase with similar computation cost. Note that HAT can support integrity verification, dynamic operations, and cross-user deduplication with good consistency.
3) We propose and implement the first efficient construction of deduplicatable dynamic PoS called Dey-PoS, which supports unlimited number of verification and update operations. The security of this construction is proved in the random oracle model, and the performance is analyzed theoretically and experimentally.

III. Deduplicatable Dynamic PoS

As discussed in Section 1, no trivial extension of dynamic PoS can achieve cross-user deduplication. To fill this void, we present a novel primitive called deduplicatable dynamic proof of storage in this section.

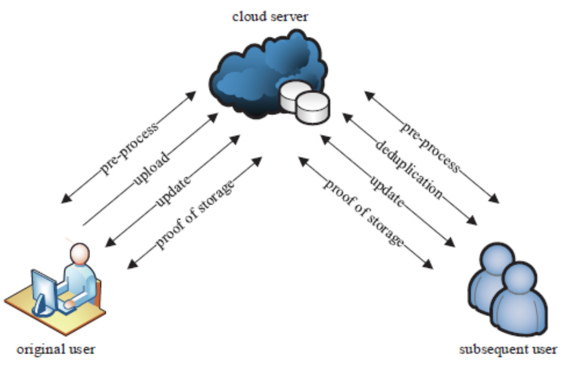

Fig. 2 The system model of deduplicatable dynamic PoS

A.System Model

Our system model considers two types of entities: the cloud server and users, as shown in Fig. 2. For each file, original user is the user who uploaded the file to the cloud server, while subsequent user is the user who proved the ownership of the file but did not actually upload the file to the cloud server. There are five phases in a deduplicatable dynamic PoS system: pre-process, upload, deduplication, update, and proof of storage.

In the pre-process phase, users intend to upload their local files. The cloud server decides whether these files should be uploaded. If the upload process is granted, go into the upload phase; otherwise, go into the deduplication phase.

In the upload phase, the files to be uploaded do not exist in the cloud server. The original users encodes the local files and upload them to the cloud server.

In the deduplication phase, the files to be uploaded already exist in the cloud server. The subsequent users possess the files locally and the cloud server stores the authenticated structures of the files. Subsequent users need to convince the cloud server that they own the files without uploading them to the cloud server.

Note that, these three phases (pre-process, upload, and deduplication) are executed only once in the life cycle of a file from the perspective of users. That is, these three phases appear only when users intend to upload files. If these phases terminate normally, i.e., users finish uploading in the upload phase, or they pass the verification in the deduplication phase, we say that the users have the ownerships of the files.

In the update phase, users may modify, insert, or delete some blocks of the files. Then, they update the corresponding parts of the encoded files and the authenticated structures in the cloud server, even the original files were not uploaded by themselves. Note that, users can update the files only if they have the ownerships of the files, which means that the users should upload the files in the upload phase or pass the verification in the deduplication.

IV. HOMOMORPHIC AUTHENTICATED TREE

A. Overview

To implement an efficient deduplicatable dynamic PoS scheme, we design a novel authenticated structure called homomorphic authenticated tree (HAT). A HAT is a binary tree in which each leaf node corresponds to a data block. Though HAT does not have any limitation on the number of data blocks, for the sake of description simplicity, we assume that the number of data blocks n is equal to the number of leaf nodes in a full binary tree. Thus, for a file F = (m1,m2,m3,m4) where m_i represents the i-th blockof the file, we can construct a tree as shown in Fig. 1a.

Each node in HAT consists of a four-tuple $V_i = (i, l_i, v_i, t_i)$. *i* is the unique index of the node. The index of the root node is 1, and the indexes increases from top to bottom and from left to right. l_i denotes the number of leaf nodes that can be reached from the *i*-th node. v_i is the version number of the *i*-th node. t_i represents the tag of the i-th node. When a HAT is initialized, the version number of each leaf is 1, and the version number of each non-leaf node is the sum of that of its two children. For the *i*-th node, m_i denotes the combination of the blocks corresponding to its leaves. The tag t_i is computed from $\mathtt{F}(m_i)$, where $\mathtt{F}$ denotes a tag generation function. We require that for any node V_i and its children V_{2i} and V_{2i+1},

$$\mathtt{F}(m_i) = \mathtt{F}(m_{2i} \parallel m2i+1) = \mathtt{F}(m_{2i}) \,\&\, \mathtt{F}(m_{2i+1})$$

holds,where || denotes the combination of m_{2i} and m_{2i+1}, and & indicates the combination of $\mathtt{F}(m_{2i})$ and $\mathtt{F}(m_{2i+1})$, which is why we call it a "homomorphic" tree. An implementation of the tag generation function is described in Section 4.3.

B. Path and Sibling Search

To facilitate operations on HAT structures, we exploit two major algorithms for path search and sibling search. We define the path search algorithm $\mathrm{P}_l \leftarrow \mathrm{Path}(T, l)$. It takes a HAT T and a block index *i* of a file as input, and outputs the index set of nodes in the path from the root node to the *i*-th leaf node among all the leaves which corresponds to the *i*-th block of the file. We extend the path search algorithm to support multi-path search as Algorithm 1, where the i-th node in T consists of $v_i = (i, l_i, v_i, t_i)$.The algorithm takes as input a HAT and an ordered list of the block indexes, and outputs an ordered list of the node indexes. Lines 2-5 initialize two auxiliary variables for each legal block index *l* where $i_l\_$ defines a subtree whose root is the $i_l\_$-th node in T , and $\mathrm{ord}_l\_$ indicates the location of the corresponding leaf node in that subtree. Line 6 initializes a path P and a state *st*. The loop of lines 7-18 calculates the node that should be inserted into P by breadth-first search. For each level of T, the loop of lines 9-18 calculates the node in ρ for each block index l. For example, the path (gray nodes) to the 2nd leaf (the 10th node in the HAT) and the 5th leaf (the 7th node in the HAT) in Fig. 1b is $\rho_{2,5}$ = Path(T, {2, 5}) = {1, 2, 3, 5, 7, 10}.

Algorithm 1 Path search algorithm

```
1: procedure PATH(T, I)
2:      for $l \in$ I do
3:        if $l > l_1$ then
4:          return 0
5:        $i_l \leftarrow 1$, $ord_l \leftarrow l$
6:      $\rho \leftarrow \{1\}$, st $\leftarrow$ TRUE
7:      while $st$ do
8:        $st \leftarrow$ FALSE
9:        for $l \in$ I do
10:           if $l_{il} = 1$ then
11:               continue
12:           else if $ord_l \leq l_{2i}$ then
13:               $i_l \leftarrow 2i_l$
14:           else
15:               $ord_l \leftarrow ord_l - l_{2i}$, $i_l \leftarrow 2i_l + 1$
16:           $\rho \leftarrow \rho \cup \{ i_l \}$
17:           if $l_i > 1$ then
18:               $st \leftarrow$ TRUE
19: return ρ
```

Algorithm 2 Sibling search algorithm

```
1: procedure SIBLING ( ρ )
2:      $\psi \leftarrow ∮$, $\rho \leftarrow \rho \setminus \{1\}$, $£ \leftarrow ∮$, $i \leftarrow 1$
3:      while $\rho \neq ∮ \vee £ \neq ∮$ do
4:        if $2_i \in \rho$ then
5:          $i \leftarrow 2i$, $\rho \leftarrow \rho \setminus \{i\}$
6:          if $i + 1 \in \rho$ then
7:              $\rho \leftarrow \rho \cup \{(i+1, \text{FALSE})\}$
8:          else
9:              $\rho \leftarrow \rho \cup \{(i + 1, \text{TRUE})\}$
10:       else if $2_i + 1 \in \rho$ then
11:       $i \leftarrow 2i + 1$, $\rho \leftarrow \rho \setminus \{i\}$
12:       else if $£ \leftarrow ∮$ then
13:         pop the last inserted $(\alpha, \beta)$ in £
14:         $i \leftarrow \alpha$
15:         if $\beta$ = TRUE then
16:         $\psi \leftarrow \psi \cup \{i\}$
17:  return ψ
```

We define the sibling search algorithm ψ ←Sibling (ρ) as Algorithm 2. It takes the path ρ as input, and outputs the index set of the siblings of all nodes in the path ρ . Note that, the output of the sibling search algorithm is not an ordered list. It always outputs the leftmost one in the remaining siblings. Line 2 initializes the sibling set ψ and an auxiliary set £ . The loop of lines 3-16 first determines how many children of a node in ρ also appear in ρ (line 4, 6, and 10). If the answer is two, the algorithm removes these children from ρ and inserts the right child into £ for further validation (line 4-7). If the answer is one, the algorithm removes this child from ρ and inserts the other child into the sibling set ψ (line 8-11). However, there is a subtle difference between the left child is in ρ and the right child is in ρ . In the former, the right child will be inserted into ψ (line 15-16) later, while the left child will be immediately inserted into ψ (line 11) in the latter since the left child is the leftmost sibling in the remaining siblings. Lines 12-16 process the node in .From Fig. 1b, we have ψ = Sibling (ρ 2,5) = {4, 11, 6}. From Algorithm 1 and Algorithm 2, it is clear that both the path search algorithm and the sibling search algorithm have the same computation complexity O(b log(n)), where b is the number of block indexes (i.e., the size of I) and n is the number of leaf nodes.

V. The Construction Of Deypos

In this section, we propose a concrete scheme of deduplicatable dynamic PoS called DeyPoS. It consists of five algorithms as described in Section 2: Init, Encode, Deduplicate, Update, and Check.

A. Building Blocks

We employ the following tools as our building blocks:

1) **Collision-resistant hash functions:** A hash function H : {0, 1}∗ → {0, 1}∗ is collision-resistant if the probability of finding two diffenent values x and y that satisfy H(x) = H(y) is negligible.
2) **Deterministic symmetric encryption:** The encryption algorithm takes a key k and a plaintext m as input, and outputs the ciphertext. We use the notation $Enc_k(m)$ to denote the encryption algorithm.
3) **Hash-based message authentication codes:** A hash-based message authentication code HMAC : $\{0, 1\}^* \times \{0, 1\}^* \rightarrow \{0, 1\}^*$ is a deterministic function that takes a key k and an input x, and outputs a value y. We define $HMAC_k(x) \overset{def}{=} HMAC(k, x)$.
4) **Pseudorandom functions:** A pseudorandom function f : $\{0, 1\}^* \times \{0, 1\}^* \rightarrow \{0, 1\}^*$ is a deterministic function that takes a key k and a value x, and outputs a value y that is indistinguishable from a truly random function of the same input x. We define $f_k(x) \overset{def}{=} f(k, x)$.
5) **Pseudorandom permutations:** A pseudorandom permutation π : $\{0, 1\}^* \times [1, n] \rightarrow [1, n]$ is a deterministic function that takes a key k and an integer x where $1 \leqslant x \leqslant n$, and outputs a value y where $1 \leqslant y \leqslant n$ that is indistinguishable from a truly random permutation of the same input x. We define $\pi_k(x) \overset{def}{=} \pi(k, x)$.

Algorithm 3 The tag generation algorithm for a leaf node

```
1: procedure LEAFTAG ($\alpha_s$, $k_c$, $\alpha_c$, $c_i$, $l_i$, $v_i$)
2:     $T_l \leftarrow \alpha_s c_l$
3:     $t_{il} \leftarrow f_{kc}(i_l || l_{il} || v_{il}) + \alpha_c T_l$
4:     return $T_l$, $t_{il}$
```

Algorithm 4 The tag generation algorithm for a non leaf node

```
1: procedure NONLEAFTAG($k_c$, $i$, $l_i$, $v_i$)
2:     $T_{2i} \leftarrow T_{2i} - f_{kc}(2_i || l2_i || v_{2i})$
3:     $T_{2i+1} \leftarrow t_{2i+1} - f_{kc}(2_i + 1 || l_{2i+1} || v_{2i+1})$
4:     return $t_i \leftarrow f_{kc}(i || l_i || v_i) + T_{2i} + T_{2i+1}$
```

6) **Key derivation functions:** A key derivation function KDF : $\{0, 1\}^* \times \{0, 1\}^* \rightarrow \{0, 1\}^*$ is a deterministic function that can derive a secret key from two secret values.

B The Pre-Process Phase

In the pre-process phase, a user runs the initialization algorithm (*id*, *e*) ← Init(1^λ, F) which computes:

$$e \leftarrow H(F), id \leftarrow H(e).$$

Then, the user announces that it has a certain file via *id*. If the file does not exist, the user goes into the upload phase. Otherwise, the user goes into the deduplication phase.

C. The Upload Phase

Let the file F = ($m_1, . . . ,m_n$). The user first invokes the encoding algorithm (*C, T*) ← Encode(*e*, F) which is executed as follows.

1. Generate a random key $k \leftarrow \{0, 1\}^{|e|}$,and compute $r \leftarrow k \oplus e$.
2. Compute an encryption key $k_e \leftarrow KDF(k, 0)$. For each block m_l ($1 \leq l \leq n$), compute $c_l \leftarrow Enc_{ke}(m_i)$.
3. Build a HAT from F where the tags in HAT are unassigned.
4. Compute $\alpha_s \leftarrow KDF(k, 1)$, $k_c \leftarrow KDF(k, 2)$, and $\alpha_c \leftarrow KDF(k, 3)$. Compute all tags of leaf nodes in HAT with (*c1, . . . , cn*) via Algorithm 3.
5. Based on the tags of leaf nodes, compute all tags of non-leaf nodes in HAT.
6. Compute $w \leftarrow HMAC_e(t_1)$.
7. Set C = {*c1, . . . , cn*} and T = (r, α s, T , *w*,N), where T = {$T_1, . . . , T_n$} and N = {*V*1, *V*2, . . . } is the set of all HAT nodes.

We use a random key k rather than the hash value e as the encryption key. Thus, the encoding algorithm is probabilistic, which is very important to dynamic operations. Note that identical blocks always lead to the same ciphertext, but it is not a security issue because data confidentiality is not the goal of deduplicatable dynamic PoS, and the encryption algorithm in our construction is used for protecting e.

Algorithm 3 is designed for computing the tags of leaf nodes. Line 2 randomizes the data block which is used for deduplication, and line 3 calculates the tag of the data block which is attached to the node index of HAT i_l and other information, such as version number. Algorithm 4 is designed for computing the tags of non-leaf nodes. Lines 2-3 calculate the function of its children. Note that, T_{2i} and T_{2i+1} satisfy homomorphism. Then, line 4 binds the tag to the node index of HAT and other information.

At the end of the upload phase, the user uploads *C* and *T* to the cloud server and only stores *e* locally. Note that, e is an element of small constant size and can be encrypted and stored in the cloud server. In contrast with in [14] that requires users possessing or downloading a structure which has logarithmic size of the number of file blocks, all owners of the file can run the deduplication protocol, the checking protocol, and the update protocol without the complete structure of HAT in our scheme.

Algorithm 5 The deduplication proving algorithm

```
1: procedure DEDUPPROVE ($\alpha_s$, $k_c$, $\alpha_c$, {c1, . . ,cn},$Q$)
2:     $c \leftarrow 0, t \leftarrow \emptyset, \zeta \leftarrow 1, l \leftarrow 1$
3:     while $\zeta \leq n$ do
4:         $\delta \leftarrow 0$
5:         while $\zeta < l_{jl}$ do
6:             $\delta \leftarrow \delta + c_\zeta, \zeta \leftarrow \zeta + 1$
7:         pop the first element in Q
8:         $t \leftarrow t \cup \{f_{kc}(i \,||\, l_i \,||\, v_i) + \alpha_c\alpha_s\delta\}$
9:         $c \leftarrow c + c_\zeta$
10:        $l \leftarrow l + 1, \zeta \leftarrow \zeta + 1$
11:    return c, t
```

D. The Deduplication Phase

If a file announced by a user in the pre- process phase exists in the cloud server, the user goes into the deduplication phase and runs the deduplication protocol res ∈ {0, 1} ← Deduplicate(U(e, F), S(T)) as follows.

S executes the following instructions.

a) Choose $b \leftarrow [1, n]$ and $k \leftarrow \{0, 1\}^{\lambda}$. For each j $(1 \leq j \leq b)$, compute $lj \leftarrow \pi_k(b)$.
b) Compute the path ρ = Path(I), where I = {ι1, . . . , ιb}, and the siblings ψ =Sibling(ρ).
c) Send (r, b, κ,Q) to U, and keep L local, where Q is the set of (i, li, vi) and L is the set of ti for all i ∈ψ.

E. The Update Phase

In this phase, a user can arbitrarily update the file, such as modify a block, insert a batch of blocks, and delete some blocks, by invoking the update protocol res ∈ {(e*, (C*, T *))} ← Update(U(*e, ι,m,OP*), S(C, T)). After all operations are finished, the user uploads the updated blocks of the file and the updated nodes of the HAT to the cloud server as shown in Section 3.3. Then, the user computes the updated metadata e* and verifies the updated blocks via the checking protocol (described in Section 4.6).

Algorithm 6 The response algorithm

```
1: procedure RESPONSE(I)
2:     ρ ←PATH(T, I), ψ ←SIBLING(ρ)
3:     $c \leftarrow 0, t \leftarrow 0$
4:     for $l \in$ I do
5:         $c \leftarrow c + c_l$
6:     for i $\in \psi$ do
7:         $t \leftarrow t + t_i$
8:     return $resp \leftarrow (c, t, \{v_{il} \mid l \in I\}, \{(i, l_i, v_i)\}$
```

F. The Proof of Storage Phase

At any time, users can go into the proof of storage phase if they have the ownerships of the files. The users and the cloud server run the checking protocol *res ∈{0, 1} ← Check(S(C, T), U(e))* interactively to check the file integrity in the cloud server as follows.

1) U choose $b \in [1, n]$, $k \in \{0, 1\}^{\lambda}$, sends (b, κ) to S.
2) For each j $(1 \leq j \leq b)$, S computes $l_j \leftarrow \pi_k(b)$.Then, the cloud server invokes the response algorithm in Algorithm 6, where I = $\{l_1, \ldots, l_b\}$, and sends the proof *resp* to U with *(r, v1, ω)*.
3) U computes k ← r ⊕ e, α_s ← KDF(*k*, *1*), k_c ← KDF(*k*, 2), and α_c ← KDF(*k*, 3). Then, it verifies v1, and invokes the verification algorithm in Algorithm 7 to accomplish the verification. It outputs 1 if verification succeeds and 0 otherwise.

Algorithm 6 generates a proof on the server-side, which consists of the combination of the challenged file blocks (line 4-5) and the combination of corresponding tags (6-7). The algorithm also returns other information about the HAT (line 8), such as the version number of challenged leaf nodes. Algorithm 7 is designed for verifying the proof generated by Algorithm 6.

Algorithm 7 The verification algorithm

```
1: procedure VERIFY($\alpha_s, k_c, \alpha_c, v_1, I, resp$)
2:     ctr ← 1, $T \leftarrow 0$
3:     for $l \in$ I do
4:         while ctr < $l$ do
5:             pop the first element in $\{(i, l_i, v_i) \mid i \in \psi\}$
```

```
6:          $ctr \leftarrow ctr + l_i$, $T \leftarrow T + f_{kc}(i \,||\, l_i \,||\, v_i)$
7:       **if** $ctr \neq l$ **then**
8:          **return** 0
9:       **else**
10:         $ctr \leftarrow ctr + 1$
11: **for** $(i, l_i, v_i) \in \{(i, l_i, vi) \mid i \in \psi\}$ **do**
12:      $ctr \leftarrow ctr + l_i$, $T \leftarrow T + f_{kc}(i \,||\, l_i \,||\, v_i)$
13: **if** $ctr \neq n + 1$ **then**
14:     **return** 0
15: **else if** $t + f_{kc}(i \,||\, l_i \,||\, v_i) - T + \alpha_s \alpha_c c$ 6= t1 **then**
16:      **return** 0
17:  **else**
18:      **return** 1
```

The loop of line 3-10 first calculates current indexes of file blocks and the combination of tags (line 5-6). If the indexes of file blocks do not match the challenged indexes, the algorithm is terminated (line 7-8). The loop of line 11-12 calculates the remaining indexes and tags. In the end, the algorithm checks whether the number of file blocks is correct (line 13-14), and whether the HAT is authentic and up-to-date (line 15-16). The computation costs to generate challenge on the client-side, to generate a proof on the
server-side, and to verify the proof on the client-side in this phase are O(1), O(b log n), and O(b log n), respectively. Thecommunication cost is O(b log n).

VI. CONCLUSION

We proposed the comprehensive requirements in multi-user cloud storage systems and introduced the model of deduplicatable dynamic PoS. We designed a novel tool called HAT which is an efficient uthenticated structure. Based on HAT, we proposed the first practical deduplicatable dynamic PoS scheme called DeyPoS and proved its security in the random oracle model. The theoretical and experimental results show that our DeyPoS implementation is efficient, especially when the file size and the number of the challenged blocks are large.

REFERENCES

[1] S. Kamara and K. Lauter, "*Cryptographic cloud storage*," in P*roc.of FC*, pp. 136–149, 2010.

[2] Z. Xia, X. Wang, X. Sun, and Q. Wang, "A Secure and Dynamic Multi-Keyword Ranked Search Scheme over Encrypted Cloud Data," *IEEE Transactions on Parallel and Distributed Systems*, vol. 27, no. 2, pp. 340–352, 2016.

[3] Z. Xiao and Y. Xiao, "Security and privacy in cloud computing,"*IEEE Communications Surveys Tutorials*, vol. 15, no. 2, pp. 843–859,2013.

[4] C. A. Ardagna, R. Asal, E. Damiani, and Q. H. Vu, "From Security to Assurance in the Cloud: A Survey," *ACM Comput. Surv.*, vol. 48, no. 1, pp. 2:1–2:50, 2015.

[5] G. Ateniese, R. Burns, R. Curtmola, J. Herring, L. Kissner, Z. Peterson,and D. Song, "Provable data possession at untrusted stores,"in *Proc. of CCS*, pp. 598–609, 2007.

[6] G. Ateniese, R. Di Pietro, L. V. Mancini, and G. Tsudik, "Scalable and Efficient Provable Data Possession," in *Proc. of SecureComm*, pp. 1–10, 2008.

[7] G. Ateniese, S. Kamara, and J. Katz, "Proofs of storage from homomorphic identification protocols," in *Proc. of ASIACRYPT*, pp. 319–333, 2009.

[8] C. Erway, A. K¨upc ¨u, C. Papamanthou, and R. Tamassia, "Dynamic provable data possession," in *Proc. of CCS*, pp. 213–222, 2009.

[9] R. Tamassia, "Authenticated Data Structures," in *Proc. of ESA*, pp. 2–5, 2003.

[10] Q. Wang, C. Wang, J. Li, K. Ren, and W. Lou, "Enabling public verifiability and data dynamics for storage security in cloud computing," in *Proc. of ESORICS*, pp. 355–370, 2009.

[11] F. Armknecht, J.-M. Bohli, G. O. Karame, Z. Liu, and C. A. Reuter, "Outsourced proofs of retrievability," in *Proc. of CCS*, pp. 831–843, 2014.

[12] H. Shacham and B. Waters, "Compact Proofs of Retrievability,"*Journal of Cryptology*, vol. 26, no. 3, pp. 442–483, 2013.

[13] Z. Mo, Y. Zhou, and S. Chen, "A dynamic proof of retrievability (PoR) scheme with o(logn) complexity," in *Proc. of ICC*, pp. 912–916, 2012.

[14] E. Shi, E. Stefanov, and C. Papamanthou, "Practical dynamic proofs of retrievability," in *Proc. of CCS*, pp. 325–336, 2013.

[15] S. Halevi, D. Harnik, B. Pinkas, and A. Shulman-Peleg, "Proofs of ownership in remote storage systems," in *Proc. of CCS*, pp. 491–500, 2011.

[16] J. Douceur, A. Adya, W. Bolosky, P. Simon, and M. Theimer, "Reclaiming space from duplicate files in a serverless distributed file system," in Proc. of ICDCS, pp. 617–624, 2002.

[17] A. Juels and B. S. Kaliski, Jr., "PORs: Proofs of retrievability for large files," in Proc. of CCS, pp. 584–597, 2007.

[18] H. Shacham and B. Waters, "Compact proofs of retrievability," in Proc. of ASIACRYPT, pp. 90–107, 2008.

[19] Y. Dodis, S. Vadhan, and D. Wichs, "Proofs of retrievability via hardness amplification," in Proc. of TCC, pp. 109–127, 2009.

[20] K. D. Bowers, A. Juels, and A. Oprea, "HAIL: A high-availability and integrity layer for cloud storage," in Proc. of CCS, pp. 187–198, 2009.