EasyChair Preprint

№ 375

Non-Repudiable Dynamic PDP with Stronger Security based on Stateful Rank Based Skip List

Minseok Lee, Doo Ho Choi and Taek-Young Youn

July 24, 2018

Non-Repudiable Dynamic PDP with Stronger Security based on Stateful Rank Based Skip List

Minseok Lee
UST
Daejeon, Korea
minseok717@etri.re.kr

Doo Ho Choi, Taek-Young Youn*
Electronics and Telecommunications Research Institute (ETRI)
Daejeon, Korea
{dhchoi,taekyoung}@etri.re.kr

Abstract

To guarantee the integrity of data stored in cloud servers, a number of techniques have been proposed. For security analysis, we generally assume honest but curious servers and honest clients. However, for dynamic data, we need to assume stronger adversaries since a malicious server or a malicious client can deny the update of stored data. Until now, some schemes have been proposed to give the security against the stronger adversaries. However, almost schemes are safe for adversarial clients or malicious servers, but not both. So, in this paper, we propose a PDP scheme supporting dynamic data and guarantees the integrity of stored data against malicious servers and clients. To design the scheme, we firstly design a new authenticated data structure so-called *Stateful Rank based Skip List* which reflects the latest version of stored data. Based on the data structure, we design a non-repudiable dynamic PDP which permits servers and clients to manage the version of stored data in an authenticated and non-repudiable way, so that our scheme prevents malicious servers and clients from cheating the latest version of the stored data.

1 Introduction

In cloud storage services, a client entrusts his/her data to a remote storage server and does not maintain the data in his local storage. Therefore the integrity of stored data relies on cloud server. If the storage server is attacked by a hacker or infected with a malware, stored data is compromised or deleted. So, data auditing technologies are important for clients to ensure that outsourced data are intact.

To check the integrity of data stored in remote cloud server, remote data auditing technologies, such as Provable Data Possession[1] and Proof of Retrievability[6], have been researched. Among them, we focus on PDPs in this work. The schemes permit clients to verify the integrity of the data stored in server's storage without downloading whole data from the server. To achieve the goal, in general, a client pre-processes his/her own data before uploading to cloud storage. Sometimes, in the procedure, the client makes metadata for the data to use it in the verification phase. To verify the integrity of the stored data, the client makes a set of challenges, and sends them to the server. Then, the server makes a proof as the response for given challenges using stored data. The client verifies the proof to check the integrity. If the given proof is verified to be valid, the client can assured that his/her file is intact.

Though a number of schemes have been proposed for verifiable integrity check, most of them are made for static data. They do not assume the scenario where stored data are dynamic which can be changed according to a client's request. However, stored data in cloud storage might be updated by data owner, such as modification, insertion and deletion. So, the research of PDP schemes for dynamic data is important and some PDP schemes have been proposed to support dynamic data. The first dynamic PDP scheme has been proposed in [2]. Thought the scheme can support *modification*, *deletion*, and *append*, it

G. Sutcliffe, A. Voronkov (eds.): easychair 1.0, 2008, volume 1, issue: 1, pp. 1-13

*Corresponding Author. This work was supported by Electronics and Telecommunications Research Institute (ETRI) grant funded by the Korean government [18ZH1200, Core Technology Research on Trust Data Connectome] and Institute for Information communications Technology Promotion(IITP) grant funded by the Korea government(MSIT) (No. 2016-0-00399, Study on secure key hiding technology for IoT devices [KeyHAS Project]).

cannot support *insert* operation. After the first scheme has been introduced, some dynamic PDP schemes have been proposed with fully dynamic operations [3, 4, 12].

To design dynamic PDP schemes, we need some additional data structure to manage the changes made in dynamic data. Almost dynamic PDP schemes have been designed based on skip list [9, 3, 4] or Merkle Hash Tree [7, 12] since the data structures are suitable for managing dynamic data. In the case of a skip list, the *rank based skip list* which is a variant of skip list is used in [3]. In the data structure, each internal node stores the homomorphic tag of data block, level of node, and the rank which is the number of reachable leaf nodes from the node, seachable variables and label of node. By using ranks, a client can fast search leaf node associated with data block which the client want to find. The complexity of the data update in the average case is $O(logN)$ When using a rank based skip list, which increases the efficiency of the dynamic PDP scheme. However, the data structure can not support the variable size of updated data. Then it incurs more overhead on other data blocks with $O(N)$ complexity in the worst case since block indices are used in this data structure. To overcome this problem, the PDP scheme in [4] uses flexible length based authenticated skip list which is called *FlexList*. The main advantage of the data structure supports the variable size of data block and data update. That is, in the data structure, each node stores the number of bytes which can be reached from the node. Therefore, the PDP scheme in [4] is faster than the PDP scheme in [3] in terms of data update with $O(u)$ complexity, where u is the size of update. In the case of Merkle hash tree, the PDP scheme in [12] uses *block update tree* which is a variant of Merkle hash tree. The data structure is always balanced regardless of the number and order of update operations. Also, the size of the data structure is independent of the stored data size because when the owner restores some data blocks in the cloud, all previous copies of the corresponding data block are deleted.

Almost existing dynamic PDP schemes assume honest clients and honest but curious servers in the view of security. The honest but curious servers honestly manage stored data but can corrupt less frequently accessed data for their benefit. This security model assuming honest but curious server is suitable for PDP scheme for static data. On the other hand, in a dynamic environment, malicious servers can store previous data if clients can not detect this type of misbehaviour although clients require some updates. As well as, there is possibility that a dishonest client legally accuses honest storage server as tampering stored data. So, we need to consider misbehaviour of dishonest clients in the security model of dynamic PDP. The security against a malicious client was firstly considered in [13]. In [13], the scheme detects misbehaviour of a malicious client by using incremental signature technique called hash-compress-and-sign. But after data update, the client does not have any way to prove the server's misbehaviour even if it restores the data for the previous without corrupting it. Because the client isn't given an undeniable information of a server, such as a signature, and the scheme is designed against not malicious servers but honest but curious servers. Most recently, dynamic PDP scheme in [11] has been announced, which assumes an adversarial client and an honest but curious server. This technology uses a data structure called Indext Logic Table(ILT) to allow the server and client to directly manage versions of stored data. However, ILT doesn't provide authentication information but only shows the logic numbers for each data block and all data. Therefore, the scheme has same problem as the scheme in [13] for a malicious server although a server and a client exchange undeniable information of each other. As mentioned above, we need to consider malicious servers which are more malicious than before in dynamic scenario. To design PDP scheme against malicious server, in [5], the proposed PDP scheme assumes the more malicious server than before. A way to prevent a server from passing verification even after storing an older version of stored data after the update is presented in [5]. When generating the tag to prevent this misbehaviour of the server, [5] inserts the version number of the data block which is the same as the logic number in [11]. However, [5] does not detect the misbehaviour of the malicious client because it assumes only the adversarial server. In order to guarantee strong security against both dishonest clients and malicious server, our paper designs dynamic PDP for a stronger server than before and a malicious client.

In this paper, we firstly design *non-repudiable PDP scheme* which guarantees strong security against malicious servers and dishonest clients. To design the PDP scheme, we propose a new authenticated dynamic data structure called *Stateful Rank based Skip List*. The data structure provides a version of stored data since the latest version of stored data is updated every time the data is updated. Then the client and server can generate an undeniable for both the version and authentication information of stored data. So, malicious servers and dishonest clients can not deny their misbehaviours by using PDP scheme, and it extends our scheme to *non-repudiable PDP scheme*.

2 Data Structure

We have already mentioned the problem of the use of Index Logic Table. In short, the Index Logic Table is vulnerable to the attack of a malicious server described in [5] since it is not authenticated. Let ILT denote Index Logic Table, T_{F_i} denote the tag of a data block F_i and σ_C denote the signature of client. For example, a client wants to modify F_i to F_i'. Then the client updates ILT and T_{F_i} to ILT' and T'_{F_i}, respectively before sending the updated data to the server. According to the method of [11], the client sends F', ILT', $T_{F_i'}$ and data list with the signature σ'_C for $FN \parallel U \parallel ILT'$. After receiving these, the sever only updates its own ILT to ILT' and other data stored in server is not updated. If the server has well managed the data which is not updated, the proof containing T_{F_i} is passed from the client. The main reason why this problem happens is that ILT is not authenticated. That is, a client doesn't know whether the server exactly stores updated data or not. Thus although a server doesn't perform update of some file blocks, the server can make a cheating proof which is successfully passed from verification of the client.

In this section, we propose two data structures, "version-sequence" and *stateful rank-based skip list*. The former enables a client to handle the version of outsourced data, the latter is an authenticated data structure to solve the above problem.

2.1 Version-sequence

In this section, we explain a data structure to enable a client to handle the version of data. This means that a client can distinguish which data is the latest from the proposed data structure. A version sequence is the sequence of logic numbers defined in [11]. From now on, let $vseq$ denote a version-sequence. In $vseq = (vseq[1], \ldots, veseq[n], VN)$, $vseq[i]$ denotes the logic number of i^{th} data block and VN denotes the version number of stored data. The version number increases whenever data update happens by 1.

Version-sequence provides 3 operations, modification, insertion and deletion. Divide data F into n blocks and suppose that the length of each data block is same. Then the initial version-sequence is $(1, \ldots, n, n+1)$. When a stored data is updated, the corresponding version-sequence is updated too. Let algorithms concerned with modification, insertion and deletion denoted by `vseqM`, `vseqI` and `vseqD`, respectively. 1) a client wants to modify the i^{th} data block F_i to F_i'. Then $vseq[i]$ is updated to VN of $vseq$, and VN is increased to $VN+1$. 2) In the case of insertion operation, a client would insert a data block F' after i^{th} data block. Then from the index $i+1$ to the index n, each $vseq[k]$ for $k = i+1, \ldots, n$ is increased by 1 but each the value of $vseq[k]$ maintains. After that, $vseq[i+1]$ is updated to VN and VN is increased to $VN+1$. 3) when a client deletes the i^{th} data block F_i, from the index $i+1$ to the index n, each $vseq[k]$ for $k = i+1, \ldots, n$ is decreased by 1 but each the value of $vseq[k]$ maintains. After that, VN is updated to $VN+1$.

2.2 Stateful rank-based skip list

In this section, we make a new authenticated skip list originated by the rank-based skip list in [3] which

Table 1: Notation

Notation	Meaning
$l(v)$	the level of the node v
$r(v)$	the rank of the node v
$f(v)$	the label of v
$t(v)$	the data block tag stored in the node v
$LN(v)$	the logic number of a data block stored in the node v

stores logic numbers of all data blocks. Briefly, the rank of a node is defined by the number of reachable leaf nodes from the node. From [9], for each internal node in a skip list, there are two points, right pointer and down pointer, denoted by *rgt* and *dwn*. For a node v, $rgt(v)$ and $dwn(v)$ denote the nodes right to v and down to v, respectively. We uses the example of a rank-based skip list described in **Figure1** of [3]. In **Figure1** of [3], $rgt(w_3)$ and $dwn(w_3)$ are v_1 and v_5, respectively. Before looking details, we define some notations as like Table.1

Let a data F be divided into n data blocks, $F_1,\ldots,F_n$, and let LN_i be the logic number of F_i for $i = 1,\ldots,n$. The i^{th} data block F_i will be stored in i^{th} leaf node of a skip list. Each leaf node stores data block tags $t_1,\ldots,t_n$(the making tags will be referred later), corresponding level, rank, label and the logic number of the data block. Except for leaf nodes, all internal nodes store corresponding level, rank, and label. Each rank of nodes is used for search operation.

We can find the leaf node to search by using ranks. For example, in **Figure1** of [3], the rank of node w_3 is 4 because from the node, 4 nodes, v_5, v_4, v_3, v_1 can be reached. Let v be the current node, and let $high(v)$ and $low(v)$ be right most reachable node and left most reachable node of v, respectively assume that a client wants to find i^{th} node. For example, in **Figure1** of [3], $high(w_3), low(w_3)$ are 6, 3 respectively. If i is in $[low(v), high(v)]$, then the current v is moved to $rgt(v)$, otherwise, the node v is moved to $dwn(v)$. The relation between $r(v)$, $low(v)$ and $high(v)$ is referred in [3]

For the proposed skip list to be authenticated is needed to make an authentication information which is like hash accumulation in [7]. Let $f(v)$ be the label of a node v defined by the value of hash accumulation of labels which belongs to neighborhood nodes of v. The method of hashing is same as referred one in [3] but the logic number stored in a node v are added to [3] as a parameter. As referred in [3], we can define follows:

Definition 1. *Let h be a collision-resistant hash function. For input $x_1,\ldots,x_n$, $h(x_1,\ldots,x_n)$ is defined as follow:*

$$h(x_1,\ldots,x_n) := h(h(x_1) \parallel \cdots \parallel h(x_n))$$

Definition 2. *Let h be a hash function defined in above definition.The label $f(v)$ of a node v in the proposed skip list containing the logic number stored in v is defined as follows:*

CASE1. $v = null$

$$f(v) \equiv 0$$

CASE2. $l(v) > 0$

$$f(v) = h(l(v), r(v), 0, f(dwn(v)), f(rgt(v)))$$

CASE3. $l(v) = 0$

$$f(v) = h(l(v), r(v), LN(v), t(v), f(rgt(v))$$

By the above definition, a client can construct a proposed skip list and after that, the client stores the label $f_C(s)$ of the start node in his/her local storage and sends the skip list to the server.

2.2.1 Proof

In this section, we make a proof to authenticate data stored in cloud storage. To authenticate the i^{th} data block F_i, the data owner sends the corresponding query $Q(i)$ to the server. After receiving $Q(i)$, the server uses the algorithm in 2 to make proof path P and sends the proof path with $t(v_i)$ where v_i denotes the i^{th} leaf node. The client computes the label of the start node from P and $t(v_i)$ and check whether computed label is equal to $f_C(s)$ or not. If it is not equal, then the client considers that the i^{th} data is corrupted or lost.

The method to generate a proof path is equal to [3]. But as mentioned above, the logic number is added to the algorithm proposed in [3] as a parameter. Assume that the server receives the query $Q(i)$ sent by the client. Let $v_1, \ldots, v_s$ denote the nodes that have gone from the start node to the i^{th} leaf node in reverse order. Then v_1 is the i^{th} node arrived, v_2 is the start node, and let $search(i) = (v_1, \ldots, v_s)$ be the search path. By [3], a proof path $P(i)$ consists of $A(v_1), \ldots, A(v_s)$ and each $A(v_j)$ consists of $l(v_j), LN(v_j), q(v_j), d(v_j), g(v_j)$ which the logic number is added to existing $A(v_j)$ member described in [3].

The element $d(v_j)$ for each $A(v_j)$ in the proof path P indicates whether the element v_{j-1} of the search path is the right node or the bottom node of v. $g(v_j)$ and $q(v_j)$ represent the labels and ranks of $rgt(v_j)$ or $dwn(v_j)$ nodes, respectively, that are not included in the search path. Finally, $LN(v_j)$ is the logic number stored in the node. Since $q(v_j), d(v_j)$ and $g(v_j)$ in this paper are the same as those presented in [3], details regarding these can be read in [3]. However, $LN(v_j)$ is as follows:

$$LN(v_j) = \begin{cases} LN(v_j) & \text{if } l(v_j) = 0 \\ 0 & \text{if } l(v_j) \neq 0 \end{cases}$$

Table.3 is the example of the proof path for the 2^{nd} data block(or leaf node) in the **Figure1** of [3] and the Table.2 shows the algorithm of the generating a proofpath.

Table 2: The algorithm `ProofPath`$(Q(i))$

1: **Input** $Q(i)$
2: Let $search(i) = (v_1, \ldots, v_s)$ for i^{th} data block
3: **return** $P(i) = (A(v_1), \ldots, A(v_s))$

Table 3: Proof path $P(2)$ for 2^{nd} data blcok, where $search(6) = (v_6, w_4, w_5, w_6, w_7)$

node v	v_6	w_4	w_5	w_6	w_7
$l(v)$	0	2	3	3	4
$LN(v)$	$LN(v_6)$	0	0	0	0
$d(v)$	rgt	dwn	dwn	rgt	dwn
$q(v)$	0	4	5	1	
$g(v)$	0	$f(w_3)$	$f(v_7)$	$f(v_8)$	$f(v_9)$

2.2.2 Verification

When the client receives $(P(i), t(v_i))$ from the server, the client computes the label $f_S(s)$ of the start node through the proof sent by the server using the $\texttt{SKVerify}(i, P(i), t(v_i), f_C(s))$. The operation of $\texttt{SKVerify}(i, P(i), t(v_i), f_C(s))$ is similar to the verification algorithm described in [3]. However, the variables related to the logic number are added to the algorithm of [3]. Hence, our verification algorithm is the same as the verification algorithm of [3] but only adds the variables of logic number.

2.2.3 The Method of update of Sateful Rank-Based Skip List

In this section, we propose algorithms that allow the skip list proposed in this paper to reflect data updates. There are three types of update: modification, insertion, deletion. Each update operation is denoted by M, I and D. To briefly describe the update process, the client sends a query (op, i, F_i', t_i') to the server to update a data, where op is one of M, I and D, and i is an index of a data block to be updated. The meaning of each query is as follows:

- (M, i, F_i', t_i'): Modify i^{th} data block F_i and the corresponding tag t_i to F_i' and t_i', respectively.
- $(I, i, F_{i+1}', t_{i+1}')$: Insert F_{i+1}' after i^{th} data block and store the tag t_{i+1}' for F_{i+1}'.
- $(D, i, NULL, NULL)$: Delete the i^{th} data block.

When the server receives each query, it sends proof path $P(i)$ and t_i for the current data without immediately executing the update. The server sends $(P(i), t_i)$ for modification and insertion, and $(P(i-1), t_{i-1})$ for deletion. If the verification for each proof to update data is successfully passed, then the server proceeds to update the data and the stateful rank-based skip list through the update algorithms which are described in Table.4, 5 and 6

The client verifies each server through SKVerify as shown in the previous section. If the verification succeeds, the update is considered successful, otherwise the server did not perform the update as requested by the client.

Table 4: The algorithm $\texttt{SKUpdateM}(i, F_i', t_i')$ for modification

1: **Input** i, F_i', t_i'
2: Update labels, logic numbers, levels, ranks of nodes in $search(i)$
3: **return** $(P'(i), t_i')$

Table 5: The algorithm $\texttt{SKUpdateI}(i, F'_{i+1}, t'_{i+1})$ for insertion

1: **Input** i, F'_{i+1}, t'_{i+1}
2: Insert $F'_{i+1} after i^{th}$ data block and store the coressponding tag t'_{i+1}
3: Update labels, logic numbers, levels, ranks of nodes in $search(i+1)$
4: **return** $(P'(i+1), t'_{i+1})$

Table 6: The algorithm $\texttt{SKUpdateD}(i, NULL, NULL)$ for deletion

1: **Input** i
2: Delete i^{th} data block and the corresponding tag
3: Update labels, logic numbers, levels, ranks of nodes in $search(i-1)$
4: **return** $(P'(i-1), t'_{i-1})$

3 Proposed Scheme

In this section, we describe dynamic PDP protocol that enables a client to handle the versions of stored data and supports non-reputation. The proposed scheme consists of four phases: SetUp, Upload, Proof-Verification, and Update. Briefly, the SetUp phase is where the client generates keys for tag generation and signature schemes. The step of Upload is a step of generating a ta, splitting a data to store, generating a version-sequence, a stateful rank-based skip list, and a signature of the client, and transmitting the signature to the server. The proof-verification phase is the verification phase of the proof generated by the server for the data blocks challenged by the client. Finally, the Update step is the step where the server executes the data update requested by the client and the client verifies the update execution.

3.1 SetUp and Upload

Divide data F into n blocks. Let each data block be denoted by $F_1, \ldots, F_n$, where each F_i is in $\mathbb{Z}_p$ for all $i = 1, \ldots, n$. To enable the client to upload data to store to cloud storage, and the client and server to verify data exchanged by each other, the following algorithms are used

- $\texttt{KeyGen}(1^\lambda)$: 1) the client generates signing key pair (ssk_C, spk_C). And the client chooses randomly an element α in $\mathbb{Z}_p$ and computes $v = g^\alpha$. Then (ssk_C, α) is securely managed as the private key sk_C of the client and (spk_C, v) is publicly released as the public key pk_C of the client.

  2) the server similarly generates signing key (ssk_S, spk_S). Then ssk_S is securely stored as the private key of the server, spk_S is public as the public key of the server.

- $\texttt{TagGen}(sk_C, F, H, h)$: This algorithm is performed by the data owner. Let G and G' be multiplicative cyclic groups with order p which is a sufficiently large prime, and let $e : G \times G' \longrightarrow G'$ be a bilinear map. Let $H : \{0,1\}^* \longrightarrow G$ be a map-to-hash function, and h be a collision-resistant hash function defined by previous section. For each data block $F_i \in \mathbb{Z}_p$, for $i = 1, \ldots, n$, generate the tag t_i as follow:

  $$t_i = (H(F_i \parallel LN_i) \cdot u^{F_i})^\alpha$$

  Also, the client constructs the version-sequence $vseq_C$ and the skip list SKL_C for the data F. Let $f_C(s)$ be the label of the start node of SKL_C. The client makes the signature for $f_C(s) \parallel vseq_C$ as follow:

  $$sig(ssk_C, f_C(s) \parallel vseq_C) = [h(f_C(s) \parallel vseq_C)]^{ssk_C}$$

Let sig_C denote $sig(ssk_C, f_C(s) \parallel vseq_C)$. After generating sig_C, the client sends $\{(F_i,t_i)\}_{i=1}^n$, SKL_C, $vseq_C$, sig_C to the cloud server.

- `UploadCheck_server`$(pk_C, ssk_C, SKL_C, vseq_C, sig_C)$: 1) Let $SKL_S, vseq_S$ be state of $SKL_C, vseq_C$ respectively, after server receives $SKL_C, vseq_C$ from the client. The server computes the label $f_S(s)$ of SKL_S and verifies sig_C from $f_S(s) \parallel vseq_S$ and spk_C. 2) And verifies the relation between tags and corresponding data blocks from following formula:

$$e(t_i, g) =? e(H(F_i \parallel LN_i) \cdot u^{F_i}, v)$$

If one of 1) and 2) is not passed, the server returns reject. Otherwise, the server generates the signature $sig_S = sig(ssk_S, h(f_S(s) \parallel vseq_S))$ for $h(f_S(s) \parallel vseq_S)$. Then the server makes the receipt $sig_S(sig_C) = sig(ssk_S, h(f_C(s) \parallel vseq_C))$ of sig_C. The server sends sig_S and $sig_S(sig_C)$ to the client.

- `UploadCheck_client`$(spk_S, ssk_C, sig_S, sig_S(sig_C))$: 1) the client verifies $sig_S(sig_C)$ from sig_C and spk_S. 2) And also the client verifies sig_S from $f_C(s)$ and $vseq_C$. If one of 1) and 2) fails, then the client returns reject to the server. Otherwise, the client generates the receipt $sig_C(sig_S) = sig(ssk_C, sig_S)$ of sig_S and sends the receipt to the server.

The server verifies the receipt of the client by using `UploadCheck_server`. If the verification succeeds, the client stores $f_C(s)$, $vseq_C$, sig_C and $sig_S(sig_C)$ in his/her local storage, and the server stores $\{(F_i,t_i)\}_{i=1}^n, SKL_S, vseq_S, sig_S$ and $sig_C(sig_S)$.

3.2 Proof-Verification

Briefly, the client randomly chooses elements $\gamma_{i_1}, \ldots, \gamma_{i_k}$ in $\mathbb{Z}_p$ for data block indices $\gamma_{i_1}, \ldots, \gamma_{i_k}$. Then the client sends $i_j, \gamma_{i_j}{}_{j=1}^k$ to the server as the challenge C. As soon as receiving the challenge, the server generates the proof P and sends this to the client. Then the client verifies the proof. Before showing details, let the tag set $\{t_1, \ldots, t_n\}$ be denoted by T. Algorithms are follows:

- `GenProof`$(C, T, F, SKL_S, vseq_S)$: This algorithm is performed by the server. The server makes the proof paths $P(i_1), \ldots, P(i_k)$ for $i_1, \ldots, i_k$ and generates $\{P(i_j), t_{i_j}\}_{j=1}^k$. Compute $\mu = \sum_{j=1}^k \gamma_{i_j} \cdot F_{i_j}$, $t = \Pi_{j=1}^k t_{i_j}^{\gamma_{i_j}}$ for data blocks $F_{i_1}, \ldots, F_{i_k}$ and makes $h(vseq_S)$. Then the server sends the proof $P = (\{P(i_j), t_{i_j}\}_{j=1}^k, (\mu, t), h(vseq_S))$ to the client.

- `Verify`$(pk_C, P, f_C(s), vseq_C)$: 1) the client check the coincidence between $h(vseq_C)$ and $h(vseq_S)$. 2) For each $(P(i_j), t_{i_j}), j = 1, \ldots, k$, the client uses `SKVerify` to authenticate proof pahts. 3) Also, the client verifies the integrity of challenged data blocks by using following formula:

$$e(\mu, \sigma) =? e(\Pi_{j=1}^k H(F_{i_j} \parallel LN_{i_j}) \cdot u^{\mu}, v)$$

If one of 1), 2) and 3) is not passed, then the client returns reject. Otherwise, the verification succeeds.

3.3 Update

In this section, we would the update phase. Briefly speaking, the client sends an update request (op, i, F') to server. op denotes operations of update, and let M, I and D denote modification, insertion and deletion, respectively. Also i is the index of data block to update. Specially, in the case of insertion, i means that the data block F' would be inserted after i^{th} block. Also F' is $NULL$ when a deletion happens. After the server receives client's request, the server performs the data update along with the request.

- **Modification**

Step1. the client wants to modify the i^{th} data block F_i to F_i'. The client computes the tag t_i' for F_i' and sends the update request (M, i, F_i', t_i') to the server. And by using the algorithm `vseqM`, the client updates his/her own version sequence $vseq_C$ to $vseq_C'$.
Step2. When the server receives client's request, firstly the server makes the proof path $P(i)$ for i^{th} data block by using the algorithm `ProofPath`, then the server generates $(P(i), t(v_i))$. According to the update request, the server modifies the data block F_i to F_i' and replaces the corresponding tag t_i with t_i'. And by using `SKUpdateM` and `vseqM`, update server's skip list SKL_S and version-sequence $vseq_S$ to SKL_S' and $vseq_S'$, respectively. After performing the required update, the server makes the proof path $P'(i)$ and generates $(P'(i), t'(v_i))$. Then the server sends $((P(i), t(v_i)), (P'(i), t'(v_i))$ to the client.
Step3. the client uses the algorithm `SKVerify` to authenticate $(P(i), t(v_i))$. If the verification is passed, then the client predicts or computes the updated label $f_C'(s)$ of the start node from $(P(i), t(v_i))$. The method to update the authentication information is described in [8]. After computing the updated label of the start node, the client verifies $(P'(i), t'(v_i))$ from $f_C'(s)$ and `SKVerify`. If the verification succeeds, the client makes $sig_C' = sig(ssk_C, h(f_C'(s) \parallel vseq_C'))$ and sends this to the server.
Step4. the server computes $h(f_C'(s) vseq_S')$ from SKL_S' and $vseq_S'$. Then the server verifies the sig_C'. If sig_C' is successfully verified, then the server generates the receipt as follow:

$$sig_S'(sig_C') = sig(ssk_S, sig_C')$$

And makes $sig_S' = sig(ssk_S, h(f_S'(s) \parallel vseq_S'))$. Then the server sends sig_S' and the receipt.
Step5. the client verifies sig_S' from $h(f_C'(s) \parallel vseq_C')$ and spk_S. And also the client checks the receipt from sig_C' and spk_S. If the two signatures are successfully verified, then the client generates $sig_C'(sig_S') = sig(ssk_C, sig_S')$ as the receipt and sends the receipt to the server.
Step6. the server verifies the receipt of the client from sig_S' and spk_C.

All steps are successfully completed, the client updates $f_C(s)$ to $f_C'(s)$. Then the client stores $f_C'(s), vseq_C', sig_C'$ and $sig_S'(sig_C')$ in his/her own local storage. The server stores updated data, $SKL_S', vseq_S', sig_S'$ and $sig_C'(sig_S')$.

- **Insertion**

Step1. the client wants to insert a data block F' after the i^{th} data block. The client computes the tag t_{i+1}' for F' and sends the update request (I, i, F', t_{i+1}') to the server. After that, the client updates the existing version-sequence $vseq_C$ to $vseq_C'$ by using `vseqI`.
Step2. After receiving the update request, the server generates $(P(i), t(v_i))$ by making the proof path $P(i)$ from the `ProofPath`. According to the update request, the server inserts F' after i^{th} data block and stores the corresponding tag t_{i+1}'. At the same time, the server replaces the existing skip list SKL_S and version-sequence $vseq_C$ with SKL_S' and $vseq_S'$ by using the algorithms `SKUpate` and `vseqI`. After performing update, the server makes proof path $P'(i+1)$ for $i+1^{th}$ data block to generate $(P'(i+1), t'(v_{i+1}))$. Then the server forward $((P(i), t(v_i)), (P'(i+1), t'(v_{i+1}))$ to the client.
Step3. the client verifies $(P(i), t(v_i))$ from `SKVerify`. If the verification is passed, then the client computes the updated label $f_C'(s)$ and verifies $(P'(i+1), t'(v_{i+1}))$ from the $f_C'(s)$ and `SKVerify`. When the verification succeeds, the client generates $sig_C' = sig(ssk_C, h(f_C'(s) \parallel vseq_C'))$ to send to the server.

Step4. the server verifies sig'_C by computing $h(f'_S(s) \parallel vseq'_S)$. If the verification succeeds, the server generates $sig'_S(sig'_C) = sigh(ssk_S, sig'_C)$,and makes the receipt $sig'_S(sig'_C) = sig(ssk_S, sig'_C)$. Then the server sends thses to the client.
Step5. the client verifies sig'_S by using $h(f'_C(s) \parallel vseq'_C)$ and skp_S. And similarly, the client verifies the receipt. If sig'_S and $sig'_S(sig'_C)$ are valid, then the client makes the receipt $sig'_C(sig'_S) = sig(ssk_C, sig'_S)$ and sends it to the server.
Step6. the server verifies the receipt.

All steps are successfully passed, the client and the server stores their own data in each storage.

- **Deletion**
Step1. the client sends the update request $(D, i, NULL, NULL)$ to delete the i^{th} data block F_i, and update the existing version-sequence $vseq_C$ to $vseq'_C$ by using `vseqD`.
Step2. After receiving the update request, the server generates $(P(i-1), t(v_{i-1}))$ by making the proof path $P(i-1)$ from the `ProofPath`. According to the update request, the server deletes F_i and the corresponding tag t_i. At the same time, the server replaces the existing skip list SKL_S and version-sequence $vseq_C$ with SKL'_S and $vseq'_S$ by using the algorithms `SKUpate` and `vseqI`. After performing update, the server makes proof path $P'(i-1)$ for $i-1^{th}$ data block to generate $(P'(i-1), t'(v_{i-1}))$. Then the server forward $((P(i-1), t(v_{i-1})), (P'(i-1), t'(v_{i-1}))$ to the client.
Step3. the client verifies $(P(i-1), t(v_{i-1}))$ from `SKVerify`. If the verification is passed, then the client computes the updated label $f'_C(s)$ and verifies $(P'(i-1), t'(v_{i-1}))$ from the $f'_C(s)$ and `SKVerify`. When the verification succeeds, the client generates $sig'_C = sig(ssk_C, h(f'_C(s) \parallel vseq'_C))$ to send to the server.
Step4. the server computes $h(f'_C(s) vseq'_S)$ from SKL'_S and $vseq'_S$. Then the server verifies the sig'_C. If sig'_C is successfully verified, then the server generates the receipt as follow:

$$sig'_S(sig'_C) = sig(ssk_S, sig'_C)$$

And makes $sig'_S = sig(ssk_S, h(f'_S(s) \parallel vseq'_S))$. Then the server sends sig'_S and the receipt.
Step5. the client verifies sig'_S from $h(f'_C(s) \parallel vseq'_C)$ and spk_S. And also the client checks the receipt from sig'_C and spk_S. If the two signatures are successfully verified, then the client generates $sig'_C(sig'_S) = sig(ssk_C, sig'_S)$ as the receipt and sends the receipt to the server.
Step6. the server verifies the receipt of the client from sig'_S and spk_C.

The subsequent steps are the same as described above.

4 Security Analysis

As mentioned above, we assume that both client and cloud server are dishonest. For example, suppose that server doesn't have corrupted stored data of data owner , and assume that the proof of server for challenged data blocks from a client is passed. But to either earn money or undermine the external reputation of a cloud service provider, client maliciously claims that the proof of server is not correct and accuse the service provider as tampering his/her data to a settlement. Since two parties are dishonest, the 3rd party, such as judge, cannot convince which evidences(e.g. metadata) are truthful.

In this paper, when there is disputes between client and server, we propose the method of settling disputes by exchanging each other's signature. Briefly mentioning, the use of a version-sequence which

is a monotonic data structure and the exchange of each other's signatures can resolve disputes between clients and servers.

The security of the proposed scheme is proved from the correctness and soundness defined in [10]. Briefly speaking, the security is based on the secure BLS signature, computational Diffie-Hellman problem in bilinear map.

Definition 3. *The computational Diffie-Hellman problem is that for given g, g^x, g^y in G for unknown x, y in $\mathbb{Z}_p$, to compute g^{xy}*

Theorem 1. *Suppose that the signature scheme used to make data block tags is existentially unforgeable and the computational Diffie-Hellman problem is infeasible in bilinear map. Then no adversary against the soundness of our public-verification scheme could cause a verifier to accept in the proposed auditing protocol instance with non-negligible probability, except by responding with correctly generated proof.*

The proof of the soundness of the protocol can be guaranteed if the computational Diffie-Hellman problem is unfeasable. It is proved from the sequences of the challenge games described in [10]. Briefly mentioning, in random oracle model if an adversary can make a cheating proof then the adversary can solve the CDH problem with unnegligible probability because this means that the adversary can make forgeries of the data block tags. The rigorous proof can be provided latter, and the reader can find the proof in [10]

The next theorems are concerned with the method to resolve disputes between the client and the server. Before showing the theorem to resolve disputes, we first show that a version-sequence is a monotonic data structure which is defined in [11] the following theorem is regarding with it.

Theorem 2. *A version-sequence is a monotonic data structure*

Proof. Let $vseq_1$ and $vseq_2$ be version-sequences originated from the same data F, respectively, and let VN_1 and VN_2 be the version numbers of $vseq_1$ and $vseq_2$, respectively. We already know that each version number is increased by 1 whenever the data F is updated. If $VN_1 > VN_2$, then $vseq_1$ is a version-sequence that reflects the most recently updated data F from $vseq_2$. If $VN_1 < VN_2$, then $vseq_2$ is a version-sequence that reflects the most recently updated data F from $vseq_1$. If $VN_1 = VN_2$ and $h(vseq_1) = h(vseq_2)$, then $vseq_1$ is absolutely same as $vseq_2$. Hence, a version-sequence is monotonic □

Theorem 3. *If the signature scheme which is exchanged between a client and a server is existentially unforgeable and a version-sequence is monotonic, then the proposed PDP protocol can settle the following disputes:*

CASE1. The client have required data update to the server but the client didn't update his/her own the label of the start node of SKL_C. Then the client blames the server as framing up the stored data.

CASE2. The server has saved the data back to the previous version since the update was verified. At this time, the server claims to have updated the data.

CASE3. The server has corrupted data.

We skip the rigorous proof of the above theorem. Briefly sketching the proof, Since the signature scheme is existentially unforgeable, both client and server can not make forge their own receipt. The third party, such as a judge, receives the receipts $sig_S(sig_C)$ and $sig_C(sig_S)$ of the client and the server from the two party. And the third party requires the client to generate sig'_C from $f_C(s)$ and $vseq_C$ stored in his/her local storage, at the same time requires the server to make sig'_S from SKL_S and $vseq_S$. If their data

is corrupted or forged, then the verification for their receipt by the third party can not be passed. Also, although each receipt is valid, the third party can distinguish whose data is latest because the version sequence is monotonic. Hence, the proposed protocol can resolve disputes. The rigorous proof can be provided latter.

5 Conclusion

We design the *non-repudiable PDP scheme* against malicious servers and dishonest clients for the first time by using new data structure. Our scheme allows a clients and a server to generate undeniable information for the latest version and authentication information of stored data. Hence, each malicious party can not deny misbehaviours of each other under our PDP scheme. In the view of security, we also prove that our scheme guarantees stronger security than before when compared to existing PDP schemes in this paper. Hence, we expect that our scheme can improve the security of cloud storage service.

References

[1] Giuseppe Ateniese et al. "Provable Data Possession at Untrusted Stores". In: *Proceedings of the 14th ACM Conference on Computer and Communications Security*. CCS '07. Alexandria, Virginia, USA: ACM, 2007, pp. 598–609. ISBN: 978-1-59593-703-2. DOI: 10.1145/1315245.1315318. URL: http://doi.acm.org/10.1145/1315245.1315318.

[2] Giuseppe Ateniese et al. "Scalable and Efficient Provable Data Possession". In: *Proceedings of the 4th International Conference on Security and Privacy in Communication Netowrks*. SecureComm '08. Istanbul, Turkey: ACM, 2008, 9:1–9:10. ISBN: 978-1-60558-241-2. DOI: 10.1145/1460877.1460889. URL: http://doi.acm.org/10.1145/1460877.1460889.

[3] C. Chris Erway et al. "Dynamic Provable Data Possession". In: *ACM Trans. Inf. Syst. Secur.* 17.4 (Apr. 2015), 15:1–15:29. ISSN: 1094-9224. DOI: 10.1145/2699909. URL: http://doi.acm.org/10.1145/2699909.

[4] Ertem Esiner et al. "Flexlist: optimized skip list for secure cloud storage". In: *Technical Report, Koç University* (2013).

[5] Clémentine Gritti et al. "Dynamic Provable Data Possession Protocols with Public Verifiability and Data Privacy". In: *Information Security Practice and Experience*. Ed. by Joseph K. Liu and Pierangela Samarati. Cham: Springer International Publishing, 2017, pp. 485–505. ISBN: 978-3-319-72359-4.

[6] Ari Juels and Burton S. Kaliski Jr. "Pors: Proofs of Retrievability for Large Files". In: *Proceedings of the 14th ACM Conference on Computer and Communications Security*. CCS '07. Alexandria, Virginia, USA: ACM, 2007, pp. 584–597. ISBN: 978-1-59593-703-2. DOI: 10.1145/1315245.1315317. URL: http://doi.acm.org/10.1145/1315245.1315317.

[7] Merkle and Ralph C. "Protocols for public key cryptosystems". In: *IEEE* (1980), pp. 122–122.

[8] Charalampos Papamanthou and Roberto Tamassia. "Time and Space Efficient Algorithms for Two-Party Authenticated Data Structures". In: *Information and Communications Security*. Ed. by Sihan Qing, Hideki Imai, and Guilin Wang. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 1–15. ISBN: 978-3-540-77048-0.

[9] William Pugh. "Skip Lists: A Probabilistic Alternative to Balanced Trees". In: *Commun. ACM* 33.6 (June 1990), pp. 668–676. ISSN: 0001-0782. DOI: 10.1145/78973.78977. URL: http://doi.acm.org/10.1145/78973.78977.

[10] Hovav Shacham and Brent Waters. “Compact Proofs of Retrievability”. In: *Advances in Cryptology - ASIACRYPT 2008*. Ed. by Josef Pieprzyk. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 90–107. ISBN: 978-3-540-89255-7.

[11] Feng Wang et al. “Identity-based non-repudiable dynamic provable data possession in cloud storage”. In: *Computers Electrical Engineering* (2017). ISSN: 0045-7906. DOI: https://doi.org/10.1016/j.compeleceng.2017.09.025. URL: http://www.sciencedirect.com/science/article/pii/S0045790617331038.

[12] Yihua Zhang and Marina Blanton. “Efficient Dynamic Provable Possession of Remote Data via Update Trees”. In: *Trans. Storage* 12.2 (Feb. 2016), 9:1–9:45. ISSN: 1553-3077. DOI: 10.1145/2747877. URL: http://doi.acm.org/10.1145/2747877.

[13] Qingji Zheng and Shouhuai Xu. “Fair and Dynamic Proofs of Retrievability”. In: *Proceedings of the First ACM Conference on Data and Application Security and Privacy*. CODASPY ’11. San Antonio, TX, USA: ACM, 2011, pp. 237–248. ISBN: 978-1-4503-0466-5. DOI: 10.1145/1943513.1943546. URL: http://doi.acm.org/10.1145/1943513.1943546.