An Efficient Dynamic Proof of Retrievability (PoR) Scheme

Zhen Mo Yian Zhou Shigang Chen
Department of Computer & Information Science & Engineering
University of Florida, Gainesville, FL 32611, USA

***Abstract*—Cloud storage has been gaining popularity because its elasticity and pay-as-you-go manner. However, this new type of storage model also brings security challenges. This paper studies the problem of how to ensure data integrity in cloud storage systems.**

In the Proof of Retrievability (PoR) model, after outsourcing the preprocessed data to the server, the client will delete its local copies and only store a small amount of meta data. Later the client will ask the server to provide a proof that its data can be retrieved correctly. However, most of the prior PoR works apply only to static data and the existing dynamic version of PoR scheme has an efficient problem.

In this paper, we extend the static PoR scheme to dynamic scenario. That is, the client can perform update operations, e.g., insertion, deletion and modification. After each update, the client can still detect data losses even if the server tries to hide them. We develop a new version of authenticated data structure based on a B+ tree and a merkle hash tree. We call it Cloud Merkle B+ tree ($CMBT$). By combining the $CMBT$ with the BLS signature, we propose a dynamic version of PoR scheme. Compared with the existing dynamic PoR scheme, We improve the worst case performance from $O(n)$ to $O(\log n)$.

I. INTRODUCTION

Cloud storage is a type of online storage model. Instead of providing a product, the cloud storage business provides data access and storage services in a pay-as-you-go manner. Developers and users do not need to know about the physical location and configuration of the system that delivers the services. They can easily and quickly adjust the resources to their needs. This elasticity of resources, without any pre-investment, attracts more and more people join the cloud storage.

Although envisioned as a promising service model, cloud s-torage also brings security concerns. One of the major concerns is about integrity of the data stored at the cloud side. After outsourcing the data to an off-site storage system and deleting the local copies, clients can be relieved from the burden of storage. However, at the same time, the clients lose physical control of their data. As the cloud storage system is maintained by a third party who cannot be totally trusted, it is extremely important for the clients to find an effective and efficient way to check the integrity of their data periodically.

In order to solve this problem, many schemes are proposed [1], [2], [3], [4], [5], [6], [7]. Considering different design goal-s, these schemes fall into two categories: Proof of Retrievability (PoR) [1], [2], [4], [6] and Provable Data Possession (PDP) [3], [7]. The PoR model is proposed by Juels and Kaliski in [4]. Their design goal is to ensure that the clients can retrieve the data from the server side. Ateniese et al. [3] propose a similar construction called Provable Data Possession (PDP) which demonstrates with the clients that the server side stores the files correctly. The PDP model is weaker than the PoR because its assurance is weaker than the PoR model. The PDP model does not guarantee that the clients can retrieve their data intactly. In the PDP model, the clients query the server periodically and the server returns a proof to guarantee that a certain percentage (e.g., 99%) of the file are intact. But if a very small amount of the file is lost or corrupted, the clients may not be able to detect it. In this case, the clients cannot retrieve their data intactly. However, in the PoR model, even if the clients may not detect the corruption, they can still recover the file with the help of erasure code. So we mainly consider the PoR scheme.

Another important concern is about supporting dynamic updates. In a cloud storage system, clients should not only be able to access the data, but also perform dynamic update operations, e.g., modification, deletion and insertion. However, most of previous works [2], [4], [6], [3], [7] can only apply to static data files. Though Wang $et\,al.$ propose a dynamic version of PoR model in [1], unfortunately, the performance of their scheme is not tightly bounded.

In this paper, we propose a new dynamic PoR scheme constructed based on a modified merkle hash tree and the Boneh-Lynn-Shacham (BLS) signature construction [8]. Our contribution can be summarized as follows: (1) We design a dynamic version of PoR model for the cloud storage system. (2) We propose a new data structure called Cloud Merkle B+ Tree (CMBT). By combining the $CMBT$ with the BLS construction, the worst case performance of our scheme is $O(logn)$.

The rest of the paper is organized as follows: In Section II, we define the system model and security model. Then we introduce preliminary works in Section III and present our scheme in Section IV. Finally we analyze the simulation results in Section V.

II. SYSTEM AND THREAT MODEL

A typical cloud storage system includes two parties: cloud storage servers and clients. The clients are limited in storage but have a large amount of data to be stored. On the contrary, the cloud storage servers have a huge amount of storage space and are providing storage services in a pay-as-you-go manner.

The cloud storage servers are maintained by a cloud service provider (CSP), such as Amazon or google. The clients will divide the data files into blocks. After putting data files to the cloud storage servers, the clients will delete the local copies and only keep a small amount of meta data. In addition, The storage service is not static. The clients will perform block-level update operations, such as modify a block, insert a block or delete a block.

As a third party, the CSP cannot be completely trusted. We define the following semi-trust model: In normal cases, the CSP will perform operations correctly, and will not deliberately delete or modify clients' data. But because of management errors, Byzantine failures and external intrusions, the CSP may lose or corrupt the hosted data inadvertently. When these errors happen, the CSP will try to save its reputation by hiding the truth of data loss.

In this paper, we fix the efficient problem in the existing dynamic PoR scheme, and propose a new dynamic version of PoR scheme. Our scheme can detect file corruptions with high probability even if the CSP tries to hide them. Moreover, our scheme is able to support dynamic updates while keeps the same detection probability of file corruption.

To simplify our discussion, we logically treat the cloud storage servers as one entity, called the server and the clients as the other entity, called the client.

III. RELATED WORK

Juels and Kaliski first formalize a scheme called Proofs of Retrievability (PoRs) [4]. By randomly embedding "sentinel" blocks into the outsourcing file and hiding these "sentinel" blocks' position by encryption, their scheme can detect static data corruption effectively. However, their scheme cannot support any data update, and the number of queries a client can perform is fixed.

Ateniese $et\,al.$ [3] first propose the provable data possession (PDP) model to ensure the integrity of outsourced data. they implement RSA-based homomorphic tags in their scheme. However, their scheme cannot be apply in dynamic scenario. Following their previous work, Ateniese $et\,al.$ [7] introduce a dynamic version of PDP model. But their scheme cannot support fully dynamic data operations.

Shacham $et\,al.$ introduce an improved version of PoRs scheme called Compact PoR [2] with rigorous security proofs. Based on the BLS signature, they aggregate the proofs into a small value and their scheme can support public verifications. However, using their scheme in dynamic scenario is impractical and insecure due to the following two reasons: First, its block signatures contain the indices of blocks. If a client deletes (or inserts) a block with index i, then any block with index j larger than i will have to change its index from j to $j-1$ (or $j+1$). So the client will need to re-sign all of the blocks whose indices have been changed, which makes this scheme impractical for supporting dynamic updates. Second, using [2] in dynamic scenario cannot prevent replay attacks.

Following the work of [2], Wang $et\,al.$ [1] define a dynamic version of PoR model based on the BLS signature and the Merkle Hash Tree (MHT) [9]. They try to use a modified BLS signature and the classic MHT construction to realize integrity verification in cloud storage. In their scheme, in order to build a MHT over a large piece of data, such as a file, the client first divides the file into a series of data blocks m_i $(1 \leq i \leq n)$ and computes the hash value for each block $n_i = H(m_i)$. We call n_i the "block tag" of the block m_i. Then the client constructs a binary tree whose leaf nodes are the hashes of the "block tags" and the nodes further up in the tree are the hashes of their respective children. Finally, the client generates a root R based on the construction of MHT and takes the signature of the root $sig_{sk}(R)$ as meta data.

However, using the classic MHT construction will cause an efficiency problem: After inserting or deleting some blocks, the MHT will become unbalanced. Particularly, if the client keeps appending blocks at the tail of the file, the height of the tree will increase linearly. As a result, the worst case of integrity check will be $O(n)$ instead of $O(logn)$ as described in [1], where n is the total number of blocks.

IV. OUR SCHEME

A. Overview

Our scheme can be summarized as the following three stages: (1) Preprocess stage: Before outsoucing the file to the server, the client will first encode the file with an erasure code and divide the encoded file into blocks. Then it constructs an authenticated data structure and generate the meta data. Next it will only keep the meta data and outsource others to the server. (2) Verification stage: After oursourcing the file to the server, the client will periodically check the integrity of its data. It queries the server with a subset of the data blocks and requires the server to provide a proof. By verifying the proof with the meta data, the client can detect the file corruption with high probability. (3) Update stage: The client will send the server a request to update the file. After each update, the server will prove to the client that the update operation is correctly executed.

B. Model

Our scheme can be described by the following algorithms:

• $KeyGen(1^k) \rightarrow (pk, sk)$ is an algorithm run by the client. It takes a security parameter as input, and returns a public key pk and a private key sk. The client stores the private key and sends the public key to the server.

• $Prepare(sk, F', F_{tags}) \rightarrow (\Phi, sig_{sk}(l(R)), CMBT)$ is executed by the client. As input, It takes an encoded file F' which is composed by a sequence of blocks m_i, where $0 \leq i \leq n$, the block tag set $F_{tags} = \{H(m_i), 0 \leq i \leq n\}$ and the private key sk. It outputs a signature set Φ which is an ordered collection of signatures $\{\sigma_i\}$ on $\{m_i\}$, where $0 \leq i \leq n$. We will define the signature set in the following subsection. The client also constructs a $CMBT$ based on the block tags F_{tags} and signs the label value of root $sig_{sk}(l(R))$ using the private key sk.

• $GenChallenge(n) \rightarrow Q$ is an algorithm executed by the client. The input is the total number of blocks and the output is a query Q which contains a set of IDs $I = \{i_1, i_2, ..., i_k\}$. Q is sent to the server as a request to verify the integrity of blocks whose index number $i \in I$.

• $GenProof(Q, CMBT, F', F_{tags}, \Phi) \rightarrow P$ is run by the server. It takes the query Q, the $CMBT$, the encoded file F', the block tag set F_{tags} and the signature set Φ as input. It outputs a proof P to let the client check the integrity of the blocks in query Q.

• $Verify(pk, Q, P, l(R)) \rightarrow (TRUE, FALSE)$ is an algorithm executed by the client. After receiving the proof P, the client will check the integrity of blocks in Q. It outputs $TRUE$ if the integrity of the blocks are verified as correct. Otherwise, it returns $FALSE$.

• $UpdateRequest() \rightarrow Request$ is executed by the client. It takes nothing as input and outputs an update request $Request$ which contains: an $Order \in \{Insert, Delete, Modify\}$, a index number i. Also if the $Order$ is $Modify$ or $Insert$, the request R should also contain: a new file block m^* and its signature σ^*.

• $Update(F', F_{tags}, \Phi, R) \rightarrow (P_{old}, P_{new})$ is an algorithm run by the server. After receiving the update request from the client, It takes the encoded file F', the block tag set F_{tags}, the signature set Φ and an update request R as input, outputs two proofs P_{old} and P_{new}.

• $UpdateVerify(P_{old}, P_{new}) \rightarrow (TRUE, FALSE)$ is executed by the client. With the inputs P_{old} and P_{new}, the client outputs $TRUE$ if the server's behaviors are honest in the update process. Otherwise, it returns $FALSE$.

C. Preprocess

Before outsourcing the files to the server, the client will first encode the file F to F' using an erasure code. Then it will run the algorithms $KeyGen(1^k)$ to create a pair of keys, and use $Prepare(sk, F, F_{tags})$ to generate a signature set Φ, a $CMBT$ and the meta data $sig_{sk}(l(R))$. The signature set Φ and the $CMBT$ are defined as follows.

1) BLS signature: We use the same BLS signature as defined in [1]. For a bilinear map $e : G \times G \rightarrow G_T$, the private key and the public key are defined as $x \in \mathbb{Z}_p$ and $v = g^x \in G$ separately, where g is a generator of G. For each block m_i, where $i \in [1, n]$, the signature on the block m_i is defined as $\sigma_i = [H(m_i)u^{m_i}]^x$, where u is a generator of G. We denote the set of signature as $\Phi = \{\sigma_i\}$, where $1 \leq i \leq n$.

2) CMBT: The merkle hash tree [9] has been widely used in checking memory integrity [10], [11] and certificate revocation [12], [13] because it is easy to realize and has $O(logn)$ complexity in both the worst case and the average case. However, directly using the classic merkle tree in cloud storage may cause some problems (see III). So we develop an authenticated data structure based on a B+ tree and a merkle hash tree. We call it **Cloud Merkle B+ tree (**$CMBT$**)**. In our construction,

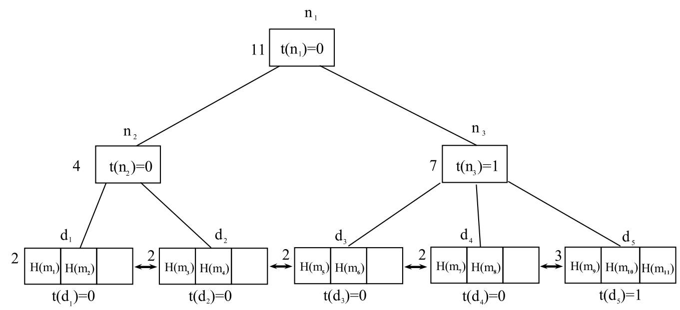

Fig. 1. The cloud merkle B+ tree

we choose a B+ tree of order three[1] and require that each data node can store three elements at most.

We treat the sequence of block tags $H(m_1)$, $H(m_2)$,..., $H(m_n)$ as elements and insert them into a B+ tree sequentially, then we can get a B+ tree (see Figure 1), we will construct the $CMBT$ based on it.

For each node w in $CMBT$, we store six values:

• $left(w)$, $middle(w)$ and $right(w)$: For an index node, these three variables represent its left child, middle child and right child. If this node has only two children, then $right(w)$ will be NIL. For a data node, these three variants represent the elements it stores from left to right. If corresponding position has no element, NIL will be set.

• $r(w)$: Rank of node. For an index node w, $r(w)$ stores the number of elements[1] that belong to the subtree whose root is w. For a data node w, $r(w)$ stores the number of elements that belong to w. In Figure 1, we show the rank value for each node. For example, the rank of node d_1 is 2 because from d_1 we can visit 2 elements $H(m_1)$ and $H(m_2)$.

• $t(w)$: We do not store keys in index node because we do not need to search the $CMBT$. Instead, we store the type of the node as $t(w)$. The definition of $t(w)$ shows as follows.

For a node w in the tree:

DEFINITION 1.

$$t(w) = \begin{cases} 0 & \textit{if } w \textit{ has 2 children or contains 2 elements} \\ 1 & \textit{if } w \textit{ has 3 children or contains 2 elements} \end{cases}$$

• $l(w)$: The label of node. $l(w)$ is defined as follows:

First, we define a collision resistant hash function $h(*)$, which has two inputs.

DEFINITION 2.

$$h(a, b) = h(a||b)$$

[1]The B+ tree[14] is different from the B tree in following three aspects: 1. A B+ tree has two types of nodes - index nodes and data nodes. Index nodes store keys while data nodes store elements. But a B tree has only one type of node - data nodes 2. All data nodes in a B+ tree are linked together by a doubly linked list, but data nodes in a B tree are not linked. 3. The capacity of data nodes and index nodes can be different in a B+ tree, while the capacity of nodes in a B tree should be the same. For example, a B+ tree of order n means that the index nodes (except for the root node) can hold $n-1$ keys at most and hold $\lceil n/2 - 1 \rceil$ keys at least. But each data node can contain c elements at most and $\lceil n/2 \rceil$ elements at least. c and n can be different. The root node can hold n children at most and two children at least.

where || means concatenation.

Then we extend the function to more than two inputs.

DEFINITION 3.

$$h(a_1, a_2, ..., a_{n-1}, a_n) = h(a_1||a_2||...||a_{n-1}||a_n)$$

Now we define the value of node $l(w)$ as:

DEFINITION 4.

$$l(w) = h(l(left(w)), l(middle(w)), l(right(w)), t(w), r(w))$$

Also for each element e that contains a block m, we define the value of the element as follows:

DEFINITION 5.

$$l(e) = h(H(m))$$

With above definitions, the client can construct a $CMBT$ and get the label value of the root R. Then the client will sign the root label $l(R)$ using its private key: $sig_{sk}(l(R)) \leftarrow (l(R))^{sk}$. Next the client will outsources the encoded file F, the block signature set Φ, the $CMBT$ and the root signature $sig_{sk}(l(R))$ to the server.

D. Query

Suppose the encoded file F', $CMBT$ etc have been outsourced to the server. The client only stores the meta data and the number of blocks n. The client generates a query to check the integrity of a series of random blocks whose index numbers belong to the set $I = \{i_1, i_2, ...i_k\}$. It uses algorithm $GenChallenge(n) \rightarrow Q$ to generate a query Q. For each index number $i \in I$, the client chooses a random element $v_i \leftarrow \mathbb{Z}_p$. Then a query Q is defined as $Q = \{(i, v_i)\}_{i_1 \leq i \leq i_k}$.

After receiving the query, the server will run algorithm $GenProof(Q, CMBT, F, F_{tags}, \Phi) \rightarrow P$ to generate a proof P by first computing:

$$\mu = \sum_{i=i_1}^{i_k} v_i m_i \in \mathbb{Z}_p$$

$$\sigma = \prod_{i=i_1}^{i_k} {\sigma_i}^{v_i}$$

Then the server generates a sequence of messages for each block tag $H(m_i)$ in the block tag set $S = \{H(m_i)$, where $i \in I$. Suppose $\{w_1, w_2, w_3, ..., w_h, w_{h+1}\}$ denotes the path from the root node to the element $H(m_i)$, where $i \in [i_1, i_k]$, h is the height of the $CMBT$ and the node w_j is the parent of w_{j+1}. For each node w_j, $j \in [1, h]$, the server will provide a message M_j.

The message M_j is a 2-tuple value. We define n_{j+1} and n'_{j+1} as two neighbors of the node w_{j+1} and the location of $n_(j+1)$ is always on the left of n'_{j+1}. We denote T as a set of nodal information which is defined as follows.

$$T_{n_{j+1}} = \{l(n_{j+1}), r(n_{j+1}), t(n_{j+1}), p(n_{j+1})\}$$
$$T_{n'_{j+1}} = \{l(n'_{j+1}), r(n'_{j+1}), t(n'_{j+1}), p(n'_{j+1})\}$$

If w_{j+1} has only one sibling, then $T(n'_{j+1}) = NULL$. We use $p(n_{j+1})$ to represent the location relationship between n_{j+1} and w_{j+1}.

$$p(n_{j+1}) = \begin{cases} 0 & \text{if } n_{j+1} \text{ is on the left of } w_{j+1} \\ 1 & \text{if } n_{j+1} \text{ is on the right of } w_{j+1} \end{cases} \quad (1)$$

So the definition of message for a node w_j is

DEFINITION 6.

$$M_j = \{T_{n_{j+1}}, T_{n'_{j+1}}\}$$

We denote the message sequence $\gamma_i = \{M_1, M_2, ..., M_h\}$ for element $H(m_i)$, and for all elements in set I, the message set will be $\Gamma = \{\gamma_{i_1}, ..., \gamma_{i_k}\}$. So the definition of the proof P is $P = \{\mu, \sigma, S, \Gamma\}$.

If the client only wants to check the integrity of one block instead of a group of blocks, we use the following definition to represent the proof of a single block with index i.

DEFINITION 7.

$$Query(i) = \{v_i m_i, {\sigma_i}^{v_i}, H(m_i), \gamma_i\}$$

E. Verification

After receiving the proof P from the server, the client will run the algorithm $Verify(pk, Q, P, l(R))$ (see Algorithm 1) to check the integrity of blocks whose indices belong to the set I. In Algorithm 1, $\{w_1, w_2, w_3, ..., w_h, w_{h+1}\}$ is the node sequence from the root to the element $H(m_i)$. To compute the value of $w_j (1 \leq j \leq h)$, the client first determines how many children the node w_j has. Then it uses a function $GetValue$ who takes the children's values and their location relationship p (see Equation 1) as inputs, and compute the value of w_j (see definition 4). The computation procedure will continue until reaching the root node. During the procedure, the client can verify the index number idx of the block tag $H(m_i)$.

F. Updates

Now we will show that our scheme can effectively and efficiently support dynamic update operations which include: modification, insertion and deletion. We assume that the encoded file F', the block signature set Φ, the $CMBT$ etc have been generated and stored at the server.

Suppose the client wants to update the j^{th} block, where $1 \leq j \leq n$. The client will first run algorithm $UpdateRequest() \rightarrow Request$ to generate an update request and send the request to the server. Upon receiving the modification request the server will update the block and run the algorithm $Update(F', F_{tags}, \Phi, R)$ to generate two proofs P_{old} and P_{new}. Based on the P_{old} and P_{new}, the client will use the algorithm $UpdateVerify(P_{old}, P_{new})$ to ensure correctness of the update.

• Modification: Suppose a client wants to modify the j^{th} block, where $1 \leq j \leq n$, from m_j to m'_j. The client generates an update request $Request = \{Modify, j, m'_j, \sigma'_j\}$ and sends the request to the server. The server will update the block, reconstruct the $CMBT$ and generate the proof

Algorithm 1 $Verify(pk, Q, P, l(R)) \rightarrow (TRUE, FALSE)$

Verify $e(\sigma, g) \stackrel{?}{=} e(\prod_{i=i_1}^{i_k} H(m_i)^{v_i} \cdot u^{\mu}, v)$
for i from i_1 to i_k **do**
  $\gamma_i = \{M_1, M_2, ..., M_h\}$, $M_j = \{T_{n_{j+1}}, T_{n'_{j+1}}\}$
  $T_{n_{j+1}} = \{l(n_{j+1}), r(n_{j+1}), t(n_{j+1}), p(n_{j+1})\}$
  $T_{n'_{j+1}} = \{l(n'_{j+1}), r(n'_{j+1}), t(n'_{j+1}), p(n'_{j+1})\}$
  $idx = 1$
  for j from h down to 1 **do**
    if $T_{n'_{j+1}} \neq NULL$ **then**
      $r(w_j) = r(w_{j+1}) + r(n_{j+1}) + r(n'_{j+1})$
      $t(w_j) = 1$
      $l(w_j) = GetValue(T_{n_{j+1}}, T_{n'_{j+1}})$
      if $p(n'_{j+1}) = 0$ **then**
        $idx = idx + r(n'_{j+1})$
      end if
    else
      $r(w_j) = r(w_{j+1}) + r(n_{j+1})$
      $t(w_j) = 0$
      $l(w_j) = GetValue(T_{n_{j+1}})$
    end if
    if $p(n_{j+1}) = 0$ **then**
      $idx = idx + r(n_{j+1})$
    end if
  end for
  if $l(w_1) = l(R)$ AND $idx = i$ **then**
    if $i = i_k$ **then**
      return $TRUE$
    end if
  else
    return $FALSE$
  end if
end for

$P_{old} = Query(i)$ (see Definition 7). Based on the P_{old}, the client can not only check the integrity of the block m_j (see Algorithm 1), but also construct a partial $CMBT$. Figure 2 shows an example. The partial $CMBT$ constructed from the query on the $CMBT$ in Figure 1. The client will get enough information to update the $CMBT$ from the partial $CMBT$. In this case, the client can compute the new root R_{new} based on P_{old}. The server only needs to send $P_{new} = R'$ which is the new root node to the client. After verifying the correctness of the new root R', the client will sign the new root $sig_{sk}(l(R'))$ and send it back.

• Insertion: The procedure of insertion is similar to modification. The only difference is that when we insert an new element into a data node which already contains three elements, the data node will split into two nodes. The procedure will keep going up until one index node has only two children or we need to generate a new root and increase the height of the tree by one. With the partial $CMBT$ constructed from the proof $P_{old} = Query(i)$, the client will have enough information to compute new root R_{new}. Also, after verifying the correctness of the new root R', the client will sign the new

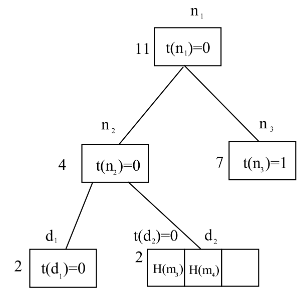

Fig. 2. The partial $CMBT$ constructed from $P_{old} = Query(4)$.

root $sig_{sk}(l(R'))$ and send it back.

• Deletion: The procedure of deletion is different from insertion and modification because deleting an element from a data node who has two elements will cause the data node become deficient. So we need to do some "borrow" or merge operations to keep the tree balanced (See [14] for more details). However, with the partial $CMBT$ constructed from the proof $P_{old} = Query(i)$, the client may not acquire enough information to finish these operations and compute a new root R_{new}. Accordingly, server will need to send another proof P_{new} to help the client verify the correctness of the new root R'. Here we define another algorithms: Algorithm $Query_{new}(i)$ is used to return the proof of the i^{th} element in the updated $CMBT$.

Based on $P_{old} = Query(i)$, the client can generate a partial $CMBT$ which contains the node sequence $\{w_i\}$ and their siblings $\{n_i, n'_i\}$, where $i \in [1, h+1]$. w_{h+1} is the element that the client wants to delete. The deletion procedure falls into three cases:

1) If the leaf node w_h contains three elements, then the client only need to delete the element w_{h+1} and generate a new root R' based on the proof P_{old}. Otherwise, the client will keep searching the node sequence from w_h to w_1 until it finds a node w_j, where $j \in [1, h]$ whose right or left sibling node has three children or w_j itself has three children.

2) If one of w_j's sibling has three children, then the client needs to "borrow" a child from its sibling to generate a new node. But the proof P_{old} does not contain the information of this child. So the client will use algorithm $Query_{new}(i)$ and $Query(k)$ to acquire additional information to delete the element w_{h+1} and generate a new root. k is the index number of the element belongs to the subtree whose root is the sibling node. The client can get k easily based on P_{old}. The information from $Query_{new}(i)$ can be verified by P_{old}.

3) If w_j has three children, the client will delete the element and merge two children into one. By using algorithm $Query_{new}(i)$, it will acquire enough information to generate the new root. If the client cannot find the node until it reaches the root, the client will need to generate a new root. In this case, the height of the $CMBT$ will be decreased by one. Due to space limitations, we cannot provide the complete algorithm, but it is easy to prove that the complexity of deletion is $O(logn)$.

TABLE I
COMPARISON OF EXISTING PoR SCHEMES.

Features	Different Schemes			
	[4]	[2]	[1]	Our Scheme
Dynamic updates	NO	NO	YES	YES
Public verification	NO	YES	YES	YES
Worst comm. complexity	$O(1)$	$O(1)$	$O(n)$	$O(logn)$
Average comm. complexity	$O(1)$	$O(1)$	$O(logn)$	$O(logn)$

V. SIMULATION RESULTS

We first list the features of our scheme and make a comparison with existing PoR schemes in Table I. Our experiment is running on a system with Intel Core 2 2.53 GHz, 4 GB RAM, and a 7200 RPM TOSHIBA 120 GB SATA driver. Algorithms are implemented using C++.

We evaluate the performance of our scheme in terms of communication overhead. Based on precious analysis, we know that the communication cost of proofs of retrievability for a file depends on the block size and the number of messages (hashes) send to the client. As proved in [3], detecting a 1% file corruption with 99% confidence needs query a constant number of 460 blocks. Accordingly, if the block size is fixed, the performance is determined by the communication cost of sending these messages to prove the index of a block in the tree. In our experiment, we implement the hash function using SHA1 with output size 160 bits. As the average communication cost of our scheme and the scheme in [1] are similar, we compare the maximum communication cost of proving a block in the $CMBT$ and the MHT in Figure 3.

The client first divides the encoded file F' into 128 blocks and uses these blocks to construct the MHT and the $CMBT$. Then the client outsources the encoded file F', the MHT and the $CMBT$ to the server. Next, suppose the client keeps appending blocks at the tail of F'. Figure 3 shows the comparison of the maximum communication cost between the MHT and the $CMBT$. The x axis represents the number of blocks that the client appends to the encoded file after initialization. The y axis represents the communication cost to prove a block in the tree. From Figure 3, we learn that the worst case communication cost of the MHT increase linearly with the number of inserting blocks. We know that the worst case communication cost of MHT is $O(n)$.

VI. CONCLUSION

Cloud storage brings security concerns. One major concern is about the data integrity. In this paper, we extend the static PoR scheme to dynamic scenario. We propose a new authentication data structure called Cloud Merkle B+ tree ($CMBT$). Compared with the existing dynamic PoR scheme, our worst case communication complexity is $O(logn)$ instead of $O(n)$.

ACKNOWLEDGMENT

This work was supported in part by the US National Science Foundation under grant CNS-1115548 and a grant from Cisco Research.

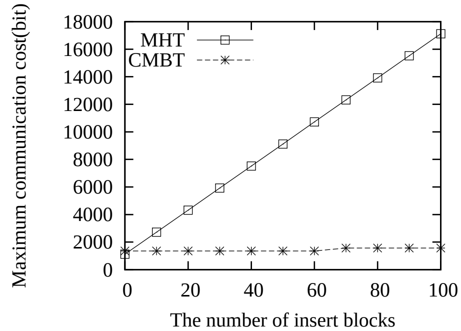

Fig. 3. Comparison of the maximum communication cost of the MHT and the $CMBT$. The x-axis represents the number of blocks that the client appends to the file after initialization. The y-axis represents the communication cost to prove one block in the MHT and the $CMBT$.

REFERENCES

[1] Q. Wang, C. Wang, J. Li, K. Ren, and W. Lou, "Enabling public verifiability and data dynamics for storage security in cloud computing," *Computer Security–ESORICS 2009*, pp. 355–370, 2009.
[2] H. Shacham and B. Waters, "Compact proofs of retrievability," *Advances in Cryptology-ASIACRYPT 2008*, pp. 90–107, 2008.
[3] G. Ateniese, R. Burns, R. Curtmola, J. Herring, L. Kissner, Z. Peterson, and D. Song, "Provable data possession at untrusted stores," in *Proceedings of the 14th ACM conference on Computer and communications security*. ACM, 2007, pp. 598–609.
[4] A. Juels and B.S. Kaliski Jr, "Pors: Proofs of retrievability for large files," in *Proceedings of the 14th ACM conference on Computer and communications security*. ACM, 2007, pp. 584–597.
[5] C. Erway, A. Küpçü, C. Papamanthou, and R. Tamassia, "Dynamic provable data possession," in *Proceedings of the 16th ACM conference on Computer and communications security*. ACM, 2009, pp. 213–222.
[6] K.D. Bowers, A. Juels, and A. Oprea, "Proofs of retrievability: Theory and implementation," in *Proceedings of the 2009 ACM workshop on Cloud computing security*. ACM, 2009, pp. 43–54.
[7] G. Ateniese, R. Di Pietro, L.V. Mancini, and G. Tsudik, "Scalable and efficient provable data possession," in *Proceedings of the 4th international conference on Security and privacy in communication netowrks*. ACM, 2008, p. 9.
[8] D. Boneh, B. Lynn, and H. Shacham, "Short signatures from the weil pairing," *Advances in CryptologyłASIACRYPT 2001*, pp. 514–532, 2001.
[9] R. Merkle, "A digital signature based on a conventional encryption function," in *Advances in CryptologyłCRYPTO87*. Springer, 2006, pp. 369–378.
[10] D. Williams and E.G. Sirer, "Optimal parameter selection for efficient memory integrity verification using merkle hash trees," in *Network Computing and Applications, 2004.(NCA 2004). Proceedings. Third IEEE International Symposium on*. IEEE, 2004, pp. 383–388.
[11] B. Gassend, G.E. Suh, D. Clarke, M. Van Dijk, and S. Devadas, "Caches and hash trees for efficient memory integrity verification," in *High-Performance Computer Architecture, 2003. HPCA-9 2003. Proceedings. The Ninth International Symposium on*. IEEE, 2003, pp. 295–306.
[12] H. Kikuchi, K. Abe, and S. Nakanishi, "Online certification status verification with a red-black hash tree," *IPSJ Digital Courier*, vol. 2, no. 0, pp. 513–523, 2006.
[13] M. Naor and K. Nissim, "Certificate revocation and certificate update," *Selected Areas in Communications, IEEE Journal on*, vol. 18, no. 4, pp. 561–570, 2000.
[14] E. Horowitz and S. Sahni, *Fundamentals of data structures*, Computer science press, 1983.