Enabling Non-repudiable Data Possession Verification in Cloud Storage Systems

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

Chengzhong Xu
Department of Electrical and Computer Engineering
Wayne State University, Detroit, MI 48202

***Abstract*—After clients outsource their data to the cloud, they will lose physical control of their data. Many schemes are proposed for clients to verify the integrity of their data. This paper considers a complementary problem: When a client claims that the server has lost their data, how can we be sure that the client is correct and honest about the loss? It is possible that the client's meta data is corrupted or the client is lying in order to blackmail the server. In addition, most previous work relies on sequential indices. However, the indices bring significant overhead to bind an index to each block. We propose to replace sequential indices with much flexible non-sequential *coordinates*. The binding of coordinates to data blocks is performed through a *Coordinate* Merkle Hash Tree (CMHT). Based on CMHT, we can improve both the average and the worst-case update overhead by simplifying the updating algorithm.**

I. Introduction

While cloud storage has become a fast-growing market, it also brings security issues. One major concern is about the integrity of user data. By outsourcing data to an off-site storage system and removing local copies, cloud users are relieved from the burden of storage, but in the meantime lose physical control of their data. Can we trust the cloud service providers? Even if we assume that service providers will not deliberately hinder clients' correct access to their own data (after all this is the providers' lifeline business), involuntary security breaches may occur. For example, a provider may lose user data due to hardware failure, human mistakes or external intrusion. Not having the data, the provider may attempt to hide such an incident in order to save reputation. To address this issue, we should give clients a means to challenge their provider for a proof that it indeed possesses all client data.

Most existing solutions can be categorized along two research threads: Proof of Retrievability (PoR) [1], [2], [3], [4] and Provable Data Possession (PDP) [5], [6]. Both categories allow users to verify if the cloud correctly possesses their data. That is, by keeping some local meta data and verifying the proof returned from the cloud, users can (probabilistically) determine whether their data are intact. The above schemes have two limitations: First, they either do not support dynamic data update or do so with significant overhead, particularly in terms of worst-case complexities. Second, they protect users from the cloud's misbehavior, but do not protect the cloud from the users' misbehavior. This paper expands data integrity protection by covering an important complementary problem: When a user claims a data loss, how can we be sure that the user is correct and honest about the loss? If a user tries to blackmail the cloud by lying about data loss, how can the cloud prove its innocence?

We propose a data-possession verification scheme for cloud storage that makes two major contributions:

• It has a new non-repudiation property that allows the cloud and the users to supervise each other, so that users are able to detect whether the cloud has lost their data and the cloud is able to fend off the false claims of data loss from users. Comparing with previous work, our scheme can protect the rights of both sides.

• Another important concern in designing data-possession verification schemes in cloud storage is the amount of overhead they introduce into the system. We design a new Coordinate Merkle Hash Tree (CMHT) to assist the verification procedure, whose *worst-case* computation/communication overhead for inserting/deleting/updating a data block is $O(\log n)$, comparing with $O(n)$ worst-case overhead in [7], [1]. Our simulation results demonstrate that the average overhead of CMHT is also much smaller than the best existing work.

II. Related Work

Ateniese $et\,al.$ [5] propose the first provable data possession (PDP) model to check the integrity of outsourced data. But their scheme does not support data update. A followup work by Ateniese $et\,al.$ [6] introduces a dynamic version of PDP. Unfortunately, it cannot support all types of data update operations.

Juels and Kaliski formalizes a scheme called Proofs of Retrievability (PoR) to verify data integrity through "sentinel" blocks [3], but it does not support data update, either. Shacham $et\,al.$ introduce an improved version of PoR called Compact PoR [2]. But still their scheme is not designed to efficiently support dynamic data updates.

Following the work of [2], Erway $et\,al.$ [7] propose a dynamic provable data possession scheme (DPDP). Using a rank-based skiplist, their scheme supports dynamic data update. However, maintaining the sequential order among nodes at the bottom level of a skiplist makes updating (such as deletion) complicated. Moreover, the skiplist is a probabilistic data structure, whose worst-case overhead complexity is $O(n)$ [8], where n is the number of blocks.

Wang $et\,al.$ [1] define a dynamic version of PoR based on the BLS signature and a sequenced Merkle Hash Tree (MHT) [9]. After inserting or deleting data blocks, the tree will become unbalanced. Particularly, if the client keeps appending blocks at the end of the file, the height of the tree will increase

978-1-4799-5063-8/14 $31.00 © 2014 IEEE
DOI 10.1109/CLOUD.2014.40

linearly. Our earlier work tries to rebalance the tree through rotation [10]. But performing tree rotation remotely requires significant information exchange between the client and the server. Our experiments reveal that its average update overhead is considerably higher than DPDP [7].

Zhang and Blanton take a different approach [11] that requires the client to locally record information about update history, using a balanced update tree whose size is $O(M)$, where M is the number of updates. Even though sequential indices are explicitly bound with blocks through MACs, the update tree allows the client to translate indexing information without having to re-computing MACs. The above approach however puts significant storage burden on the client. This paper will follow the path of the prior work [5], [3], [7], [1] whose client storage requirement is a constant. See table I for a summary of the existing work.

III. Models and Objective

A. System Model

Our system model consists of two parties: (1) The *clients* are individual users or companies. They have a large amount of data to be stored, but do not want to maintain their own storage systems. By outsourcing their data to the cloud and deleting the local copies, they are freed from the burden of storage management. (2) The cloud *servers* have a huge amount of storage space and computing power. They offer resources to clients on a pay-as-you-go manner.

B. Threat Model

Server: Because of management error, hardware failure or external intrusion, the server may lose or corrupt the hosted data. When this happens, the service provider may try to hide the truth of data loss and pretend to possess the data.

Client: Some clients may falsely claim data loss in order to damage the reputation of a cloud service provider (possibly backed by competitors) or to blackmail the provider.

C. Data Possession Verification and Basic Approach

Consider an arbitrary client and an arbitrary file F that the client outsources to the cloud. Suppose F consists of n data blocks, $\{m_1, m_2, ..., m_n\}$. Each block may contain a data key (ID) to allow lookup and access of a specific block of interest (for example, the record of a particularly employee indexed by the employee ID in a payroll file). The blocks do not necessarily have the same size. The problem is to design a data-possession verification scheme that allows the client to (1) detect whether some of the blocks have been lost or corrupted at the cloud server, and (2) in the meantime make sure that the cloud can fend off false claims of client data loss.

Much existing work focuses on addressing the first part of the above problem based on a common basic approach [7], [1], [2]: The client randomly selects a subset of k blocks and queries the cloud for a proof, demonstrating that it possesses these blocks. After receiving the proof, the client verifies the proof using the meta data it has pre-computed and kept locally. If the received proof does not match what's expected from the meta data, the client claims that the cloud has lost some of its data. It is known that if the cloud loses k' data blocks, the probability of being detected after a single client query of k blocks is $1 - \frac{\binom{n-k'}{k}}{\binom{n}{k}}$ [5]. As an example, if 1% blocks are lost by the server, the client can achieve 99% detection probability by querying 460 blocks. If the above approach is performed periodically for l times, each time on an independent subset of 460 blocks, the detection probability will become $1 - (1 - 99\%)^l = 1 - 10^{-2l}$.

One problem is that the server may cheat: If one of the 460 blocks is lost, the server may simply substitute it with another block that it has. To prevent such cheating, the prior work [7], [1] has a worst-case overhead of $O(n)$ through ranked lists or Merkle tree, as we have mentioned earlier. We try to improve not only the average performance but also the worst-case overhead to $O(\log n)$ without expensive tree rotation.

Even though the presentation of our data-possession verification scheme will be based on a file F, the notation F can be generalized to be a single file, a part of a large file, a set of files, a data stream, or a segment of a data stream. Using key-based authenticated dictionaries, the proposed scheme for a file can be extended for a file system consisting of many files with a directory structure in a similar way as [7].

IV. Coordinate Merkle Hash Tree (CMHT)

We prepare data structures for our data-possession verification scheme.

A. Tag and Signature

Block tag: For each block $m_i \in F$, we define its *tag* as $t_i = H(m_i)||c_i$, where H is a collision-resistant hash function and c_i is a coordinate value assigned to m_i, which will be discussed shortly. A tag is a fixed length representation of a data block (whose length may be arbitrarily set) in the data structure of CMHT.

Homomorphic Signature: The client chooses $N = pq$ where p and q are two large primes and g is an element of high order in $\mathbb{Z}_N^*$. The client keeps p and q, and sends N and g to the server. The signature for block m_i is defined as

$$\sigma_i = t_i \cdot g^{m_i} \mod N,$$

where t_i is the tag of m_i. Note that it is possible for our scheme to use other homomorphic signatures such as BLS [12]. We denote the set of signatures as $\Phi = \{\sigma_i\}$. These signatures will be used for data-possession verification. In particular, it is essential to include t_i in the signature in order to ensure non-repudiation.

B. Tree Construction

The client binds each block m_i to a unique coordinate c_i through a coordinate Merkle hash tree. To do so, it first constructs a complete binary tree with n leaves. For convenience, we denote the n leaves from left to right as w_1, w_2, ..., w_n. Each leaf w_i represents a data block m_j and is assigned a coordinate c_j, which encodes the path from the root to the leaf: Initialize c_j to be nil. Traverse from the root to w_i. Append a bit '0' to c_j whenever taking a left-child link or a bit '1' whenever taking a right-child link.

Features	Different Schemes				
	Ateniese	J&K [3]	Shacham [2]	Wang [1]	Erway [7]
Dynamic updates	NO	NO	NO	YES	YES
Public verification	NO	NO	YES	YES	YES
Worst comm. complexity	O(1)	$O(1)$	$O(1)$	$O(n)$	$O(n)$
Worst comp. complexity	O(1)	$O(1)$	$O(1)$	$O(n)$	$O(n)$
Average comm. complexity	$O(1)$	$O(1)$	$O(1)$	$O(\log n)$	$O(\log n)$
Average comp. complexity	$O(1)$	$O(1)$	$O(1)$	$O(\log n)$	$O(\log n)$

TABLE I
SUMMARY OF EXISTING WORK

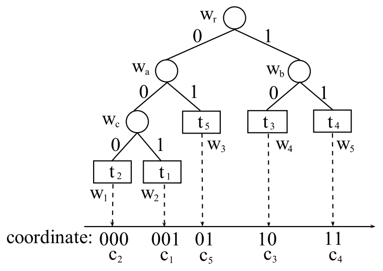

Fig. 1. A Coordinate Merkle Hash Tree (CMHT) constructed for 5 blocks

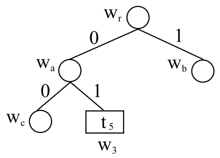

Fig. 2. The partial CMHT from the root to w_3

The client arbitrarily decides which leaf node represents which block. For example, in Figure 1, w_2 represents m_1, whose tag t_1 is shown inside the rectangle node of w_2. The value c_1 is 001 as the path from the root takes two left-child links and then a right-child link before reaching w_2. In the figure, we use letter subscribes (such as w_a, w_b and w_c) for internal nodes to help distinguish them from leaves. We use w_r for the root of the tree.

Let w_x be an arbitrary node in the tree. To support the Merkle tree operations, we define its *label*, $l(w_x)$, as follows: If w_x is a leaf node representing a block m_j, its label is simply the tag t_j of the block; if w_x is an internal node whose two children are w_{left} and w_{right}, its label is computed by hashing the concatenation of the labels of the children.

$$l(w_x) = \begin{cases} H(l(w_{left})||l(w_{right})) & \text{if } w_x \text{ is an internal node} \\ t_j & \text{if } w_x \text{ is a leaf node for block } m_j \end{cases}$$

Note that the CMHT is a complete binary tree and it does not have any internal node with one child.

C. Meta Data

After the client constructs the CMHT, it sends the tree to the cloud server, together with F and Φ. The server verifies the labels on the tree. Let T be the current time stamp. We define the *meta data* as follows:

$$\mathcal{M} = \{l(w_r), T, n, \sigma_{\mathcal{M}}\},$$

which includes the root's label $l(w_r)$, the time stamp T, the total number n of blocks, and a digital signature [13], $\sigma_{\mathcal{M}} = Sign_{sk_s}(Sign_{sk_c}(l(w_r)||T||n))$, jointly signed by the client and the server using their private keys, sk_c and sk_s. The meta data is stored by the client and the server separately as $\mathcal{M}_c$ and $\mathcal{M}_s$. Both sides can use the other's public key to verify the signature on the meta data, which enforces authenticity and consistency between the two sides. Compared to the meta data in [1], [2], our meta data has the following two security properties.

Unforgeability: As the meta data includes the signature signed by both the client and the server, neither the client nor the server can forge the meta data.

Distinguishability: After each update of the tree, the client and the server will agree on a new time stamp and update the signature. As the time stamp T increases monotonically, if the two sides have dispute over which meta data is current, it is easy to resolve the dispute by authenticating the signatures with their public keys and comparing the time stamps.

The client only stores the meta data $\mathcal{M}_c$. Everything else, including CMHT, F and Φ, is outsourced to the server. The server needs to maintain an internal data structure to map between data blocks and their corresponding nodes in the CMHT. When the server stores the CMHT, it keeps a location field in each leaf node, specifying where the corresponding data block is stored. The server also keeps track of each data block's size and its coordinate in the CMHT, allowing flexible access of the node in the tree for any given data block.

D. Use of Coordinates

Interestingly, the client does not need to keep track of the exact coordinate of each data block because it is not used in data access. The *only* purpose of introducing coordinates is to help the client detect data loss, which is performed in a random-sampling way: Since the client knows n from its meta data, it knows the exact shape of the complete binary tree CMHT and thus knows the set of valid coordinates. The client challenges the server with a randomly selected subset of valid coordinates $\{c_i\}$, which corresponds to a subset of data blocks $\{m_i\}$ to be verified, where m_i is the block *currently* assigned with c_i. We stress that the whole purpose of designing CMHT is to make sure that the server will return the *correct* tags $\{H(m_i)||c_i\}$ that match the client's *current* meta data, more specifically, the root's label $l(w_r)$, using the standard Merkle tree operations (which prevent the server from cheating). After that, the server is required to produce a compact proof of its knowledge of blocks $\{m_i\}$ that match the blocks' fingerprint $\{H(m_i)||c_i\}$.

All current data blocks must be represented in the CMHT, but their specific coordinates (e.g., locations in the tree) are *not important* to data-possession verification because coordinates are randomly sampled and each block (its coordinate) has equal chance to be sampled.

E. Partial CMHT

We define a *partial CMHT to a leaf node w_j at coordinate c* as the subset of nodes including w_j and the siblings of the nodes on the path from the root to w_j. As an example, for the CMHT in Figure 1, we show a partial tree to the leaf w_3 at coordinate 01 in Figure 2, where the nodes in the partial tree are shaded. Note that the shape of the partial tree is uniquely determined by the coordinate c, which specifies the exact sequence of left or right child links that the path from the root to the leaf w_j must take.

A partial CMHT to multiple leaf nodes at multiple coordinates is the combination of the partial trees to each of those leaf nodes.

V. Non-repudiation Possession Verification Scheme

A. Interaction between Client and Server

Before we present the implementation of our scheme, we give an overview of the interaction between the client and the server in the form of thirteen basic algorithms.

- $Gen(1^k) \rightarrow (pk, sk)$ is the algorithm in the digital signature scheme Π defined by [13]. Gen is executed by both the client and the server to produce a pair of public and private keys. The client stores its private key sk_c and sends the public key pk_c to the server. The server stores its private key sk_s and sends the public key pk_s to the client.
- $Sign_{sk}(m) \rightarrow \sigma$ is the algorithm in the digital signature scheme Π. It takes the private key sk and a message m as input, and outputs a signature σ on the message using the private key. In our scheme, the client and the server apply this algorithm to jointly produce the signature $\sigma_{\mathcal{M}}$ of the meta data.
- $VerifySign_{pk}(m, \sigma) \rightarrow (TRUE, FALSE)$ is the algorithm in the digital signature scheme Π. It takes as input the message and the signature. It verifies the correctness of the signature using the public key and outputs the result of the verification. It is used by both the client and the server to verify the authenticity of the meta data.
- $Prepare(sk_c, F) \rightarrow (\Phi, CMHT, Sign_{sk_c}(l(w_r)||T||n))$ is an algorithm run by the client. It takes as input the client's private key sk_c and a data file $F = \{m_i\}$. The output contains (1) a set of block signatures, $\Phi = \{\sigma_i\}$, as defined in Section IV-A, (2) a CMHT constructed based on the block tags $\{t_i\}$, where $t_i = H(m_i)||c_i$, and (3) a digital signature on the meta data, $Sign_{sk_c}(l(w_r)||T||n)$. The client sends the output and F to the server.
- $GenMeta(sk_s, pk_c, sig_{sk_c}(l(w_r)||T||n)) \rightarrow \mathcal{M}_s$ is executed by the server. After verifying the correctness of the signature $Sign_{sk_c}(l(w_r)||T||n)$ using pk_c, the server signs the signature using its private key sk_s: $\sigma_{\mathcal{M}} = Sign_{sk_s}(Sign_{sk_c}(l(w_r)||T||n))$. Then the server sends the meta data $\mathcal{M}_s = \{l(w_r), T, n, \sigma_{\mathcal{M}}\}$ to the client.
- $Contract(pk_c, pk_s, \mathcal{M}_s, l(w_r)||T||n) \rightarrow \mathcal{M}_c$ is an algorithm run by the client. After verifying the correctness of the signature $\sigma_{\mathcal{M}}$ using pk_s and pk_c, the client deletes all local copies of data and only stores a copy of meta data $\mathcal{M}_c$, identical to $\mathcal{M}_s$.
- $GenChallenge(n) \rightarrow \mathcal{R}_k$ is an algorithm executed by the client. It takes n as input, and outputs a request $\mathcal{R}_k$ which contains a set of k randomly-selected coordinates as well as k randomly-selected constants. The client sends $\mathcal{R}_k$ to the server and asks the server to return a proof that it has the blocks whose coordinates are in $\mathcal{R}_k$.
- $GenProof(\mathcal{R}_k, CMHT, F, \Phi) \rightarrow P$ is executed by the server after receiving $\mathcal{R}_k$. The input contains the request $\mathcal{R}_k$, the CMHT, the file F, and the block signatures Φ. The server returns a proof P that allows the client whether it indeed has the blocks in the $\mathcal{R}_k$.
- $VerifyProof(\mathcal{R}_k, P, \mathcal{M}_c) \rightarrow (TRUE, FALSE)$ is an algorithm executed by the client. After receiving the proof P, the client can verify if the server possesses the blocks in $\mathcal{R}_k$ based on its meta data. It outputs $TRUE$ if the proof P passes the verification. Otherwise, it returns $FALSE$.
- $Judge(pk_s, pk_c, \mathcal{E}_c, \mathcal{E}_s) \rightarrow (ClientWin, ServerWin)$ is an algorithm executed by a judge during litigation after the client detects data loss but the server disputes that. It takes as input the public keys of the server and the client, the evidence from the client, $\mathcal{E}_c = \{\mathcal{R}_k, \mathcal{M}_c\}$, and the evidence from the server, $\mathcal{E}_s = \{P, \mathcal{R}_k, \mathcal{M}_s\}$. The requests $\mathcal{R}_k$ in both $\mathcal{E}_c$ and $\mathcal{E}_s$ must be the same. It decides whether the client wins or the server wins.
- $UpdateRequest() \rightarrow \mathcal{R}_U$ is an algorithm executed by the client. It outputs an update request $\mathcal{R}_U$ which contains an update $Order \in \{Modify, Insert, Delete\}$ and a block *Location* in form of data key or byte offset. If the $Order$ is $Modify$ or $Insert$, $\mathcal{R}_U$ should contain a new block m^* and its signature σ^*.
- $Update(F, \Phi, CMHT, \mathcal{R}_U) \rightarrow P_{update}$ is an algorithm run by the server. After receiving the update request $\mathcal{R}_U$ from the client, the server takes F, Φ, and the CMHT as input. It performs the update, outputs a proof P_{update}, and sends the proof back to the client.
- $VerifyUpdate(P_{update}) \rightarrow (TRUE, FALSE)$ is executed by the client. It takes the proof P_{update} as input and outputs $TRUE$ if the the proof passes the verification. Otherwise, the client will return $FALSE$.

B. Our Scheme

Our scheme has three components. 1) *Preprocessing*: Before outsourcing a file to the server, the client generates the meta data that it keeps locally as well as the information that it outsources to the server together with the file. 2) *Data-possession Verification*: After outsourcing, the client will periodically check the integrity of its remotely-stored data. 3) Updating: When needed, the client sends the server requests to update the file. After each update, the server will prove to the client that the update is correctly executed.

1) Preprocessing: It includes four algorithms: Gen, $Prepare$, $GenMeta$ and $Contract$. Before outsourcing data to the server, the client generates the set Φ of block signatures and the CMHT. It agrees on a time stamp T with the server, and produces the meta data. Then, it outsources F, Φ, and the CMHT to the server, only keeping the meta data locally.

2) Data-possession verification: It is performed periodically, including four algorithms: $GenChallenge$, $GenProof$, $VerifyProof$ and optionally $Judge$. The client queries the server with a randomly-choosing subset of coordinates, and the server generates a proof in response. After verifying the proof, the client will return $TRUE$ or $FALSE$.

• *Generate a Request*: Knowing the value of n, the client knows the exact shape of the complete tree of CMHT. Hence, it knows all valid coordinates for leaf nodes. The client randomly selects $k(<< n)$ leaf nodes, whose coordinates are denoted as $\{c_{i_1}, c_{i_2}, ..., c_{i_k}\}$, for data blocks $\{m_{i_1}, m_{i_2}, ..., m_{i_k}\}$ that are represented by the selected leaf nodes. For convenience, we let $\Omega = \{i_1, i_2, ..., i_k\}$, and the set of selected coordinates is $\{c_i \mid i \in \Omega\}$. The corresponding data blocks are $\{m_i \mid i \in \Omega\}$. The client generates a request $\mathcal{R}_k = \{(c_i, v_i) \mid i \in \Omega\}$, where v_i is a constant. It sends $\mathcal{R}_k$ to the server.

• *Generate a Proof*: After receiving the request, the server generates a proof P. It computes

$$\mu = \sum_{i\in\Omega} v_i m_i, \qquad \sigma = \prod_{i\in\Omega} {\sigma_i}^{v_i} \mod N,$$

where σ_i is the block signature of m_i.

The proof P sent to the client consists of μ, σ, and a partial CMHT, denoted as Γ, consisting of the leaf nodes w_j with the selected coordinates c_i and the siblings of the nodes on the paths from the root to w_j. For each node in Γ, the server only needs to send the label of the node. If a node is a leaf, the label is simply the tag of the block represented by the leaf. Hence, the tags t_i of blocks with coordinates c_i are carried by Γ.

• *Verify*: After receiving the proof P, the client will run the algorithm $VerifyProof$ to check the correctness of the proof. The client first verifies the integrity of the partial CMHT by the standard Merkle tree operations, which ensures the correctness of the tags t_i carried by the leaf nodes of Γ. The client then checks whether the following equation holds:

$$\sigma = (\prod_{i\in\Omega} {t_i}^{v_i}) \cdot g^{\mu} \mod N. \tag{1}$$

If it does, the client is sure that the server possesses the queried blocks correctly, as we will prove later in our security analysis. Otherwise, the server fails to prove it has all the data intact.

• *Judge*: If the client detects data loss through $VerifyProof$ but the server disputes it, they may present their evidences to a court where $Judge$ is executed. After receiving the evidence $\mathcal{E}_c = \{\mathcal{R}_k, \mathcal{M}_c\}$ from the client and the evidence $\mathcal{E}_s = \{P, \mathcal{M}_s\}$ from the server, the judge first verifies the correctness of the proof P through $VerifyProof$ based on information from the client. Then it checks the signatures in $\mathcal{M}_c$ and $\mathcal{M}_s$. If both signatures are correct, it compares the time stamps to determine whose evidence is valid. Depending on whose time stamp is more recent, the judge decides the winner based on the algorithm in Fig. 3.

```
Input: $pk_c$, $pk_s$, $\mathcal{E}_c = \{\mathcal{R}_k, \mathcal{M}_c\}$, $\mathcal{M}_c = \{l_c(w_r), T_c, n_c, \sigma_{\mathcal{M}_c}\}$,
       $\mathcal{E}_s = \{P, \mathcal{M}_s\}$, $\mathcal{M}_s = \{l_s(w_r), T_s, n_s, \sigma_{\mathcal{M}_s}\}$

1.  if ($VerifyProof(\mathcal{R}_k, P, \mathcal{M}_c) = TRUE$)
2.      return server as the winner;
3.  else
4.      if ($VerifySign_{pk_s,pk_c}(\sigma_{\mathcal{M}_c}, l_c(w_r)||T_c||n_c) = FALSE$)
5.          return server as the winner;
6.      if ($VerifySign_{pk_s,pk_c}(\sigma_{\mathcal{M}_s}, l_s(w_r)||T_s||n_s) = FALSE$)
7.          return client as the winner;
8.      else
9.          if ($\sigma_{\mathcal{M}_c} = \sigma_{\mathcal{M}_s}$)
10.             return client as the winner;
11.         else if ($T_s > T_c$)
13.             return server as the winner;
14.         else
15.             return client as the winner;
```

Fig. 3. Algorithm for Judge

3) Updating: If the client wants to update a block, it first runs algorithm $UpdateRequest() \rightarrow \mathcal{R}_U$ to generate an update request and send the request to the server. Upon receiving the request, the server will update the block and execute the algorithm $Update(F, \Phi, CMHT, \mathcal{R}_U)$ to generate a proof P_{update}. The client will use the algorithm $VerifyUpdate(P_{update})$ to verify the update. If the update is correct, the client and the server will agree on a new meta data using algorithms $GenMeta$ and $Contract$.

• *Modification*: Suppose a client wants to change a data block m_i to m^*. It generates an update request $\mathcal{R}_U = \{Modify, Location, m^*\}$, and sends it to the server, which will replace the old data m_i with the new one m^*. Let c_i be the coordinate of m_i. The server constructs a partial CMHT (denoted as Γ) to the leaf node w_j at coordinate c_i, updates w_j with a new label $H(m^*)||c_i$, and recomputes the labels of the nodes on the path from w_j to the root. Let $l(w_{r'})$ be the new label of the root. After that, the server generates a proof $P_{update} = \{\Gamma, c_i, l(w_{r'})\}$, where Γ is the partial CMHT to w_j *before modification*.

To ensure that Γ has the correct leaf node w_j for m_i, the client checks whether the received label of w_j (before modification) contains $H(m_i)$ and verifies the labels of the partial CMHT using the Merkle tree operations. Then it performs what the server does: replacing the label of w_j with $H(m^*)||c_i$ and re-computing the labels of nodes on the path to the root, whose new label is denoted as $l_c(w_{r''})$. The modification is successful only if $l_c(w_{r''})$ is equal to the received value of $l(w_{r'})$. In this case, the client will send the new homomorphic signature σ^* to the server and generate a new meta data with the server, where $\sigma^* = H(m^*)||c_i \cdot g^{m^*} \mod N$.

• *Insertion*: Suppose a client wants to insert a new block m^*. It will inform the server to insert m^* into the file at a specified offset location, insert a corresponding leaf node w^* into the CMHT, and update the meta data. The location in the CMHT where w^* will be inserted is determined by finding the coordinate c of the leftmost leaf node w_i with minimum level, where the *level* of a node is defined as the length of the path from the node to the root. We call w_i the *split node*; its location is where we will insert w^*. See the left plot of

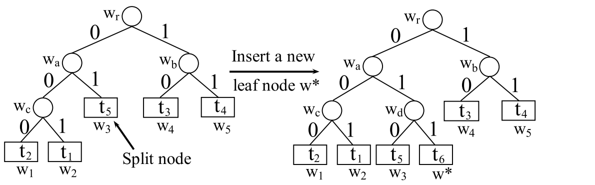

Fig. 4. Insert a new leaf node w^* into the CMHT, where w_3 is the split node

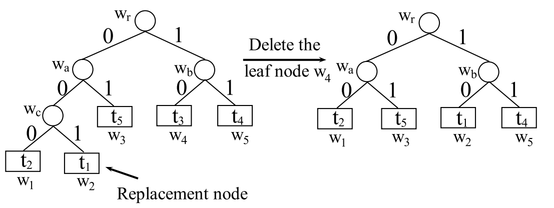

Fig. 5. Delete the leaf node w_4 from the CMHT

Figure 4 for an example. Both the server and the client can independently determine c. Recall that the client knows the shape of the complete binary tree based on the value of n.

The client generates an update request $\mathcal{R}_U = \{Insert, Location, m^*, \sigma^*\}$, and send it to the server. After receiving the request, the server will insert the block into the file, constructs a partial CMHT (denoted as Γ) to the split node w_i at coordinate c, and then inserts a leaf node w^* as follows: replacing w_i with a new internal node w_d, and making w_i and w^* to be the left and right children of w_d, respectively. (See Figure 4 for a simple, illustrative example.) The server adds m^* and w^* into its data structure that maps between data blocks and their leaf nodes in the CMHT. Finally, it updates the labels of all nodes on the path from w^* to the root. Let $l(w_{r'})$ be the new label of the root.

The server generates a proof $P_{update} = \{\Gamma, l(w_{r'})\}$ and sends it to the client, where Γ is the partial CMHT to the split node before insertion. For each node in the partial CMHT, the server only needs to send its label. Recall that the client can independently determine c and thus know the exact shape of this partial CMHT. Using the labels of the nodes and following the standard Merkle tree operations, the client can verify the integrity of the partial CMHT. Next, the client re-performs the same insertion as the server does, and re-computes the label of the root $l(w_{r''})$ based on the partial CMHT after insertion. The insertion is successful only if $l(w_{r'})$ equals to $l(w_{r''})$. In this case, the client will agree on a new time stamp T' with the server, and together they will generate a new meta data with the new root label $l(w_{r'})$.

• *Deletion*: Suppose a client wants to delete a data block m_i. It sends an update request $\mathcal{R}_U = \{Delete, Location\}$ to the server, which deletes m_i from the file, reconstructs a partial CMHT (denoted as Γ) to a leaf node w_j which represents m_i at a certain coordinate c_i, and deletes w_j using the following algorithm: Let w_k be w_j's sibling node. There are two cases. (1) If w_j or w_k is the rightmost leaf node at the highest level, the server deletes w_j and replaces w_j's parent with w_k. (2) Otherwise, it finds the rightmost leaf node w' at the highest level, and expands Γ to that node. In this expanded partial CMHT that covers both w_i and w', the server replaces the parent of w' with its sibling and then moves w' to the location of w_j after removing w_j from the tree. Hence, we call w' the *replacement node*. (See Figure 5 for a deletion example.) The server re-computes the labels in the partial CMHT from leaf(s) to the root. Let $l(w_{r'})$ be the new root value.

Next, the server generates a proof $P_{update} = \{\Gamma, l(w_{r'})\}$, where Γ is the partial CMHT to w_j (possibly also to w') before deletion. Knowing the shape of the CMHT based on the value of n, the client can independently determine the coordinate of the replacement node for case (2) of the deletion algorithm. After receiving the proof, the client first verifies whether the label of w_j contains $H(m_i)$, verifies if the coordinate of the replacement node (if there is one in Γ) is correct, and then verifies the integrity of Γ through the Merkle tree operations. Finally, it performs the same deletion algorithm as the server does. Let $l_c(w_{r''})$ be the new label of the root that the client computes. The deletion is successful only if $l_c(w_{r'})$ equals to $l(w_{r''})$. In this case, the client will generates a new meta data with the server.

C. Client Caching

To improve the client's performance, we may cache the upper levels of the tree on the client side. Due to the nature of a complete binary tree structure, the upper levels account for a very small fraction of the whole tree, but it can significantly reduce the amount of computation and communication between the client and the server. For a file of 10^6 data blocks, if we cache the top 10 levels of CMHT at the client side (which accounts for less than 0.1% of the whole CMHT tree), we can reduce the communication overhead by half.

VI. Security Analysis

Theorem 1: Suppose factoring $N = pq$ is polynomially infeasible for two sufficiently large primes p and q. Given a client request $\mathcal{R}_k$, if the server does not possess one or more data blocks whose coordinates belong to $\mathcal{R}_k$ (due to data loss or corruption), the probability for a proof P produced by the server in polynomial time to pass the client's integrity check $VerifyProof(\mathcal{R}_k, P, \mathcal{M}_c)$ is negligibly small.

Proof: We prove the theorem by contradiction. Recall that $R_k = \{(c_i, v_i) \mid i \in \Omega\}$, and we use m_i to denote the data block that is represented by a leaf node in CMHT whose coordinate is c_i. Assume (1) the server does not possess one or more blocks in $\{m_i \mid i \in \Omega\}$, and yet (2) it has a polynomial method to produce a proof $P = (\mu, \sigma, \Gamma)$ that can pass the client's integrity check with non-negligible probability,

$$\sigma = (\prod_{i \in \Omega} {t_i}^{v_i}) \cdot g^{\mu} \mod N, \tag{2}$$

where $t_i = H(m_i)||c_i$. The integrity of the tags t_i is protected by the Merkle tree operations performed on Γ. Hence, unless the server has a way to break the collision-resistant hash function used by the CMHT, the correct tags for blocks in $\{m_i \mid i \in \Omega\}$ must be used in (2). This prevents the server from using other blocks not in $\{m_i \mid i \in \Omega\}$ and their tags to produce a proof that would make (2) hold.

Recall that μ is supposed to be set to $\sum_{i \in \Omega} v_i m_i$. But because the server does not have one or more of these blocks, it does not know the value of $\sum_{i\in\Omega} v_i m_i$. Hence, the probability for a chosen value μ to equal $\sum_{i\in\Omega} v_i m_i$ will be negligibly small if the data blocks are sufficiently large. In other words, with high probability,

$$\mu \neq \sum_{i\in\Omega} v_i m_i. \tag{3}$$

By definition, $\sigma = \prod_{i\in\Omega} {\sigma_i}^{v_i} \mod N$, and $\sigma_i = t_i \cdot g^{m_i} \mod N$. Applying them to (2), we have

$$\prod_{i\in\Omega}(t_i \cdot g^{m_i})^{v_i} = (\prod_{i\in\Omega} {t_i}^{v_i}) \cdot g^{\mu} \mod N$$
$$\prod_{i\in\Omega} g^{v_i m_i} = g^{\mu} \mod N$$
$$g^{\sum_{i\in\Omega} v_i m_i} = g^{\mu} \mod N.$$

That, together with (3), means we have found $\mu - \sum_{i\in\Omega} v_i m_i \neq 0$ such that $g^{\mu - \sum_{i\in\Omega} v_i m_i} = 1 \mod N$. Therefore, $\mu - \sum_{i\in\Omega} v_i m_i$ can be used to factor N, following Miller's Lemma [14].

The above analysis shows that if the server has a polynomial method to produce a proof that passes integrity check with non-negligible probability, then that method can factor N with non-negligible probability, which contradicts the theorem assumption that factoring a large integer N is polynomially infeasible. ■

Theorem 2: If the client makes false claim about data loss, the server is able to provide non-repudiable evidence that the client have lied.

Proof: Let the client request be $R_k = \{(c_i, v_i) \mid i \in \Omega\}$, and we use m_i to denote the data block that is represented by a leaf node in CMHT whose coordinate is c_i. If the server possesses all blocks in $\{m_i \mid i \in \Omega\}$, it can correctly calculate $\mu = \sum_{i\in\Omega} v_i m_i$. The server can also correctly compute $\sigma = \prod_{i\in\Omega} {\sigma_i}^{v_i} \mod N$. Note that the correctness of the tags $t_i = H(m_i)||c_i$ and the signatures $\sigma_i = t_i \cdot g^{m_i} \mod N$ is verifiable by the server based on m_i. Hence, we have

$$\sigma = \prod_{i\in\Omega} {\sigma_i}^{v_i} = (\prod_{i\in\Omega} {t_i}^{v_i}) \cdot g^{\sum_{i\in\Omega} v_i m_i} = (\prod_{i\in\Omega} {t_i}^{v_i}) \cdot g^{\mu} \mod N.$$

Moreover, the correctness of the CMHT is also completely verifiable by the server through hashing. The server will not sign the meta data at any time when it finds that the CMHT fails the integrity check based on its meta data $\mathcal{M}_s$. Hence, the partial tree Γ produced by the server will also pass the Merkle-tree operations.

In order for the client to make a successful false claim, it has to make sure that $Judge(pk_s, pk_c, \mathcal{E}_c, \mathcal{E}_s)$ does not execute Line 2, 5 or 13 in Figure 3 because otherwise the server will be declared as the winner. To avoid Line 2, the client must provide a false meta data $\mathcal{M}_c$, which is different from $\mathcal{M}_s$, because the latter (together with the proof P also provided by the server) will pass the data-possession verification in Line 1 as we have argued previously. Hence, $\mathcal{M}_c \neq \mathcal{M}_s$.

To avoid Line 5, the client must provide $\mathcal{M}_c$ that was signed by the server such that the signature verification in Line 5 can pass.

The server has signed both $\mathcal{M}_s$ and $\mathcal{M}_c$. Because the server always keeps the latest meta data as $\mathcal{M}_s$, $\mathcal{M}_c$ must have been signed earlier with a smaller timestamp. In this case, Line 13 will be executed, which still declares the server as the winner. Therefore, the evidence provided by the server, $\mathcal{E}_s = \{P, \mathcal{R}_k, \mathcal{M}_s\}$, will non-repudiably result in the judicator declaring the server as the winner when the client makes false claim, regardless of what evidence the client will provide. ■

VII. Evaluation

We use experiments to evaluate the performance of our scheme in terms of communication overhead and computational overhead. The results show that our scheme performs much better than the best existing work, DPDP [7].

Our experiments are performed on a desktop computer with Intel Core i7-3770 @3.40 GHz, 8 GB RAM, and a 2TB hard driver. Algorithms are implemented using C++. We implement the block signature and the hash function using the crypto library of OpenSSL version 1.0.1 [15].

The size of the file used in our experiments is 1GB. After dividing the file into blocks, we measure the communication and computational overhead incurred at the client side and the server side for performing data-possession verification and update operations. We evaluate the performance of our scheme and compare it with DPDP under different block sizes. We let the client cache the upper half of the levels in the CMHT or in the skiplist of DPDP. The cached data is about 0.1% of the tree (or skiplist). For CMHT, when the block size is 1024KB, the total cached data is just 1.25KB; when the block size is 2KB, the total cashed data is 80KB, which is still very small comparing with the file size of 1GB. For DPDP, the amount of cached data is larger because each node in skiplist needs to store an extra rank number.

Note that DPDP does not address false client claims as our scheme does. This is a qualitative difference not included in the quantitative comparison below.

A. Communication overhead

We first compare our scheme and DPDP in terms of communication overhead for data-possession verification over a request $\mathcal{R}_k$ for k data blocks. As proved in [5], detecting a 1% file corruption with 99% confidence needs querying a constant number of 460 blocks. So we set $k = 460$. The dominating overhead is the proof sent from the server to the client. It is not affected by the number of corrupted blocks at the server. We measure both average overhead and maximum overhead. The former is the average over 100 independent runs, each run verifying 460 random selected blocks. The latter is the overhead for the case where $\mathcal{R}_k$ contains 460 leaf nodes with highest levels.

We present the experimental result in Figure 6. The x-axis shows the block size in KB. The y-axis shows the communication overhead in KB. The left plot presents the average overhead, and the right plot presents the maximum overhead. The overhead of our scheme is consistently less than half of the overhead in DPDP. More specifically, our average overhead is $31\% \sim 50\%$ of DPDP's in the left plot, and our maximum overhead is $13\% \sim 46\%$ of DPDP's in the right plot. When the block size is 2KB, our scheme reduces the

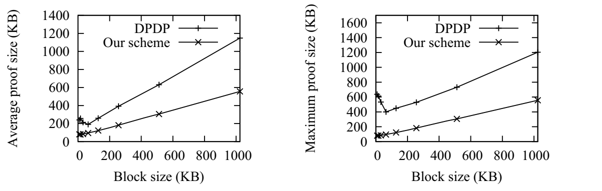
Fig. 6. Comparing our scheme (CMHT) and DPDP in terms of average or maximum communication overhead for data-possession verification with client cache

average (maximum) overhead by 69% (87%) when comparing with DPDP.

Next, we measure the communication overhead between the client and the server for updating a data block. It includes all information sent by $UpdateRequest$, $Update$, and $VerifyUpdate$. We perform query, insert, delete, and modify once for every data block of the file to measure the average communication overhead. For all operations, when we work on one block, all other data blocks of the file are assumed to be present in the server, and so does their corresponding leaf nodes in CMHT. (Using deletion as an example, we will delete one block at a time. Before we delete the next block, we put back the previously deleted one.) For insertion and modification, we do not account the transmission of the new block as overhead.

The experimental results are shown in Figure 7. The x-axis is the block size in logarithmic scale. The y-axis is the communication overhead in KB. The average overhead of our scheme is significantly lower than that of DPDP — less than one third of it when the block sizes are relatively small. The gap for the maximum communication overhead is even larger, which we omit due to space limitation.

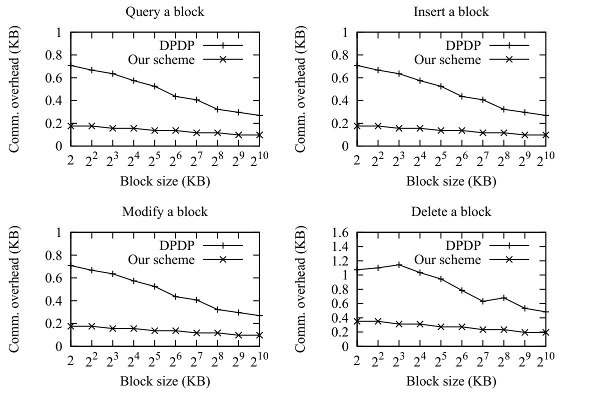
Fig. 7. Comparing our scheme and DPDP in terms of average communication overhead for updating a block with client cache

B. Computational overhead

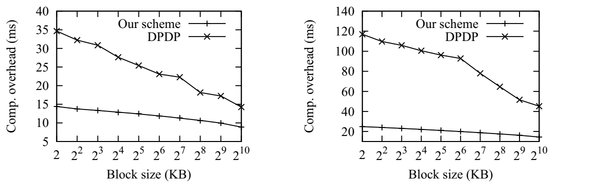
Fig. 8. Average and maximum computational overheads by a client to verify a proof with client cache

We measure the computational overhead for a client to verify a proof returned from the server for data-possession verification, including both the verification of (1) and the Merkle tree operations on the partial CMHT tree. We make 100 randomly-generated data possession verification requests and measure the average and maximum computational overhead per request. The results are presented in Figure 8. The x-axis shows the block size in logarithmic scale. The y-axis shows the computational overhead in second. Again, our scheme performs better in the average case as well as the maximum case. More specifically, our scheme reduces the average computational overhead by up to 58.5% and the maximum computational overhead by up to 78.7%, when comparing with DPDP.

VIII. Conclusion

The development of cloud storage systems brings a number of security problems. This paper proposes a non-repudiable data possession verification scheme that protects both the client and the server. The new scheme makes sure that the server cannot cheat the client by lying about data loss, and the client cannot untruthfully claim data loss. We also design a new data structure named Coordinate Merkle Hash Tree (CMHT) to optimize the communication and computational overhead. We compare our scheme with previous work through experiments, and the result shows that our scheme has better performance.

IX. Acknowledgements

This work was supported in part by Cisco Systems, the US National Science Foundation under grant CNS-1115548, and the National Natural Science Foundation of China under grant 61170277.

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," *Proc. of ESORICS*, 2009.
[2] H. Shacham and B. Waters, "Compact Proofs of Retrievability," *Proc. of ASIACRYPT*, 2008.
[3] A. Juels and B. Kaliski Jr., "Pors: Proofs of Retrievability for Large Files," *Proc. of ACM CCS*, 2007.
[4] K. D Bowers, A. Juels, and A. Oprea, "Proofs of Retrievability: Theory and Implementation," *Proc. of ACM CCSW*, 2009.
[5] G. Ateniese, R. Burns, R. Curtmola, J. Herring, L. Kissner, Z. Peterson, and D. Song, "Provable Data Possession at Untrusted Stores," *Proc. of ACM CCS*, 2007.
[6] G. Ateniese, R. Di Pietro, L. V Mancini, and G. Tsudik, "Scalable and Efficient Provable Data Possession," *Proc. of SecureComm*, 2008.
[7] C. Erway, A. Küpçü, C. Papamanthou, and R. Tamassia, "Dynamic Provable Data Possession," *Proc. of ACM CCS*, 2009.
[8] W. Pugh, "Skip lists: A Probabilistic Alternative to Balanced Trees," *Communications of the ACM*, 1990.
[9] R. Merkle, "A Digital Signature Based on a Conventional Encryption Function," *Journal of CRYPTO*, 1987.
[10] Z. Mo, Y. Zhou, and S. Chen, "A Dynamic Proof of Retrievability (PoR) Scheme with O(logn) Complexity," *Proc. of IEEE ICC*, 2012.
[11] M. Blanton Y. Zhang, "Efficient Dynamic Provable Possession of Remote Data via Balanced Update Trees," *AsiaCCS*, 2013.
[12] D. Boneh, B. Lynn, and H. Shacham, "Short Signatures from the Weil Pairing," *Proc. of ASIACRYPT*, 2001.
[13] J. Katz and Y. Lindell, *Introduction to Modern Cryptography*, 2007.
[14] G. Miller, "Riemann's Hypothesis and Tests for Primality," *Proc. of ACM STOC*, 1975.
[15] "Openssl 1.0.1," *http://www.openssl.org/*, Feb. 2013.