Efficient Dynamic Provable Data Possession from Dynamic Binary Tree

Changfeng Li[1] and Huaqun Wang[2](✉)

[1] Nanjing University of Finance and Economics, Nanjing, China
[2] Nangjing University of Posts and Telecommunications, Nanjing, China
wanghuaqun@aliyun.com

Abstract. In order to ensure the remote data integrity in cloud storage, provable data possession (PDP) is of crucial importance. For most clients, dynamic data operations are indispensable. This paper proposes an efficient dynamic PDP scheme for verifying the remote dynamic data integrity in an untrusted cloud storage. Our dynamic PDP scheme is constructed from dynamic binary tree and bilinear pairings, supporting the dynamic data operations, such as, insertion, deletion, modification. From the computation cost, communication cost, and storage cost, our proposed dynamic PDP scheme is efficient. On the other hand, our proposed concrete dynamic PDP scheme is provably secure.

Keywords: Cloud computing · Dynamic provable data possession · Binary tree

1 Introduction

By using cloud computing, the clients are relieved of the burden for storage management and data processing. Thus, the clients save the capital expenditure on hardware, software, and personnel maintenances, *etc.* In cloud computing, the clients outsource their computing and storage to remote cloud server (CS). At the same time, the clients also face the risks of confidentiality, integrity and availability of data and service. Since the clients do not store these data locally, it is especially vital to ensure their remote data integrity. In 2007, Ateniese *et al.* proposed an important remote data integrity checking primitive: PDP [1]. It is a probabilistic remote data integrity checking primitive. For PDP, the verifier can efficiently check remote data integrity with a high probability. Following Ateniese *et al.*'s pioneering work, Shacham and Waters presented the proof of retrievability (POR) scheme [2].

For most clients, their stored data is dynamic. The clients may frequently insert or delete or modify their remote data. Thus, dynamic PDP is indispensable to ensure remote dynamic data integrity. On the other hand, the dynamic

H. Wang—This work is partly supported by the Natural Science Foundation of China through projects (61272522), by the Program for Liaoning Excellent Talents in University through project (LR2014021), and by the Natural Science Foundation of Liaoning Province (2014020147).

PDP scheme must be efficient for capacity-limited end devices. Zheng *et al.* proposed the fair and dynamic POR [3]. Then, Ateniese *et al.* proposed dynamic PDP model and designed the concrete scheme although it does not support insert operation [4]. In 2009, based on the skip list, Erway *et al.* designed a full-dynamic PDP scheme which supports the insert operation [5]. In 2013, Etemad *et al.* proposed the transparent, distributed, and replicated dynamic PDP scheme [6]. Cash *et al.* proposed the dynamic proofs of retrievability via oblivious RAM [7]. On the other hand, dynamic remote data public auditability has also been studied [8–10].

Tree is an important storage structure for the remote block data. In 2013, Zhang *et al.* propose a dynamic provable scheme via balanced update tree [11]. Zhang *et al.* propose a verifiable dynamic provable data possession scheme by developing a variant authenticated 2–3 tree [12]. Shi *et al.* pointed out that Cash *et al.*'s scheme [7] is mostly of theoretical interest because it employs oblivious RAM as a black box. They also pointed out Stefanov *et al.*'s scheme has a large audit cost. Finally, they proposed a dynamic proof of retrievability scheme with constant client storage whose bandwidth cost is comparable to a Merkle hash tree [13]. Tate *et al.* proposed multi-user dynamic proofs of data possession by using trusted hardware [14].

Until now, the proposed dynamic PDP schemes are inefficient. When a novel block is inserted, many blocks have to change their index and create novel tags. It will incur heavy cost. It is an open problem to keep the other block-tag pairs unchanged even if the novel block is inserted.

1.1 Contributions

The main contributions of this work are summarized below: (1) We present a dynamic binary tree construction method which yields an efficient dynamic PDP scheme; (2) Based on the bilinear pairing and our proposed dynamic binary tree, a concrete private dynamic PDP scheme is designed. (3) Our private dynamic PDP scheme can detect the dishonest client's invalid data.

1.2 Paper Organization

This paper is organized below. Section 2 presents the construction method of dynamic binary tree. It comprises of insertion and deletion. Section 3 gives the models of dynamic PDP: system model, security model and the definition of dynamic PDP scheme. Based on the dynamic binary tree, Sect. 4 propose a concrete dynamic PDP scheme. The performance analysis is also given in this section. Section 5 analyzes the proposed dynamic PDP scheme's security. Finally, Sect. 6 gives the conclusion.

2 Dynamic Binary Tree

In order to realize the remote data integrity checking, the corresponding tag T_i must be generated and uploaded for every block m_i. In our scheme, the tag T_i relates to m_i and the leaf node index l_i. From the procedures Insertion and Deletion, the leaf nodes' index will keep constant after dynamic operations. When the data blocks are stored on the leaf nodes, these blocks' leaf node index will also keep constant after the dynamic operations. Thus, when some blocks are inserted, the other block-tag pairs will be unchanged. This dynamic binary tree can be used to support our private dynamic PDP scheme.

2.1 Binary Tree

A simple binary tree can be depicted in Fig. 1. The top node R (level 0) is the root of the tree. R has two children $1L$ and $1R$ (level 1). Continuously, $1L$ has two children 1 and 2; $1R$ has two children 3 and 4 (level 2). The level 2 is the bottom level which will store the clients' remote data. Specially, denote R's left child as 0 and right child as 1. 0's left child is 00 and right child is 01. 1's left child is 10 and right child is 11. Generally, every node's index is its parent's index plus 0 if it is the left child or plus 1 if it is the right child. Thus, Fig. 1 can be rewritten as Fig. 2. In the Fig. 2, every inner node has two elements: index and leaf node number. For example, the root node R has 4 children which lie on the bottom level. R's left child has 2 children which lie on the bottom level. R's right child has 2 children which lie on the bottom level. In the dynamic binary tree, the bottom nodes may be deleted or modified. On the other hand, some novel nodes can be inserted at any place on the bottom level. On the bottom level, the 4 leaf nodes's indexes are 00, 01, 10, 11, respectively.

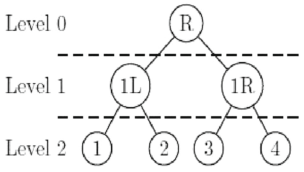

Fig. 1. Binary tree

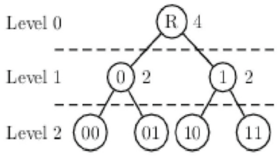

Fig. 2. Binary tree with index and leaf node number

2.2 Insertion

When a leaf node N_i is inserted after another leaf node N, the binary tree can be updated below:

1. From the node N, the left child and the right child are created and become the leaf nodes. Their parent node N becomes the inner-node.
2. The inner node N's index keeps unchanged and N's left child has the same index as the node N. N's right child's index is N's index plus 1.
3. The inner node N's leaf node number is 2. N's parent, grandfather, until to the root R, add their original leaf node number to 1 which is their new leaf node number.
4. The other nodes' index and leaf node number keep unchanged.

An inner node $< l, v >$ is associated with its index l and its leaf node number v. The leaf node only has the index, *i.e.*, it is denoted as l.

2.3 Deletion

Let a leaf node N's index be l'. Let N's parent node N_p be $< l, v >$. Let N's brother leaf node be l''. When l' is deleted, the binary tree can be updated below:

1. The parent node $< l, v >$ is substituted by the index l'' without the leaf node number. The parent node becomes the leaf node.
2. The two leaf nodes l' and l'' are deleted.
3. N_p's parent, grandfather, until to the root R, subtract 1 from their original leaf node number. The difference values are their new leaf node number.
4. The other inner nodes' index and leaf node number keep unchanged. The other leaf nodes' index keeps unchanged.

3 Model of Dynamic PDP

In our dynamic PDP, there exist two different entities: client and CS. Client's massive data will be stored in CS. CS has significant storage space and computation resource which are used to process the clients' data.

Definition 1 (Dynamic PDP). *Dynamic PDP scheme consists of the phases below. They can be performed in the PPT (probabilistic polynomial time).*

1. *$KeyGen(1^k) \rightarrow (sk, pk)$. Input a security parameter 1^k, it outputs the secret/public key pair (sk, pk). By using $KeyGen(1^k)$, the client gets his secret/public key pair (sk_c, pk_c) and CS gets his secret/public key pair (sk_s, pk_s).*
2. *$TagGen(sk_c, pk_c, pk_s, m) \rightarrow T_m$. Input (sk_c, pk_c), pk_s and the message block m, it outputs the tag T_m.*
3. *$VryTag(sk_s, pk_s, pk_c, m, T_m) \rightarrow$ accept or reject. Input the block-tag pair (m, T_m), CS's secret/public key pair (sk_s, pk_s), the client's public key pk_c, it outputs accept or reject. accept denotes the block-tag pair is valid and reject denotes the block-tag pair is invalid.*
4. *$PreUpdate(sk_c, pk_c, pk_s, F, info, M_e) \rightarrow \{e(F), e(info), e(M'_e)\}$. Input (sk_c, pk_c), pk_s, the file block F, the update information $info$, the previous metadata M_e, it outputs the encoded version of the file $e(F)$, $e(info)$, and the new metadata $e(M'_e)$. At last, the client sends $e(F), e(info)$ to CS and stores M'_e locally.*
5. *$PerUpdate(sk_s, pk_c, pk_s, F_{i-1}, M_{i-1}, e(F), e(info)) \rightarrow \{U, P_U, F_i, M_i\}$. Input pk_c, (sk_s, pk_s), the previous version of the stored file F_{i-1}, the metadata M_{i-1} and the query $(e(F), e(info))$, it outputs the new version of the file F_i and the metadata M_i, along with the update report U and its proof P_U. CS stores F_i, M_i and sends (U, P_U) to the client.*

6. *VryUpdate*$(sk_c, pk_c, pk_s, F, info, M_e, U, P_U) \rightarrow$ *accept or reject. Input* (sk_c, pk_c), pk_s, $(F, info)$, M_e *and* (U, P_U), *it outputs accept or reject. accept denotes CS's update response is valid. reject denotes CS's update response is invalid.*
7. *Challenge*$(sk_c, pk_c, pk_s, M_e) \rightarrow chal$. *Input* (sk_c, pk_c), pk_s, M_e, *it outputs the challenge chal to CS.*
8. *Prove*$(sk_s, pk_s, pk_c, F_i, M_i, chal) \rightarrow V$. *Input* (sk_s, pk_s), pk_c, *the latest version of the file* F_i *and the metadata* M_i, *and chal, it outputs the proof* V *to the client.*
9. *Verify*$(sk_c, pk_c, pk_s, M_e, chal, V) \rightarrow$ *accept or reject. Input* (sk_c, pk_c), pk_s, M_e, *chal, and the proof* V, *it outputs accept or reject. accept means that CS still keeps the file intact. reject means some challenged blocks are corrupted.*

Definition 2. *A dynamic PDP scheme is secure against any untrusted PPT CS if the probability that any such CS wins the dynamic PDP game below is negligible. The untrusted CS is the adversary* $\mathcal{A}$. *The client is the challenger* $\mathcal{C}$. *The dynamic PDP game is played between* $\mathcal{C}$ *and* $\mathcal{A}$ *below:*

1. *KeyGen:* $\mathcal{C}$ *runs KeyGen*$(1^k) \rightarrow (sk_c, pk_c)$ *and gets its own secret/public key pair* (sk_c, pk_c). $\mathcal{A}$ *runs KeyGen*$(1^k) \rightarrow (sk_s, pk_s)$ *and gets its own secret/public key pair* (sk_s, pk_s). *The public keys* pk_c *and* pk_s *are made public.*
2. *First-phase Queries:* $\mathcal{A}$ *adaptively makes a lot of different queries to* $\mathcal{C}$. *Each query can be one of the following:*
   (a) *Update queries.* $\mathcal{A}$ *sends the update query to* $\mathcal{C}$ *adaptively.* $\mathcal{C}$ *responds* $\mathcal{A}$ *according to the query.*
   (b) *Hash queries.* $\mathcal{A}$ *can make hash queries adaptively.* $\mathcal{C}$ *returns the corresponding hash values to* $\mathcal{A}$.
   (c) *Tag queries.* $\mathcal{A}$ *makes block-tag pair queries adaptively. For a block query* m_i, $\mathcal{C}$ *computes the tag* $T_i \leftarrow TagGen(sk_c, m_i)$ *and sends it to* $\mathcal{A}$.

   Without loss of generality, let (m_i, T_i) *be the queried block-tag pair or updated block-tag pair where* $i \in \mathbb{I}_1$.
3. *Challenge:* $\mathcal{C}$ *generates a challenge chal for* $\mathcal{A}$. *Let the challenged block subscript set satisfy* $\{i_1, i_2, \cdots, i_l\} \nsubseteq \mathbb{I}_1$, *where* l *is a positive integer.* $\mathcal{A}$ *is required to provide a possession proof for the blocks* $m_{i_1}, m_{i_2}, \cdots, m_{i_l}$.
4. *Second-Phase Queries. Similar to the First-Phase Queries. Let* (m_i, T_i) *be the queried (Update queries or Tag queries) and responded block-tag pairs where the subscript* $i \in \mathbb{I}_2$ *and* $\mathbb{I}_2$ *is the queried and responded block-tag pair subscript set in Second-Phase. The restriction is that* $\{i_1, i_2, \cdots, i_l\} \nsubseteq \mathbb{I}_1 \cup \mathbb{I}_2$.
5. *Forge:* $\mathcal{A}$ *computes the remote data possession proof* V *for the blocks indicated by chal and outputs* V.

We say that a dynamic PDP scheme satisfies unforgeability against the untrusted CS if the adversary $\mathcal{A}$ *wins the dynamic PDP game with negligible probability.*

4 The Proposed Dynamic PDP Scheme

4.1 Bilinear Pairings

Let $\mathcal{G}_1$ and $\mathcal{G}_2$ be two cyclic multiplicative groups with the same prime order q. Let $e : \mathcal{G}_1 \times \mathcal{G}_1 \rightarrow \mathcal{G}_2$ be a bilinear map [15,16] which satisfies the following properties:

1. Bilinearity: $\forall g_1, g_2 \in \mathcal{G}_1$ and $a, b \in \mathcal{Z}_q$, $e({g_1}^a, {g_2}^b) = e(g_1, g_2)^{ab}$.
2. Non-degeneracy: $\exists g_4, g_5 \in \mathcal{G}_1$ such that $e(g_4, g_5) \neq 1_{\mathcal{G}_2}$.
3. Computability: $\forall g_6, g_7 \in \mathcal{G}_1$, there is an efficient algorithm to calculate $e(g_6, g_7)$.

Definition 3 (CDH problem). *Let g be the generator of $\mathcal{G}_1$. Given $g, g^a, g^b \in \mathcal{G}_1$ for randomly chosen $a, b \in \mathcal{Z}_q$, calculate $g^{ab} \in \mathcal{G}_1$.*

Definition 4 (DDH problem). *Let g be the generator of $\mathcal{G}_1$. Given $(g, g^a, g^b, \hat{g}) \in \mathcal{G}_1^4$ for randomly chosen $a, b \in \mathcal{Z}_q^*$, decide whether $g^{ab} \overset{?}{=} \hat{g}$.*

In the paper, the chosen group $\mathcal{G}_1$ satisfies that CDH problem is difficult but DDH problem is easy. The DDH problem can be solved by using the bilinear pairings. Thus, $(\mathcal{G}_1, \mathcal{G}_2)$ are also defined as GDH (Gap Diffie-Hellman) groups.

4.2 The Concrete Dynamic PDP Scheme

Initially, suppose the maximum number of the stored block-tag pairs is $\hat{n}$. Let f be a pseudo-random function, Ω be a trapdoor function whose first parameter is the trapdoor, π be a pseudo-random permutation, and h be a cryptographic hash function which are given below.

$$f : \mathcal{Z}_q^* \times \{1, 2, \cdots, \hat{n}\} \rightarrow \mathcal{Z}_q^*, \ \ \Omega : \mathcal{G}_1^* \times \{1, 2, \cdots, \hat{n}\} \rightarrow \mathcal{Z}_q^*$$
$$h : \mathcal{Z}_q^* \rightarrow \mathcal{G}_1^*, \ \ \pi_{\bar{n}} : \mathcal{Z}_q^* \times \{1, 2, \cdots, \bar{n}\} \rightarrow \{1, 2, \cdots, \bar{n}\}$$

The client will upload the large message M to CS. In order to generate the corresponding tags, M (maybe encoded by using error-correcting code or encryption algorithm) is divided into n blocks $(m_1, m_2, \cdots, m_n)$ where $m_i \in \mathcal{Z}_q^*$. Without loss of generality, we denote $M = (m_1, m_2, \cdots, m_n)$. CS picks a random number $sk_s \in \mathcal{Z}_q^*$ as its secret key and computes its public key $pk_s = g^{sk_s}$. The client picks a random number $sk_c \in \mathcal{Z}_q^*$ as its secret key and computes its public key $pk_c = g^{sk_c}$. The client also picks a random point $u \in \mathcal{G}_1$ and makes u public.

TagGen(sk_c, pk_c, pk_s, m_i): The client creates the full binary tree with the depth $\lceil \log_2 n \rceil$. From the left, we denote the i-th leaf node index as l_i. For the block m_i which will be stored on the i-th leaf node, the client computes $W_i = \Omega(pk_s^{sk_c}, l_i)$, $T_i = (h(W_i)u^{m_i})^{sk_c}$. Client outputs the block-tag pair (m_i, T_i).

The above procedure is performed n times and all the block-tag pairs are generated. The client uploads $\Sigma = \{(m_1, T_1), \cdots, (m_n, T_n)\}$ to CS. CS creates the full binary tree with the depth $\lceil \log_2 n \rceil$ which is the same as the client's full binary tree. CS stores the block-tag pair (m_i, T_i) on the i-th leaf node from the left whose index is l_i.

VryTag$(sk_s, pk_s, pk_c, m_i, T_i)$: CS searches for the i-th leaf node from the left and gets its index l_i.

1. CS computes $\hat{W}_i = \Omega(pk_c^{sk_s}, l_i)$;
2. CS verifies whether $e(T_i, g) = e(h(\hat{W}_i)u^{m_i}, pk_c)$ holds: if it holds, CS accepts and stores it on the i-th leaf node from the left; otherwise, CS rejects it.

PreUpdate$(sk_c, pk_c, F, info, M_e)$: The client prepares to update the block. The update information is denoted as $info$ (*e.g.*, delete block, insert block, modify block). In order to simplify the procedure, the encoding and encrypting are omitted. Then, the client performs the procedures below:

1. If the update is insertion, the client updates its dynamic binary tree according to Sect. 2.B: insertion. Suppose F is inserted after the leaf node l_N. From the updated dynamic binary tree, the client gets the index l_{N+1} which is after the leaf node l_N. The client computes

$$W_{N+1} = \Omega(pk_s^{sk_c}, l_{N+1}),\ T_{N+1} = (h(W_{N+1})u^F)^{sk_c}$$

   The client outputs the block-tag pair (F, T_F). The original metadata M_e is also modified into the latest metadata M_e'. The client uploads $(F, T_F, info)$ to CS.
2. If the update is deletion, the client updates its dynamic binary tree according to Sect. 2.C: deletion. The original metadata M_e is updated into the latest metadata M_e'. The client uploads $(F, info)$ to CS.
3. If the update is modification, *i.e.*, the block-tag pair (m_i, T_i) is modified into (F, T_F) on the same leaf node with the same index l_i. The client computes

$$W_i = \Omega(pk_s^{sk_c}, l_i),\ T_F = (h(W_i)u^F)^{sk_c}$$

   The client outputs the block-tag pair (F, T_F). The original metadata M_e is also updated into the latest metadata M_e'. The client uploads $(F, T_F, info)$ to CS.

PerUpdate$(pk_c, pk_s, sk_s, F, info, T_F)$: Upon receiving the updating query, the corresponding leaf node (which will be inserted or modified or deleted) can be fleetly searched by using the inner node's parameter v (*i.e.*, the number of the leaf node which are the inner node's children), CS performs the procedures below:

1. If the update is insertion, CS updates its dynamic binary tree according to Sect. 2.B: insertion. Suppose F is inserted after the leaf node with the index l_N. CS gets the index l_{N+1} which is after the leaf node l_N.
   (a) If (F, T_F) can pass *VryTag*, CS stores them on the leaf node with the index l_{N+1}. Then, CS sends the insertion verification information and the corresponding signature $(Info_U, Sign_{sk_s}(Info_U))$ to the client.
   (b) If (F, T_F) can not pass the insertion verification *VryTag*, CS rejects them.

2. If the update is modification and (F, T_F) can pass *VryTag*, CS substitutes (F, T_T) for (m_i, T_i) whose index is l_i. Then, CS sends the modification verification information and the corresponding signature $(Info_U, Sign_{sk_s}(Info_U))$ to the client.
3. If the update is deletion, CS updates its dynamic binary tree according to Sect. 2.B: deletion. Then, CS deletes the corresponding block-tag pair. CS sends the deletion verification information and the corresponding signature $(Info_U, Sign_{sk_s}(Info_U))$ to the client.

VryUpdate$(\{(Info_U, Sign_{sk_s}(Info_U))\})$: Upon receiving the CS's update response $(Info_U, Sign_{sk_s}(Info_U))$ on the update query $(F, T_F, info)$, where T_F is empty for the deletion, the client verifies CS's signature for the update. If it can pass the verification, the client accepts CS's update response; otherwise, the client rejects CS's update response and sends the same query again.

Challenge(sk_c, pk_c, pk_s, M_c): In order to check the remote data integrity, the client sends the challenge $chal = (c, k_1, k_2)$ to CS, where $1 \leq c \leq \hat{n}, k_1, k_2 \in \mathcal{Z}_q$.

Prove$(sk_s, pk_s, pk_c, \Sigma, chal)$: Suppose that $\hat{n}$ block-tag pairs are stored in CS. Upon receiving the challenge $chal = (c, k_1, k_2)$, CS computes: $v_i = \pi_{\hat{n}}(k_1, i), a_i = f(k_2, i)$, for $1 \leq i \leq c$; $T = \prod_{i=1}^{c} T_{v_i}^{a_i}$, $\hat{m} = \sum_{i=1}^{c} a_i m_{v_i}$. CS outputs $V = (\hat{m}, T)$ and sends V to the client.

$Verify(sk_c, pk_c, pk_s, M_e, chal, V)$: Upon receiving the response V from CS, based on the challenge *chal* and the stored metadata, the client performs the procedures below:

1. For $1 \leq i \leq c$, the client computes: $v_i = \pi_{\hat{n}}(k_1, i), a_i = f(k_2, i)$;
2. From the left, the client searches for the v_i-th $(1 \leq i \leq c)$ leaf node from the stored dynamic binary tree. Then, the client gets the corresponding leaf node index l_{v_i} for all v_i $(1 \leq i \leq c)$;
3. For all v_i $(1 \leq i \leq c)$, the client computes $W_{v_i} = \Omega(pk_s^{sk_c}, l_{v_i})$ and checks
$$e(T, g) \overset{?}{=} e(\prod_{i=1}^{c} h(W_{v_i})^{a_i} u^{\hat{m}}, pk_c)$$
If it holds, the client outputs "*accept*"; otherwise the client outputs "*reject*".
4. When CS's response can not pass the client's verification, the client will perform the same challenge many times. If the responses still cannot pass the verification, the client will connect the CS provider to inform it this situation. CS provider will censor the client's stored data and retrieve the lost data from the offline backup. If CS provider fails, the client and the CS provider will evaluate the loss and discuss the reparation according to the loss severity.

Correctness: A dynamic PDP scheme must be workable and correct. That is, if the client and CS are honest and follow the specified procedures, the response V can pass the client's verification. The correctness is given below:

$$\begin{aligned} e(T, g) &= e(\textstyle\prod_{i=1}^{c} T_{v_i}^{a_i}, g) = e(\prod_{i=1}^{c} (h(W_{v_i}) u^{m_i})^{sk_c f(k_2, i)}, g) \\ &= e((\textstyle\prod_{i=1}^{c} h(W_{v_i})^{a_i}) u^{\hat{m}}, pk_c) \end{aligned}$$

4.3 Performance Analysis

First, we analyze the performance of our proposed dynamic PDP scheme from the computation cost and communication cost. We compare our dynamic PDP scheme with the other up-to-date dynamic PDP schemes.

Table 1. Comparison of computation cost

Protocols	Wang [9]	Zhu [10]	Ours
TagGen	$\hat{n}(2C_{exp}+1C_{mul})$	$(s+2\hat{n})C_{exp}+\hat{n}C_{mul}$	$2\hat{n}C_{exp}+\hat{n}C_{mul}$
Prove	$cC_{exp}+(c-1)C_{mul}$	$cC_{exp}+(c-1)C_{mul}$	$cC_{exp}+(c-1)C_{mul}$
Verify	$4C_e+(c+1)C_{exp}$	$3C_e+(c+s)C_{exp}+$	$2C_e+(c+1)C_{exp}+$
	$+cC_{mul}$	$(c+s-2)C_{mul}$	cC_{mul}

Computation: Suppose there are $\hat{n}$ block-tag pairs will be stored in CS. The challenged block number is c. We will consider the computation overhead in the different phases. The multiplication, exponentiation and bilinear pairings contribute most computation cost on the group $\mathcal{G}_1$. Compared with them, the other operations are faster and computation cost is small, such as Hash function, permutation, *etc.* Thus, we only consider the multiplication, exponentiation and bilinear pairings on the group $\mathcal{G}_1$. In the phase *TagGen*, the client performs $2\hat{n}$ exponentiation ($pk_s^{sk_c}$ can be finished in the precomputation once for all) and $\hat{n}$ multiplication on $\mathcal{G}_1$. In the phase of $VryTag$, CS will perform 1 exponentiation, $\hat{n}$ multiplication and $2\hat{n}$ bilinear pairing on $\mathcal{G}_1$. In the phase *PreUpdate*, for one time, the average computation cost is $\frac{1}{4}(3+3)=1.5$ exponentiation and $\frac{1}{4}(1+1)=0.5$ multiplication. In the phase *PerUpdate*, for one time, CS performs 1 signature operation and a *VryTag* operation. In the phase *VryUpdate*, the client needs to verify a signature. In the phase of *Prove*, CS will perform c exponentiation on $\mathcal{G}_1$. In the phase of $Verify$, the client will perform c multiplication, $c+1$ exponentiation and 2 pairings ($pk_s^{sk_c}$ can be finished in the precomputation). On the other hand, in 2011, Wang et al. proposed the first dynamic remote data public auditability scheme in cloud computing [9]. In 2013, Zhu et al. proposed the dynamic audit services for outsourced storages in clouds [10]. The computation comparison can be summarized in Table 1. In Table 1, C_{mul} denotes the time cost of multiplication, C_{exp} denotes the time cost of exponentiation on the group $\mathcal{G}_1$, and C_e denotes the time cost of bilinear pairing. In the above comparison, we omit the computation cost in the phase *VryTag*. In order to guard against the dishonest clients to upload invalid block-tag pairs, CS verifies every block-tag pair. Our scheme has this property while Wang et al.'s scheme [9] and Zhu et al.'s scheme [10] have not this property. Thus, we omit *VryTag* in the above comparison. Our dynamic PDP scheme has lower computation cost.

Communication: In dynamic PDP scheme, the communication cost mainly comes from the block-tag uploading, remote data integrity query and response. We give our dynamic PDP scheme's communication overhead below. For $\hat{n}$ blocks, all the block-tag pairs length is $\hat{n}(|\mathcal{G}_1| + \log_2 q)$. In the phase Prove, the client sends the challenge $chal = (c, k_1, k_2)$ to CS, *i.e.*, the communication overhead is $\log_2 \hat{n} + 2\log_2 q$. In the response, CS responds 1 element in $\mathcal{G}_1$ and 1 element in $\mathcal{Z}_q^*$ to the client, *i.e.*, the communication overhead is $|\mathcal{G}_1| + \log_2 q$. On the other hand, Wang et al. [9] and Zhu et al. [10] proposed two different dynamic provable data possession scheme. Compared with these two schemes, our dynamic PDP scheme is more efficient in the communication cost. The communication comparison can be summarized in Table 2. In Table 2, $1|\mathcal{G}_1|$ denotes the bit length of one element in $\mathcal{G}_1$, $1|\mathcal{G}_2|$ denotes the bit length of one element in $\mathcal{G}_2$ and $1|\mathcal{Z}_q|$ denotes the bit length of one element in $\mathcal{Z}_q$. Our dynamic PDP scheme has lower communication cost.

Table 2. Comparison of communication cost (bits)

Protocols	Wang [9]	Zhu [10]	Ours
Tag	$(\hat{n}+1)\|\mathcal{G}_1\|$	$(\hat{n}+1)\log_2 \hat{n} + (s+\hat{n})\|\mathcal{G}_1\|$ $+\hat{n}(k+1)$	$\hat{n}\|\mathcal{G}_1\|$
Chal	$c(\log_2 \hat{n} + \log_2 q$	$c(\log_2 \hat{n} + \log_2 q)$	$\log_2 \hat{n} + 2\log_2 q$
Response	$\log_2 \hat{n}+(c+2)\|\mathcal{G}_1\|+O(c)$	$2\|\mathcal{G}_1\| + 1\|\mathcal{G}_2\| + s\log_2 q$	$1\|\mathcal{G}_1\| + 1\mathcal{Z}_q$

Private PDP and Convertibility: From the phase *VryTag*, we know CS can identify the invalid block-tag pairs. On the other hand, in the phase *Verify*, the client's secret key sk_c is needed. Thus, only the client can perform his own data's PDP. Our proposed dynamic PDP scheme is private PDP scheme. In the verification, the crucial element is $pk_s^{sk_c}$ which can only be computed by the client and the cloud server. When the client makes the crucial element $pk_s^{sk_c}$ public, every entity can perform the process of verification. Thus, our scheme can be converted into public PDP scheme.

5 Security Analysis

Theorem 1. *The proposed dynamic PDP scheme is existentially unforgeable in the random oracle model if the CDH problem on $\mathcal{G}_1$ is hard.*

The detailed proof process is omitted due to the page limit.

Theorem 2. *Suppose that $\hat{n}$ block-tag pairs are stored, $\bar{d}$ block-tag pairs are modified or are not correctly updated, and c block-tag pairs are challenged. Then, our proposed dynamic PDP scheme is $(\frac{\bar{d}}{\hat{n}}, 1-(\frac{\hat{n}-\bar{d}}{\hat{n}})^c)$-secure, i.e.,*

$$1 - (\frac{\hat{n}-\bar{d}}{\hat{n}})^c \le P_X \le 1 - (\frac{\hat{n}-c+1-\bar{d}}{\hat{n}-c+1})^c$$

where P_X denotes the probability of detecting the dishonest CS.

The detailed proof process is omitted due to the page limit.

6 Conclusion

Based on the dynamic binary tree, this paper proposes a private dynamic PDP scheme. From the comparison of communication cost and computation cost, our proposed private dynamic PDP scheme is efficient.

References

1. Ateniese, G., Burns, R., Curtmola, R., Herring, J., Kissner, L., Peterson, Z., Song, D.: Provable data possession at untrusted stores. In: Capitani, D., di Vimercati, S., Syverson, P. (eds.) CCS 2007, pp. 598–609. ACM Press, New York (2007)
2. Shacham, H., Waters, B.: Compact proofs of retrievability. In: Pieprzyk, J. (ed.) ASIACRYPT 2008. LNCS, vol. 5350, pp. 90–107. Springer, Heidelberg (2008)
3. Zheng, Q., Xu, S.: Fair and dynamic proofs of retrievability. In: CODASPY 2011, pp. 237–248. ACM Press, New York (2011)
4. Ateniese, G., Di Pietro, R., Mancini, L.V., Tsudik, G.: Scalable and efficient provable data possession. In: Liu, P., Molva, R. (eds.) SecureComm 2008, pp. 9:1–9:10. ACM Press, New York (2008)
5. Erway, C.C., Küpçü, A., Papamanthou, C., Tamassia, R.: Dynamic provable data possession. ACM Trans. Inf. Syst. Secur. **17**(4), 15 (2015)
6. Etemad, M., Küpçü, A.: Transparent, distributed, and replicated dynamic provable data possession. In: Jacobson, M., Locasto, M., Mohassel, P., Safavi-Naini, R. (eds.) ACNS 2013. LNCS, vol. 7954, pp. 1–18. Springer, Heidelberg (2013)
7. Cash, D., Küpçü, A., Wichs, D.: Dynamic proofs of retrievability via oblivious RAM. In: Johansson, T., Nguyen, P.Q. (eds.) EUROCRYPT 2013. LNCS, vol. 7881, pp. 279–295. Springer, Heidelberg (2013)
8. Yang, K., Jia, X.: An efficient and secure dynamic auditing protocol for data storage in cloud computing. IEEE Trans. Parallel Distrib. Syst. **24**(9), 1717–1726 (2013)
9. Wang, Q., Wang, C., Li, J., Ren, K., Lou, W.: Enabling public verifiability and data dynamics for storage security in cloud computing. In: Backes, M., Ning, P. (eds.) ESORICS 2009. LNCS, vol. 5789, pp. 355–370. Springer, Heidelberg (2009)
10. Zhu, Y., Ahn, G., Hu, H., Yau, S., An, H., Chen, S.: Dynamic audit services for outsourced storages in clouds. IEEE Trans. Serv. Comput. **99**, 1 (2011)
11. Zhang, Y., Marina, B.: Efficient dynamic provable possession of remote data via balanced update trees. In: ASIA CCS 2013, pp. 183–194. ACM Press, New York (2013)
12. Wang, J., Liu, S.: Dynamic provable data possession with batch-update verifiability. In: ICADE 2012, pp. 108–113. IEEE Press, New Jersey (2012)
13. Shi, E., Stefanov, E., Papamanthou, C.: Practical dynamic proofs of retrievability. In: ACM CCS, pp. 325–336 (2013)
14. Tate, S.R., Vishwanathan, R., Everhart, L.: Multi-user dynamic proofs of data possession using trusted hardware. In: 3rd ACM CODASPY, pp. 353–364. ACM Press, San Antonio (2013)
15. Boneh, D., Lynn, B., Shacham, H.: Short signatures from the weil pairing. In: Boyd, C. (ed.) ASIACRYPT 2001. LNCS, vol. 2248, pp. 514–532. Springer, Heidelberg (2001)
16. Boneh, D., Franklin, M.: Identity-based encryption from the weil pairing. In: Kilian, J. (ed.) CRYPTO 2001. LNCS, vol. 2139, pp. 213–229. Springer, Heidelberg (2001)