International Journal of Grid Distribution Computing
Vol. 8, No.4, (2015), pp. 179-198
http://dx.doi.org/10.14257/ijgdc.2015.8.4.18

A Survey of Remote Data Integrity Checking: Techniques and Verification Structures

Yu Chen[1], Feng Wang[2], Liehuang Zhu[1] and Zijian Zhang[1]

[1]*Beijing Engineering Research Center of Massive Language Information Processing and Cloud Computing Application, School of Computer Science and Technology, Beijing Institute of Technology, Beijing, 100081, China*
[2]*Naval Academy of Armament, Beijing, 100161, China*
chenyulonger@bit.edu.cn, lionkingwf@hotmail.com, liehuangz@bit.edu.cn, zhangzijian@bit.edu.cn

Abstract

Cloud storage offers clients great convenience to relieve them from heavy burden of storage and management, so an increasing number of clients choose to outsource their data to remote cloud providers. However, for clients, this entails a sacrifice of actual control of these files. Remote servers may suffer from disk failure for uncertain reasons or even delete rarely accessed data to sell these storages to other clients. Therefore, there's a great necessity for clients to make sure that their data are well stored in remote servers. Numbers of remote integrity checking (RIC) schemes are proposed to solve issues above, including following up work Provable Data Possession (PDP) and Proofs of Retrievability (PoR), which can be applied in cloud auditing. This paper presents state-of-the-art RIC schemes and makes a classification from the perspective of whether they support dynamic verifications, i.e., whether they can still be used to make verifications after clients modify, insert or delete files. In static verification schemes, we delve into the mechanisms and techniques used for integrity checking. For dynamic ones, we discuss the authentication structures that support dynamic operations. We also present several remarks to guide readers to a wide vision of data checking as conclusions and future work.

Keywords: *cloud storage, remote integrity checking, provable data possession, proofs of retrievability, dynamic updates*

1. Introduction

With the development of computation enhancement of IT infrastructure in recent years, many clients choose to outsource their data to remote cloud servers because of their limited resource. They believe that remote servers can offer many favorable services: 1) remove burden of expensive storage and management from clients, 2) provide higher availability and scalability in comparison with clients and 3) enable clients to access their data in different places at any time. However, remote storages are out control of physical possession of clients. Remote servers may fail to store clients' data correctly for accidental reasons (adversary attacks) or intentional ones (e.g. they may delete rarely accessed data to save storage to sell these storages to other clients) [1]. Dishonest servers may convince clients into believing that their data are well stored in the server to hide their misbehavior for the sake of reputation. Once such incidents happen, it is always too late for clients to realize the loss of data when they retrieve them. Thus, it's necessary for clients to know whether their data are well stored in the remote servers via some mechanisms rather than routine retrieval.

ISSN: 2005-4262 IJGDC

A naive solution is to download outsourced files and check their intactness by cryptography measurement regularly [2]. Obviously, this is not practical at all, especially for large files in the cloud environment. To solve this problem, some literature introduced the concept of Remote Integrity Checking (RIC) [3]. Later provable data possession (PDP) [1] and proofs of retrievability (PoR) [4] are proposed to enable clients to check the integrity of remote files without retrieving files by regularly challenging the remote server to provide a proof of data possession. PDP [1] use homomorphic tags based on RSA to make integrity checking rather than the file itself. Original PoR make verifications by comparing responses from servers and prestored answers or "sentinels". In such a way, we achieve less communication overhead and the verification only needs a small and constant number of computations for the auditor. Following researches extend the original schemes to be equipped with some preferable properties, such as high efficiency [5, 6, 7, 8], public verification [5, 8, 9], unbounded use of queries [1, 5, 10], dynamics supporting [9, 10, 11, 12, 13] and retrievability of data [4, 5, 14], which are just our designing goals.

Among those above, one major focus is to design a PDP/PoR scheme that supports dynamic operations, enabling it to be used in dynamic settings, i.e., the scheme can also be used when clients modify, insert or delete their files, not only for static settings like libraries and scientific dataset [1, 3, 4, 5]. Generally, they apply authenticated structures, such as hash trees, skip lists, index hash tables to support dynamic settings. In this paper, we discuss several classical remote integrity checking schemes in detail and category them by the dynamic structure they use.

To verify remote storage, most of these schemes use tags [1, 3, 8, 10, and 14] that are of smaller size to represent blocks, which reduces communication bandwidth massively. Clients must generate secure tags, which mean these tags are unforgeable by anyone except the original maker. To help readers to grasp the idea of PDP/PoR, we throw light upon the block tags/signatures here in advance [13]. Usually in a PDP/PoR, auditors authenticate the block tags instead of original file blocks during the verification, but this does not mean we verify tags themselves only for the fact that tags cannot fully represent data blocks and the server may store tags well but not the file data. These tags will be used in the computation of verification and if in dynamic settings, they must maintain fresh and legitimate, even after update operations [13], otherwise it will fail to pass auditors verification. Note that some literature uses block signatures [5, 9, 11] to make verifications, which actually play the same role as tags.

Our paper is organized as follows: Section 1 introduces the background of cloud storage, the motivation of verifying remote files and some basic ideas of verification schemes. Section 2 shows two kinds of generic verification models, i.e., two-party model and three-party model, depending on whether the verification process is executed by clients themselves. In Section 3, we discuss some classical PDP/PoR schemes and analyze their mechanisms and techniques of making integrity verification after lucubrating classical literature on remote integrity checking. And Section 4 presents classical schemes that support dynamic verifications, which means these schemes can also work after clients append (insert), modify or delete their files. In Section 5, we compare the efficiency and performance of several typical remote checking schemes. And section 6 provides some remarks as conclusions and our future work.

2. Generic Verification Models

Having studied existing verification systems, we summarize them into two types of models according to the number of parties in the models, namely two-party and three-party models. Figure 1 shows a composition of the two models and depicts workflow for both models. Note that there're multiple clients, servers or even auditors in the models.

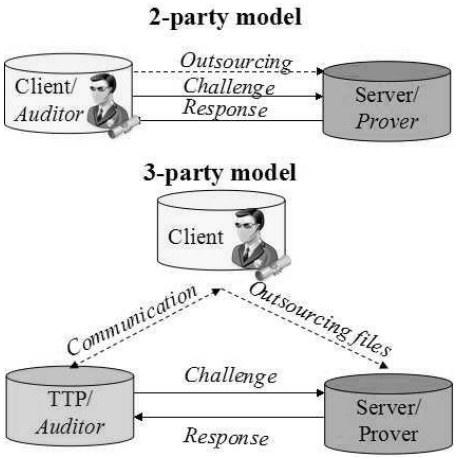


Figure 1. Generic Remote Verification Models

If a client himself verifies the remote storage, we get the two-party model, which contains two components: a client (auditor) and a server (prover). The client holds files and outsources them to the remote untrusted servers and later verifies the storage regularly. If the client delegates verification to a trusted third party (TTP) to relieve itself from heavy burden, then we get the three-party model. It mainly consists of three parties: a client, a server (prover) and TTP (auditor). The client outsources its data to the remote server and the auditor TTP checks the server for the client. In this model, the client generates metadata (signatures or tags) of his data and shares some public information with other two parties. Then the client deletes the local copy and TTP can verify whether the remote servers store the owner's files correctly when needed. The verification procedure mainly has four phases: Initialization, Challenge, Proof and Verification. Initialization is the procedure of system setup including the pre-processing of files and related start work of each party. When verifying, TTP sends challenges to the server, if the server can reply with a valid proof, the TTP can be convinced that the remote server behaves well and stores the data correctly.

Clients' remote files are out of their own physical control, so the servers may behave lazily or even maliciously for the sake of commercial benefit, e.g., they may not update corresponding files as clients requires to save storage and computation resources. When asked to make a proof, the server just sends previous proofs or regenerate proofs without updating files. When the files are corrupted or even lost, the server could still forge the metadata of the blocks and generate corresponding proofs to deceive the auditor. It can also be malicious, who may delete the outsourced files deliberately or even collude with adversaries for a certain purpose. Therefore, it is crucial to guarantee the intactness of outsourced data in the remote servers, and no sensitive information should be exposed to the auditor (in a three-party model) and outsiders when the auditing is executed. Note that in our model, the auditor is honest but curious, e.g., it behaves honestly during the verification but may be curious about data from the client and the server. Therefore, files data should not be leaked to the TTP during the verification. Our designing goals are to guarantee the integrity of remote files and avoid attacks above.

3. Classical Static Schemes

Dewarte et al. [3] first introduced the notion of "remote integrity checking" (RIC) to verify whether remote servers keep outsourced data intact (not delete or tamper with the data) without retrieving the original files. Follow up work, including PDP [1] and POR [4], provides us with formal models and techniques to verify the cloud storage in a more

efficient way. We'll delve into some classical static schemes in detail and classify them based on the

Techniques they use to verify storage.

3.1. MAC-Based Schemes

In cryptography, a message authentication code (MAC) is a short piece of information used to authenticate a message to provide integrity. It is often generated by a keyed hash function which allows auditors to detect any changes of the message content. In those MAC-based schemes, the client pre-computes MACs of files (or blocks) and sends these MACs and corresponding keys to the server. In the verification phase, the client sends a selected key to the server and waits for a corresponding reply. Then the server computes and sends MAC back to the client, which will later be checked by comparing with the pre-stored MACs. If they equals, the server are believed to store files well.

Juels and Kaliski [4] first presented the formal definition of Proof of Retrievability (POR) where the original file is first encoded, divided into blocks, and encrypted, and then some randomly selected blocks (so-called sentinels) will be embed into the encrypted file. These sentinels are indistinguishable from other blocks and disguise themselves in the file. When verifying, the auditor specifies the positions of some sentinels and send this challenge to the server, waiting for the corresponding reply. An obvious drawback is that the number of sentinels is limited (each challenge needs one), therefore times of challenging the server is limited.

Shacham et al. [5] proposed Compact Proofs of Retrievability (CPoR) for batch verification of multiple data blocks. In their scheme, the file data are encoded and divided into n blocks. And the client chooses a number $\alpha \in Z_p$ randomly and keeps a PRF (Pseudo-Random Function) key k_{PRF} to compute the MAC value for each block as a tag $\sigma_i = \alpha m_i + f_{k_{PRF}}$. Note that σ_i is the signature for the ith data block m_i. Then these MACs will be sent to the server together with the data blocks m_i. When verifying the server, the auditor (client) first sends a challenge $chal : (i, v_i)_{i \in Q}$ to the server, where $v_{i(i \in Q)} \in Z_p$ is a random coefficient? Upon receiving the *chal*, the server computes proofs of storage using $chal : \sigma = \sum_{(i,v_i)_{i \in Q}} v_i \cdot \sigma_i$ and $\mu = \sum_{(i,v_i)_{i \in Q}} v_i \cdot m_i$, and then sends them back to the auditor. After receiving the proofs, the auditor computes whether $\sigma = \alpha\mu + \sum_{(i,vi)Q} v_i f_{k_{PRF}}(i)$ holds.

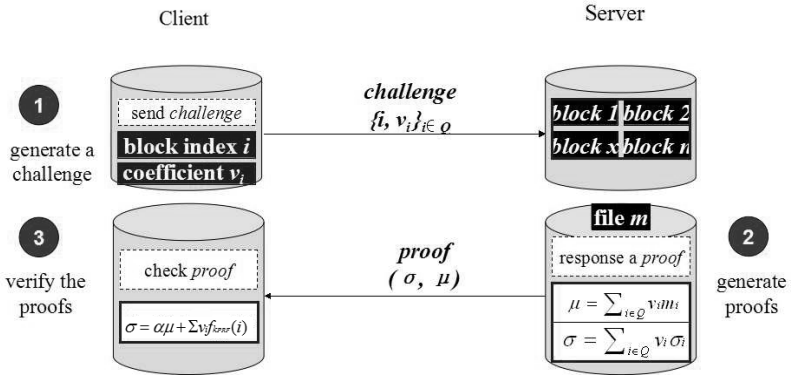


Figure 2. Challenge-Response Verification in CPoR-I

Here, there're two remarks should be noted: (1) scheme above does not support public verification so the auditor is the client itself; (2) an improvement by further dividing a

block into s sectors to give a tradeoff between storage overhead and response length can be done. Figure 2 shows the verification processes of CPoR-I.

3.2. RSA-Based Schemes

Responses taking advantage of homomorphism can reduce the bandwidth a lot in the "challenge-response" protocol and RSA-based homomorphic hash function can be used to verify data possession in remote servers. In Sebe *et al.* [6], the client generates an RSA-based homomorphic hash function for each data block. The client stores homomorphic hash values, which is not suitable for a TTP to make verifications. In [3], homomorphic tags of files are used to make integrity verification. Initially, the client randomly chooses a number $\alpha\ (1 < \alpha < N-1)$, computes homomorphic tag $T = \alpha^m \bmod N$ and then sends file *m* together with the corresponding tag to the server. When verifying the integrity of file *m*, the auditor challenges the server with *chal*: $C = \alpha^r \bmod N$, where $r \in (1, N-1)$ is a random value. Upon receiving *chal*, the server computes a corresponding *proof* $P = C^m \bmod N$ and sends it back to the auditor. Then the auditor checks whether $T^r \bmod N = P$ holds. Though the use of homomorphic tags is more efficient, the client still needs to store tags.

Ateniese et al. [1] first formally defined a PDP (Provable Data Possession) scheme, which uses RSA-based homomorphic tags and sampling checking to reduce bandwidth. In their protocol, file data are stored in blocks and the client generates *pk=(N,g)* and *sk=(e,d,v)* that satisfies *ed=1mod p'q'*, where *e* is a large secret prime and *v* is a random number. Then the data owner generates tags for all blocks m_i: $T_i = (h(W_i)g^{m_i})^d \bmod N$, where $W_i = v \| i$ and then sends $\{pk, \{m_i\}_{1 \le i \le n}, \{T_i\}_{1 \le i \le n}\}$ to the server. Afterwards, the owner can delete the original data and tags $\{T_i\}_{1 \le i \le n}$ locally.

In this scheme, k_1 is used to select *c* block indices $i_{s1}, \cdots, i_{sc}\ (s_1 \le j \le s_c)$ to be checked, which prevents the server pre-storing combination values of blocks. k_2 is used to generate coefficients $a_j (s_1 \le j \le s_c)$ that guarantees each block is intact. When verifying remote storage, the auditor (client) sends a challenge *chal* (c, k_1, k_2, g_s) to the server, where $g_s = g^s$ Then the server computes proofs (T, ρ), where $T = T_{i_{s_1}}^{a_1} \cdot \cdots \cdot T_{i_{s_c}}^{a_c} = (h(W_{i_{s_1}})^{a_1} \cdot \cdots \cdot h(W_{i_{s_c}})^{a_c} \cdot g^{a_1 m_{i_{s_1}} + \cdots + a_c m_{i_{s_c}}})^d \bmod N$ and $\rho = H(g_s^{a_1 m_{i_{s_1}} + \cdots + a_c m_{i_{s_c}}} \bmod N)$, and sends to the server. The auditor then verifies whether it holds. If so, the auditor accepts its proof and believes the remote files are well stored, and vice verse. Figure 3 shows the challenge-response procedure.

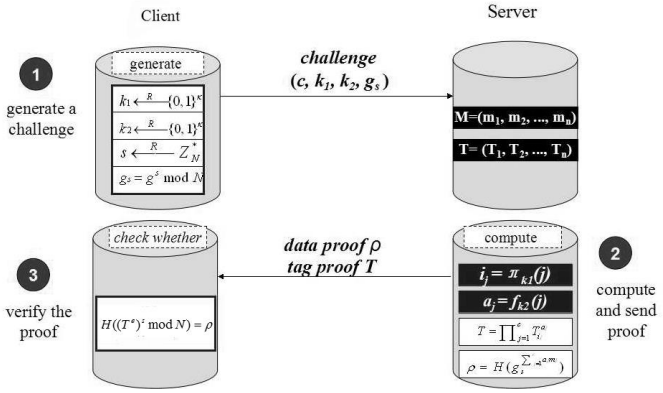


Figure 3. Challenge-Response Verification in PDP

The designing of this scheme is quite exquisite (e.g., W_i and g_s), avoiding several intractable issues compared with other auditing methods (e.g., elliptic curve based methods) and its idea of homomorphic tags has a great influence on followup works. However, this scheme has to deal with exponential computation and does not support public verification, but their another version manage this in [1].

3.3. BLS-Based Homomorphic Methods

The use of BLS (bilinear signature) can also compact responses to reduce communication bandwidth, however, it is not suitable for variable sized blocks [9] and the wanted groups are always limited in number in reality. Let G_1, $G2$ and G_T be three multiplicative groups with the same prime order p. A bilinear mapping e has following properties:

- Bilinearity: $e(u^a, v^b) = e(u, v)^{ab}$ for all $u \in G_1, v \in G_2$ and $a, b \in Z_p$.

- Non-degeneracy: There exists $u \in G_1, v \in G_2$, such that $e(u, v) \neq I$, where I is the identity element of G_T.

- Computability: e can be efficiently computed.

Shacham et al. [5] proposed a CPoR using BLS-based homomorphic tags (the second scheme), which supports public verification compared with their first scheme. Similar to schemes above, the client computes corresponding signature (tag) for block m_i as $T_i = [H(i)u^{m_i}]^x$, where u is a generator of G, and x is a private key. Here, public key g^x (g is another generator) is distributed to TTP. Then the client sends blocks m_i and corresponding tags to the server. When verifying, the auditor (TTP) sends a challenge (i,v_i) to the server, where v_i is a random coefficient. Upon receiving the challenge, the server computes corresponding proof (T, μ) back to the auditor, where $T = \prod T^{v_i}, (i, v_i) \in Q$ and $\mu = \sum v_i \cdot m_i, (i, v_i) \in Q$. Then the auditor checks whether $e(T, g) = e(\prod H(i)^{v_i} \cdot u^{\mu}, g^x)$ holds. If yes, the auditor is convinced that the files are well stored in the remote server. In this scheme, the tags are shorter and more efficient and they give a rigorous proof of security of the scheme that one can extract a file from a prover able to answer auditing queries convincingly. However, this scheme will leak information to the auditor since the server needs to send linear combination blocks back to the auditor. To resolve the issue, Wang et al. [15] proposed a privacy-preserving public scheme using random masking that is also used in [16], where the server chooses a random number to embed it into the response's computation. They claimed that their proposed scheme can resist various known attacks. However, a research [17] pointed out its security loopholes and it fails to resist some existing forgery attack and conspiracy of malicious servers and outsiders.

Zhu et al. [16] proposed an interactive PDP scheme to check data integrity as well as keeping privacy in untrusted cloud environment. The main idea of this protocol is also used in [13], which we'll discuss in section 4.4. So to save space, we'll explain the usage of BLS in that protocol detailedly later. They proved that their scheme is secure, in the zero-knowledge proof system (MP-ZKPS), which can resist data leakage attack and tag forgery attack. However, [17] demonstrates that Zhu's protocol lacks the soundness in the face of some threatens, e.g., malicious servers that generate valid responses may pass the verification even when they delete all outsourced data.

Hanser et al. [8] proposed a robust PDP based on elliptic curves which supports private and public verification simultaneously in one same system. The verification process is based on the properties of BLS rather than elliptic curves as the title shows. In this scheme, tags are computed and mapped into points and the properties of bilinear mapping for additive and multiplicative groups are used in the verification. In theory, the scheme

indeed enhances efficiency and provides convenience. However, the chosen of elliptic curves and bilinear mapping is not so easy in practical scene.

4. Schemes Supporting Dynamic Updates

Original PDP/PoR schemes are only suitable for static files, which can be applied in libraries and scientific datasets. However, in practical terms, clients update the outsourced data frequently, like inserting, modifying or deleting files or blocks. It is obvious that we have a great necessity to consider the dynamic verification for PDP/PoR schemes to render them the ability of making verifications in dynamic scenarios. Generally, in dynamic PDP/PoR schemes, the auditor not only verifies files' integrity but also checks whether remote servers have performed updates after clients' update request arrives. Remember that client needs to retrieve their data blocks if they want to update them. Now we'll show following structures used in PDP/PoR schemes that can achieve dynamic verifications.

4.1. Disguised/Special Blocks (Sentinels)

As is shown before, in POR [4], they embed some special blocks called *sentinels* into the data file that are indistinguishable from the file itself to detect server's misbehavior. There's a fact that the number of queries is limited for each challenge needs a sentinel and the pre-computed "sentinels" prevent the improvement of realizing dynamic data updates.

The scheme proposed by Ateniese et al. in [19] supports data dynamic operations like modifying, deleting and appending with the help of *tokens*. Note that there are *t* tokens and *c* blocks in each token. In this scheme, the client pre-computes *t* possible challenges and corresponding answers called *tokens* before uploading his files. Each token is computed as $t_i = H(c_i, m_{i_1}, \cdots, m_{i_c})$, where c_i is a challenge nonce to prevent pre-computed values by the server. We first have a look at the default verification without dynamics consideration. When verifying remote storage, the auditor sends challenge (k_i,c_i) to the server, where k_i is the *i*th token key to generate block indices to be challenged. Upon receiving the challenge, the server computers $z = H(c_i, m_{i_1}, \cdots, m_{i_c})$ and sends [z,t_i'] back to the client, where $t_i' = Enc(i, t_i)$ and the *c*-element set $\{i_1, \cdots, i_c\}$ denotes the indices of challenged blocks. In turn, the client computes whether $Dec(t_i') = (i, z)$ holds. Note that (*Enc*,*Dec*) is a encryption-decryption function pair.

To adapt dynamic settings, the token's structure has been modified to $t_i = H(c_i,1, m_{i_1}) \oplus \cdots \oplus H(c_i, c, m_{i_c})$ and $t_i' = Enc(ctr, i, t_i)$, where *ctr* is an integer counter to record the latest version of tokens and avoid replay attacks. The new structure is easy to factor out specified blocks to be updated. When the client wants to modify the *i*th block from values m_i to m_i' stored on the server, he needs to make a replacement of the corresponding blocks and modify all the remaining verification tokens. The deletion of a certain unwanted blocks is similar with that of modification and the deleted blocks can be replaced by a special block in respective positions. However, it is just fit for a small portion of deletion. Obviously, the cost of updating all remaining tokens is large, so they make a change to adapt their scheme to batch updates. For a block append operation, a bi-dimensional logical structure should be taken into consideration, and appending new blocks to the original blocks $m_{1,\ldots,}m_n$ in a round-robin way as:

m_1'=m_1, m_{n+1}
m_2'=m_2, m_{n+2}
...
m_k'=m_k, m_{n+k}

...

$m_n'=m_n$

Then we can use modification operation to append the blocks.

As is seen, such schemes above poor scale when blocks need to be updated and allow only for a limited number of audits [12]. So the following structures are used to enhance the performance of dynamic verifications.

4.2. Skip List Based Schemes

[10] First introduced the notion of Dynamic PDP with the help of skip list or hash trees. In this part, DPDP-I (skip list based one) will be discussed. A skip list was proposed by William Pugh [23] in as early as 1990. It is a data structure based on the idea of probabilistic balancing rather than strictly enforced balancing, which can be used instead of balanced trees while providing simpler implementation and faster speed. A skip list containing *n* elements has *log n* levels with high probability. The base level is a sorted list of the elements and a subset of these elements will also appear in upper levels. In a skip list, each node *v* stores two pointers: *rgt (v)* and *dwn (v)*. *rgt (v)* is a first pointer indicating a next node to the right of node *v* and *dwn (v)* is a second pointer indicating a next node below node *v*. The basic update such as insertion and deletion of nodes in a skip list can be depicted in figure 4. If we want to insert a node in the skip list, we should first to determine the level of this inserted node randomly. Figure 4 show that if the level of the inserted node is larger than the current top level, we should add a new level for usage. Then we find a proper position according to the search path for this node and then modify corresponding pointers. Deletion is of the same idea, i.e. first to search the node, then modify the pointer field.

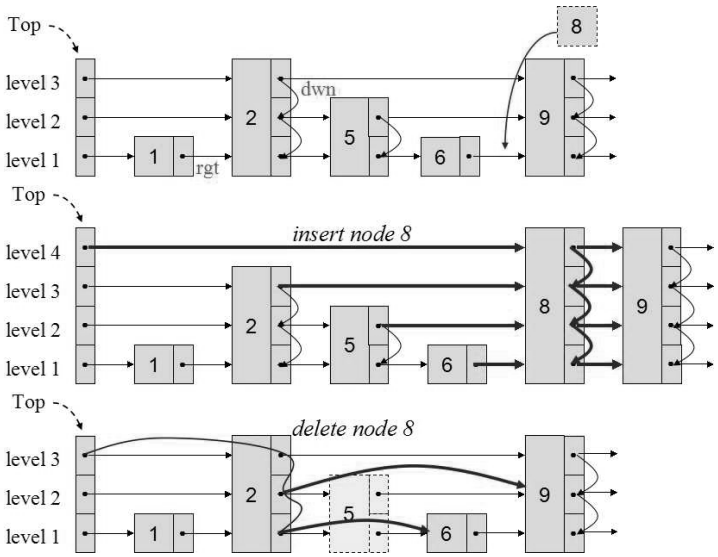


Figure 4. Dynamic Operations in a Skip List

In an authenticated skip list [24], each node has a label *f(v)*, which can be used to check the integrity of the file blocks. To make it support efficient verification of the indices of the blocks, Erway [10] made an improvement by defining a hash scheme with rank information in their first scheme DPDP-I, where the rank *r (v)* of node *v* denotes the number of nodes at the bottom level that *v* can reach. The skip list stores data at the bottom level nodes, e.g., the block tag T_i in this scheme. We can reach the *i*th node of the bottom level by traversing a path that begins at the start node. Based on work above, they proposed *hashing scheme with ranks* and the ranks of nodes are computed as follows:

$$f(v) = \begin{cases} 0, & v = null \\ h(l(v), r(v), f(dwn(v)), f(rgt(v))), & l(v) > 0 \\ h(l(v), r(v), x(v), f(rgt(v))), & l(v) < 0. \end{cases}$$

A bottom node store tags T_i to represent a block m_i, which has a much smaller size. In such a way, the client can verify the remote storage by just downloading tags instead of blocks to reduce the communication bandwidth. We use the skip list to protect the integrity of tags, where the very tags can protect the integrity of the blocks.

We start with the default verification of DPDP-I [10]. Assume $v_k,...,v_1$ is the path from the start node v_k to the specified node v_1 that is related to m_i and its reverse path is the verification path. If the auditor wants to verify the integrity of block m_i, he first sends *chal* to the server. Then the server works out T_i and a proof Π for T_i. They defined two values $q(v_j)$, $g(v_j)$ and a boolean $d(v_j)$, where $d(v_j)$ denotes whether the previous node is to the right or below v. $q(v)$ and $g(v)$ are the rank and label of the successor of v respectively, where v is not at the bottom level. So the proof for block m_i together with T_i is the sequence Π (i)=($A(v_1)$,...,$A(v_k)$), where $A(v)=(l(v),q(v),d(v),g(v))$. Upon receiving T_i of block m_i and the proof Π for it, the client computes ($l(v_j)$,$q(v_j)$,$d(v_j)$,$g(v_j)$) for each node v_j on the verification path together with a sequence of integers ε_j, where ε_j is the sum of the ranks of all nodes that are to the right of the nodes seen in the path so far. If T and Φ are correct, the auditor is convinced that remote files are well stored.

When a client wants to update his file, he'll first send an update request Update to the server, which returns T_i and its proof Π' back. The client verifies proof Π' and computes the label of the start node of the skip list after the update. Then it sends related update information to the server to perform updates. For an insertion of m_x after m_i, the server inserts corresponding T_x in the skip list after the ith element. For a modification of m_i into m_i', the server replaces T_i with T_i' of the skip list. And for a deletion of the m_i, the server deletes the i element of the skip list. Afterwards, the server updates the labels, levels and ranks of the affected nodes and returns (T_i',Π') (modification and insertion) or T_{i-1} (deletion). After updating, the server should send proofs to the auditor proving that it has updated files as required. Upon receiving the proof (T_i',Π'), the auditor will make update verifications. If so, the client will store the updated label of the start node and deletes the new block from its local storage. The updates affect only nodes along the verification path.

Note that in their scheme, they care about tags rather than real blocks in the skip list. When we verify the integrity of the blocks in static scenes, the process is similar with schemes before. The server computes proofs (T,M), where $T = \prod_{j=1}^{c} T_{ij}^{a_j} \bmod N$ and $M = \sum_{j=1}^{c} a_j m_i$ (a_j is a random number by the client as part of challenge). Then the auditor verifies whether $T=g^M$ holds. Here, T_i of block m_i is computed as $T_i = g^{m_i} \bmod N$, where $g \in Z_N^*$ is a generator of the group. We store the tags in the bottom-level nodes of the skip list rather than the blocks. As we see, with the help of skip list, we could make integrity verification when clients update their files in an efficient way.

4.3. Tree-Based Dynamic Schemes

Many researchers apply various authenticated tree structures in their verification schemes to support efficient dynamic verifications. These structures can guarantee the integrity of the tags and further data blocks. Erway [10] proposed an authentication scheme with Rank-based RSA Tree [22]. In this scheme, the server

keeps a RSA tree, where the leaves store elements S= {T_1, $T2$,..., T_n} to be verified. N_i and g_i are the RSA modulus and its base respectively for each $1 \le i \le l$. Each node v has a digest $\chi(v) = g^{\prod_{u \in N(v)} r_i(\chi(u))} \bmod N_i$, where $r_i(\chi(u))$ is a prime representative of $\chi(u)$ computed using h_i and N (v) is the set of children of node v. The server also stores $r_{i+1}(S_v)$, while the client only stores the set digest $d = \chi(S)$. Let v_0, $v1$,...,v_l be the path from T_i to the root R, r=v_l. Let B (v) denotes the set of siblings of node v. Proof $\Pi(x)$ is the ordered sequence π_1,..., π_l, where π_i of proof $\Pi(T_i)$ is computed as $\pi_i = (r_i(\chi(v_i)), g^{\prod_{u \in B(v_i)} r_i(\chi(u))} \bmod N_i)$. When verifying, the client checks whether $h_1(\alpha_1) = T_i, \cdots, h_i(\alpha_i) = \beta_{i-1}^{\alpha_i - 1} \bmod N_{i-1}$ and $d = \beta_l^{\alpha_l}$ holds, where $\alpha_i = r_i(\chi(v_i))$ and $\beta_i = g_i^{\prod_{u \in B(v_i)} r_i(\chi(u))} \bmod N_i$.

In dynamic settings, the elements are stored in a hash table that has several buckets, each storing several elements. The RSA tree is built on the prime representatives of *accumulated bucket values* rather than elements themselves. A bucket L has elements x_1,x_2,...,x_h, and they have the same value after the function of hash table. The *accumulated bucket value* of L is computed as $A_L = g_1^{r_1(x_1) r_1(x_2) \cdots r_1(x_h)} \bmod N_1$, so the tree has an extra level of accumulations. When we query for x, the server follows the path v_0,v_1,...,v_{l+1} and collects the corresponding pre-computed witnesses $\beta_1 = A_{j_1}^{(v_1)}, \cdots, \beta_{l'} = A_{j_{l+1}}^{(v_1)}$ for $j_1, \cdots, j_{l+1}$. If the client wants to insert an element x in the hash table, we first locate its bucket that it belongs to. And assume v_0,v_2,...,v_{l+1} be the path from the newly inserted element to the root of the tree. Accordingly, we update following items: digests $\chi(v_i)$ along the path from bucket L to the root, local witnesses $A_j^{(v_i)}$ for all nodes v_i, $1 \le i \le l'$. Modification and deletion are similar to that of insertion. Also the verification of updates and integrity of files can refer to DPDP-I and the scheme [20] below, we omit them here to avoid duplication.

Wang [12] proposed an auditing scheme using A Merkle Tree (MT) [20] to realize dynamic operations. It can handle dynamic data operations including modification, insertion (not only appending) and deletion. Previous schemes involve file index i in the tags generation. So when inserting a block, we need to re-compute following tags, which is a heavy work. In MT construction, they removed the block index i from tags, so individual block operation won't affect others. We'll discuss this scheme completely but the route information for verification is omitted because previous subsections have shown a lot.

Initially, the client uses *KeyGen* to generate a signing key pair (*spk*,*ssk*) and the secret-public key pair is (*sk*,*pk*), where $sk = (\alpha, ssk)$ and $pk = (g^{\alpha}, spk)(\alpha \in_R Z_p)$. Let $S = name \parallel n \parallel u \parallel SSig_{ssk}(name \parallel n \parallel u)$ be the file symbol. He generates tags for each block as $T_i = (H(mi) \cdot u^{m_i})^{\alpha}$ and then generates a root R for MHT, and the hashes of H(m_i) act as leaves. Then, the client computes $sig_{sk}(H(R)) = (H(R))^{\alpha}$ to signs R. Afterwards, he sends $\{F, S, \{T_i\}_{1 \le i \le n}, sig_{sk}(H(R))\}$ to the server and deletes $\{F, \{T_i\}_{1 \le i \le n}, sig_{sk}(H(R))\}$ his local copy.

Before discussing the implement of dynamics verification, we first have a look at the setup of the system and default verification, which is based on the properties of BLS.

If the auditor wants to verify the remote storage, he first sends a challenge $\{(i, v_i)\}_{s_1 \le i \le s_c}$ to the server, where {s_1,...,s_c} is a subset of [1,n]. After receiving the

challenge, the server computes proof $\{\mu, T, \{H(m_i), \Omega_i\}_{s_1 \le i \le s_c}, sig_{sk}(H(R))\}$, where $\mu = \sum_{i=s1}^{s_c} v_i m_i$, $T = \prod_{i=s1}^{s_c} T_i^{v_i}$ and the corresponding auxiliary information $\{\Omega_i\}_{s_1 \le i \le s_c}$ (we won't explain Ω here to save space) back to the auditor. The auxiliary information here refers to the node siblings on the path from the leaves $h(H(m_i))$ to the root R of the MHT. Upon receiving the proof, the auditor first generates root R using $\{H(m_i), \Omega_i\}_{s_1 \le i \le s_c}$ and checks whether $e(sig_{sk}(H(R)), g) = e(H(R), g^{\alpha})$ holds. If yes, he continues to check whether $e(T, g) = e(\prod_{i=s_1}^{s_c} H(m_i)^{v_i} \cdot u^{\mu}, g^{\alpha})$ holds. If yes, output success. Any checks failing to pass the verification indicate that the server does possess intact files.

When the client's update request arrives, the server executes the required dynamic operation using the information sent from the server. For modification, the client first computes the tag of the new block m': $T_i' = (H(m_i') \cdot u^{m_i'})^{\alpha}$. Then it will generate and send an update request (M, i, m_i', T_i') to the server to execute the update operation. The server then replaces block m with m', replaces T_i with T_i'. Then replaces $H(m_i)$ with $H(m_i')$ in the MHT. Afterwards, the server generates a *proof*: $\{\Omega_i, H(m_i), sig_{sk}(H(R)), R'\}$, where Ω_i is the authentication information of m_i'. Now the owner will generates root R using $\Omega_i, H(m_i)$ and verify R by checking whether $e(sig_{sk}(H(R)), g)$ equals to $e(H(R, g^{\alpha}))$. If so, the owner will computes the new root value and compare it with R'. If true, the owner will make a signature $sig_{sk}(H(R'))$ and send it to the server for update. Of course, if any one of the verification is false, the process will not proceed. When we want to insert a block m' into this tree, we can refer to the modification operation. Therefore, we just need to know the sketch of the operation for sake of space saving. To achieve this, we present a figure depicting the process. As is shown in Figure 5, if we want to insert a block m' after m_2, we'll need an extra internal node I except the inserted node N to add into the tree. So when we want to delete block m', we'll also delete node N and all the latter blocks it forward. Figure 6 shows the process of insertion and deletion in a MT tree. As we can see, the structure of the tree will not change when modifying data. However, when it deals with inserting or deleting, the tree's structure may be dramatically changed with the operations increasing.

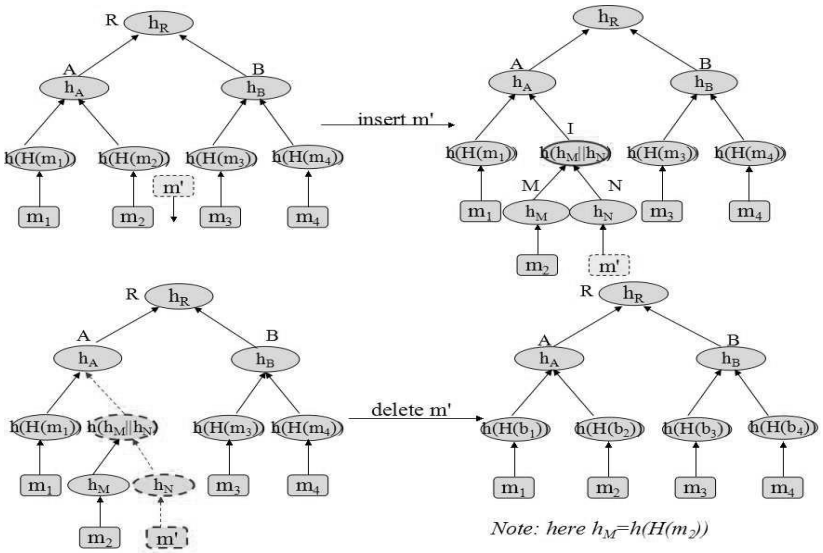


Figure 5. Insertion and Deletion in MHT

As we see above, one obvious disadvantage of trees above is that it becomes unbalanced after several insertion and deletion, resulting in the height of the tree growing or decreasing dramatically and locating different blocks in the tree consumes quite different time. Zhang and Blanton [12] proposed a balanced update tree to deal with issues above and used the notion of range to enhance efficiency ([30] also uses the range-based trees). In the update tree, each node represents a range of block indices rather than a specific index, where operating on ranges helps to lower the size of the tree. One feature of the tree is that the range of a node's left child contains data blocks whose indices are lower than *L* and the range of the child contains indices larger than *U*, where [*L*, *U*] is an attribute of a node indicating the range of it covers. This benign property favors the balancing of the tree using AVL trees like algorithms.

Then, we'll show how these dynamic operations are performed. In this scheme, unlike [12], both the client and the server store the update tree to omit verification for updates. To perform an update (modification, insertion or deletion), the client first modifies the tree, computes MACs of the updated blocks and informs server of these changes. Upon reception of this update request, the server modifies the tree accordingly. If we want to modify a range of blocks, we can insert an internal node indicating the version of the blocks has increased. If we want to insert a range of blocks, an extra node with the new blocks will be added into the tree, and the indices of the following blocks will increase by the number of inserted blocks. To delete a range of blocks, the corresponding node will be marked with operation type "-1" and the offset (explained later) of the following blocks will be adjusted accordingly.

Take figure 6 as an explanation of the detail operations. Each node is assigned a set of attributes: [*L*, *U*], {*Op*, *V*, *ID*, *R*}. [*L*, *U*] specifies the start and end indices of the data blocks; *V* is the version number indicating the number of modifications performed on the data blocks. Op represents the operation type of this node where 1, 0 and -1 represents insertion, modification and deletion respectively. A node's *ID* has different meanings according to its type (we won't explain it further here). Offset *R* indicates the number of data blocks that have been added to or deleted from, and the range of data block indices preceding the range of the node.

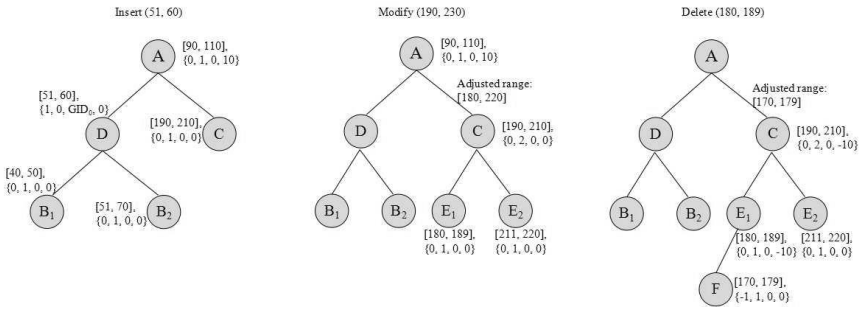


Figure 6. Dynamic Operations in Ranged Update Tree

The initial state: node A has two children B and C and their ranges are [90, 110], [40, 70] and [190, 210] respectively. If we want to insert a range into this tree, say [51, 60] as is shown in the leftmost tree. And the inserted range falls on left side of node A's range and intersects overlaps with the range of node B. So before inserting the ranged blocks, we first divide B's range into two and set D's Op into 1 indicating this insertion. The middle one shows the modification of blocks, where the modified range falls on the right of node A's range. So we modify the block range contained in the original request based on A's offset R, and thus overlaps with C's range. So we insert two nodes with ranges before and after C's range and

increment the version of C (the insertion here is just as former descriptions). And when deleting a block, as is shown in the rightmost of the figure, its range falls on the right of A and the indices in the original request are adjusted. For the adjusted range falls before C's all ranges, it will be inserted as the left child of E_1 and corresponding attributes will be adjusted.

Stefanov et al. [21] presented an authenticated file system name Iris, which supports authentications of file-system data and meta-data blocks and also handles all existing file system operations, like deleting, moving and truncating while still keeping the tree balanced. Their protocol is claimed to be the first PoR to efficiently support dynamic operations over the entire file system. The authentication of dynamics is based on Merkle trees and the scheme supports existing file-system operations.

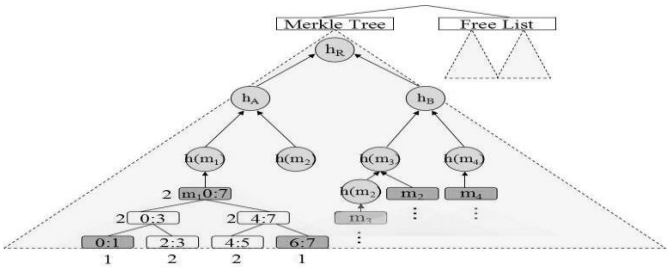


Figure 7. Balanced Merkle Tree in Iris

As is shown above in Figure 7, a file version tree for each file is constructed that authenticates version numbers for all file blocks in a compressed form. Also to authenticate file-system meta-data, every directory is mapped to a directory subtree and the file-system directory tree is transferred to a Merkle tree to support file-system operations like delete or move the entire directories. Aiming at supporting directories with large number of files efficiently, they construct a balanced Merkle tree for each directory that contains intermediate, empty internal nodes as well as subdirectory nodes for balancing. They also construct a free list that contains pointers of nodes deleted from the data files to support remove and truncate file-system operations and defer garbage collection of deleted nodes as an optimization of their scheme.

4.4. Index Hash Table Based Schemes

To support dynamic operations, [13] introduces an index-hash table to record the changes of file blocks and generate corresponding hash value for each block in the verification process. The index-hash table consists of serial number (the index of block m_i), block number B_i (the original number of block), version number V_i (the version of update for the specified block) and an extra random integer R_i to avoid collision. Let $\chi = \{\chi\}_{i\in[1,n]}$ be the index-hash table, where $\chi_i = (B_i \| V_i \| R_i)$. Figure 8 from [13] shows an example of the index-hash table and an empty record (i=0) is used to support the operations on the first record. We can implement required dynamic operations in the index-hash table, which provides a higher probability of detection, however, at the cost of more complexity of the verification.

No.	B_i	V_i	R_i	
0	0	0	0	← Head
1	1	2	r'_1	← Update
2	2	1	r_2	
3	4	1	r_3	← Delete
4	5	1	r_5	
5	5	2	r'_5	← Insert
…	…	…	…	
n	n	1	r_n	
n+1	n+1	1	r_{n+1}	← Append

Figure 8. Structure of Index Hash Table

We begin with their basic constructions:

Setup: g and h are two generators of G, where G is a multiplicative group with order p. $e: G \times G \to G_T$ is a bilinear mapping, where G_T is also a multiplicative group of the same order as G. The KeyGen algorithm outputs $sk = (\alpha, \beta)$ and $pk = (g, h, h^{\alpha}, h^{\beta})$, where $\alpha, \beta \in Z_p$ are randomly selected numbers. The client divides file F into n blocks, each having s sectors. So each sector is denoted as $m_{i,j}$ $(1 \le i \le n, 1 \le j \le s)$. Set $\xi^{(1)} = H_{\xi}(F_n)$, where $\xi = \Sigma_{j=1}^{s} k_j$ ($k_j \in Z_p$ are s numbers only known to the client) and F_n is the file name of F. Then he computes an index $\chi_i = (B_i = i, V_i = 1, R_i)$ and a tag $T_i = (\xi_i^{(2)})^{\alpha} g^{\Sigma_{j=1}^{s} k_j m_{ij} \beta} \in G$ for each block, where $\xi^{(2)} = H_{\xi^{(1)}}(\chi_i)$. Afterwards, the client sends (m_{ij}, T_i) to the server and $(\xi^{(1)}, g^{k_1}, \cdots, g^{k_s})$ to the auditor.

Challenge-Response: In this protocol, the server will send a commitment $C = ((h^{\alpha})^{\gamma}, \pi)$ to the auditor before the auditor challenges it, where γ is randomly selected from Z_p and $\pi = e(\Pi_{j=1}^{s} u_j^{\lambda_j}, (h^{\beta})^{\gamma})$. Then the auditor will know the server is ready for the verification and send it a challenge $chal : \{(i, v_i)\}_{i \in Q}$, where Q is the set of challenged blocks. Upon receiving *chal*, the server computes a linear combination value of blocks μ_j as $\mu = \{\mu_j\} = \{\lambda_j + \gamma \cdot \Sigma_{(i,v_i) \in Q} v_i m_{ij}\}$ and the combination value of tags $T = \Pi T_i^{\gamma \cdot v_i}, (i, v_i) \in Q$. Here λ_j is a random number. Then it sends $proof$ (T, μ) to the auditor. Upon receiving the *proof*, the auditor checks whether $\pi \cdot e(T, h) = e(\Pi_{(i,v_i) \in Q} (\xi_i^{(2)})^{v_i}, (h^{\alpha})^{\gamma}) \cdot e(\Pi_{j=1}^{s} u_j^{u_j}, h^{\beta})$ holds.

Now we take dynamic operations into consideration.

For modification, the client first modifies the version number by $V_i = \max_{B_i = B_j} \{V_j\} + 1$ and uses another random number R_i to gain a fresh index-hash χ_i'. Then he computes the new hash $\xi_i^{(2)} = H_{\xi^{(1)}}(B_i \| V_i \| R_i)$, and the new tag by $T_i' = (\xi_i^{(2)})^{\alpha} \cdot (\prod_{j=1}^{s} u_j^{m_{ij}'})^{\beta}$, where $u = u_j \in \varphi$ and outputs $\{\chi_i', T_i', m_i'\}$. For deletion, the client computes the new hash $\xi_i^{(2)} = H_{\xi^{(1)}}(B_i \| 0 \| R_i)$ and $T_i' = (\xi_i^{(2)})^{\alpha}$, deletes ith record to get a new φ' and then outputs $\{\chi_i', T_i, T_i'\}$. For insertion, the client first inserts a new record in ith position of the index-hash table $\chi \in \varphi$, and the following records move backward in order. It also update $B_i = B_{i-1}$, and $V_i = \max_{B_i = B_j} \{V_j\} + 1$, and a random R_i to get a new hash $T_i' = (\xi_i^{(2)})^{\alpha} \cdot (\prod_{j=1}^{s} u^{m_{ij}'})^{\beta}$, where $u = \{u_j\} \in \varphi$, and then outputs $\{\chi_i', T_i', m_i'\}$. After the update operation, the auditor can verify

whether the server executes the update as required. For Modification or Insertion, the auditor must check whether $e(T_i^{'}, h) = e(\xi_i^{(2)}, h^{\alpha}) \cdot e(\Pi_{j=1}^{s} u_j^{m_{ij}^{'}}, h^{\beta})$ holds for $\{\chi_i^{'}, T_i^{'}, m_i^{'}\}$. For Deletion, the auditor checks T_i by comparing it with the stored one and verifies whether $e(T_i^{'}, h) = e(H_{\xi^{(1)}}(B_i \| 0 \| R_i), h^{\alpha})$ holds. In [25], they use the structure of chained hash of a proof to retain freshness of the data. It is a hash over the data in the current proof and the chain hash of the previous proof, where the idea can be seen in Zhu's [13] protocol. We'll not discuss it in detail.

In summary, a skip list can be seen as a special hash tree, and it achieves similar efficiency as hash trees but provides fast search and simple implementation. Hash trees are a generalization of hash lists and hash chains. While a tree structure has higher probability of detection. In conclusion, we recommend use hash trees in the verification protocols. Note that these structures can be used in one scheme to achieve better performances.

5. Efficiency and Performance

When a protocol is designed, we often make a measurement for it to test its performance based on some criteria. Cost of computation, communication and storage should be taken into consideration. Owing to the complexity of the cloud storage, we have a necessity to discuss other respects of the protocols to valuate their performance, like supporting public verification and dynamic operations, and maintaining privacy. This section presents the performance of some integrity auditing protocols from various aspects. In former parts, we have elaborated the importance of dynamics supporting, and there's a necessity to make an auditing protocol equipped with public verifiability in cloud environment. Though private verification can achieve higher efficiency, but in practicality public verification can help achieve economies of scale for cloud computing. Refer to Table I to get a general view.

Table 1. Comparison of Integrity Checking Schemes for A File (N Blocks)

Scheme	Storage		Communication	Computation		Dynamics	Public verif.
	Server	Client		Server	Auditor		
RIC [3]	O(1)	O(1)	O(1)	O(n)	O(1)	×	×
PDP [1]	O(n)	O(1)	O(1)	O(1)	O(1)	×	√
CPOR-I [5]	O(n)	O(1)	O(1)	O(1)	O(1)	×	√
CPOR-II [5]	O(n)	O(1)	O(1)	O(1)	O(1)	×	√
SPDP [19]	O(n)	O(1)	O(1)	O(1)	O(1)	[1]√	×
DPDP-I [10]	O(n)	O(1)	O(log n)	O(log n)	O(log n)	√	×
DPDP [13]	O(n)	O(1)	O(log n)	O(log n)	O(log n)	√	×
DPDP-II [10]	O(n)	O(1)	O(1)	O(1)	O(1)	√	√
MHT-POR [9]	O(n)	O(1)	O(log n)	O(log n)	O(log n)	√	√
UTree-EPDP [12]	O(n)	[2] O(m)	[3] O(t)	O(log m+t)	O(log m+t)	√	√

1 Insertion is not supported but appending and number of dynamic operations is limited.

2 m refers to the number of dynamic operations on blocks.

3 t refers to the number of blocks in a range.

Storage overhead: In some MAC-based methods [4], the metadata is as long as each data blocks and the storage overhead of metadata is the same as the data. In RSA-based homomorphic methods, the size of metadata is equal to the size of RSA modulus. In order to reduce the storage overhead caused by metadata, people prefer to design short metadata in verification protocols. Compared to the RSA-based homomorphic tags, the BLS-based signatures/tags are much shorter. For a third auditor, it always does not have much storage as the server. We should remove storage burden from TTP as much as possible and may transfer them to servers. In [5, 7, and 8], blocks are divided into sectors to balance storage and communication. In fact, less storage on the auditors, large communication cost will be. These are methods mainly used in remote checking scheme to save storage.

Communication cost: We mainly consider the communication cost in the challenge-response phases. According to previous work, we find that spot checking [1, 5, 8, 9, and 10] can reduce the communication massively. Also, with the help of short homomorphic tags, we can aggregate multiple responses into one to reduce communication cost.

Computation complexity: Before outsourcing files to remote servers, clients may encode files (when needed) and generate related metadata. For servers, in [3], they need to exponentiate the entire data. To reduce the expensive computation cost, dividing files into blocks [1, 5] or further into sectors [5, 7, 8] is one preferable approach. For auditors, we may try to transfer the computation cost to the remote storage server that is more powerful.

In cryptography, a system has provable security if its security requirements can be stated formally in an adversarial model as opposed to heuristically, with clear assumptions that the adversary has access to the system as well as enough computational resources. Those auditing protocols guarantee security in different model, e.g., in standard model or random oracle.

Table 2. Comparison of Security for Integrity Checking Schemes

Scheme	Security model	Assumption	Privacy	Retrievability
RIC [3]	SM	DH	√	×
PDP [1]	RO	IF,DH	×	×
SPDP [19]	RO	N/A	√	√
DPDP-I [10]	SM	IF,DH	√	×
DPDP-II [10]	SM	IF,DH	√	×
POR	SS	N/A	√	√
CPOR-I [5]DPDP [13]	RO	IF,DH	×	√
CPOR-II [5]	RO	CDH	×	√
DPDP [13]	RO	CDH	√	√
MHT-POR [9]	RO	CDH	√	√
UTree-EPDP [12]	SS	MAC	√	√

* SM: Standard Model, RO: Random Model, DL: Discrete Logarithm, IF: Integer Factorization, DH: Diffie-Hellman, CDH: Computational DH.

Security guarantees that (1) from any adversary that passes the check a non-negligible amount of the time we will be able to extract a constant fraction of the

encoded blocks; (2) if this constant fraction of blocks is recovered we can use the erasure code to reconstruct the original file. And in the third party model, we should guarantee the privacy of the files during the verification process to avoid information leakage to the third party auditor; (3) the attacker can never give a forged response back to the verifier.

In table 2, we compare items related to security of these verification protocols, including security model, assumption and privacy.

6. Discussion and Conclusions

PDP and POR schemes are widely used in cloud storage to verify the integrity of outsourced data. According to discussions above, we make several remarks as our conclusions and future work.

Remark 1:

PDP and POR schemes can be used to check the integrity of remote outsourced data, and they have similar verification process in their construction. They are becoming to merge into one now from the observation of recent work. We now conclude as follows:

1) Tag T_i for each block in PDP [1, 8, 11, 12, 16] schemes play the same role as signature σ_i for each block in some POR schemes [4, 5, 9], and for convenience, we unify them as T_i in this paper. They all represent original blocks to some extent and be used in the verification process.

2) In the challenge phase, both of them generate (I, v_i) as challenge, where I is the index set of the challenged blocks, \$k\$ is the key to generate these indices and v_i is randomly selected numbers to be used in the proof computation for servers.

3) In the proofs generation of both schemes, servers compute (T, M) as proofs of the challenged blocks, where T is the tags proof and M is the blocks proof.

On the other hand, PDP and POR schemes have several subtle differences as follows:

1) In POR schemes, the client should first encode files to be outsourced, while such a process is not necessary in PDP schemes. As a result, POR schemes can recover damaged files, while in some PDP schemes files without encoding cannot manage this.

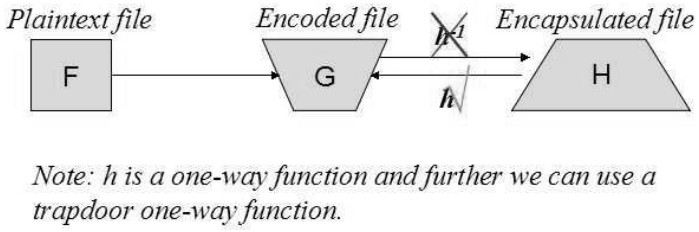
2) In [1], blocks of the same contents may result in the same tags if we remove $W(i \| v)$, which will leak information of blocks to the and outsiders. However, in [9], same tags for blocks of the same contents will not leak information because of the encoding. Later work has obscured this difference and solved the issue.

Remark 2:

RIC is just one aspect of remote data checking, in this section we discuss another aspect briefly: data format checking. Remote servers may not store the outsourced data in a safer format and can tolerant fault as clients require for self benefits. And clients generally have no effective approaches to verify the vulnerability of their data.

Dijk, Juels and Oprea discussed the challenge above. They want to enable clients to check whether a cloud provider stores outsourced data in encrypted forms while at rest, so they propose a novel hourglass scheme in [26] to prove this. The initial idea is to ensure servers have strong incentives to store encrypted files rather than original ones from the perspective of economic security. A given encoded file G is transformed into a special encoding file H according to clients' preference. Core idea of hourglass scheme is less expensive for a server to comply fully with an hourglass scheme and store only encrypted data than to cheat and store an additional, unencrypted copy of the data. It's clear to see what we want is to construct such a

proper hourglass function to achieve above goals. One possibility is using inversion of hash function as Figure 9 shows.



Note: h is a one-way function and further we can use a trapdoor one-way function.

Figure 9. Mechanism of Hourglass

A benign server stores the encapsulated ciphertext H and thus recovery of G from H is feasible with the help of hourglass scheme via the computation $G_i = h(H_i)$. However, a dishonest server that stores plaintext file F has to compute ciphertext block G_i and then compute H, which is costly as the inversion of h is a computational dilemma, which will delay the response of a dishonest server. Therefore a client can distinguish a dishonest server from a benign one. However, there exists another threaten that a adversarial server may construct H' that leaks parts of the plaintext file F. Authors of this paper construct three constructions of H to overcome such concerns. Details of the construction are out of the scope of this paper. Generally speaking, we can verify whether files are encrypted (or other encoded forms) in remote servers with the resource-constraint-based hourglass scheme.

They also develop a protocol called Remote Assessment of Fault Tolerance (RAFT) [27] to check the fault tolerance of stored files, which enables a client to have the proof of the well distribution across physical storage devices of given files to achieve ideal fault-tolerance. Their main idea is the usage of measurement of the time taken for a server to respond to a read request for a given set of blocks. Some proposed schemes take advantage of the dilemma of computation problems to make a measurement of computational resources [28]. Both storage and computational resources are taken into consideration in designing their protocols. They proposed a practical lock-step protocol to generate subsequent steps non-interactively, where the blocks indices challenged in the next step depend on the current retrieved blocks.

Remark 3:

We should not constraint our resolutions to classical cryptography only, quantum cryptography or optical cryptography can also be taken into consideration. For an instance, most previous PDP protocols are insecure when quantum computers are considered [29]. In [29], they propose a homomorphic hash-based PDP (HH-PDP) protocol under lattice assumptions, which generates homomorphic verification tags. The security of the protocol relies on the hardness of ideal lattice problems. In addition, the protocol is more efficient because the main operations in our construction are addition and multiplication on small integers. Extending our vision to propose remote storage checking schemes using other cryptography tools is a promising research aspect in our future work.

Acknowledgements

This paper is supported by DNSLAB, China Internet Network Information Center , Beijing 100190, National 863 Plans Projects No. 2013AA01A214, Program for New Century Excellent Talents in University (NCET-12-0046), National Science Foundation of China under Grant No. 61100172, No. 61272512, and No. 61300177, Beijing

Municipal Natural Science Foundation No.4121001, Basic Research Foundation of Beijing Institute of Technology No.20120742010 and No.20130742005. The authors would like to thank anonymous reviewers for their valuable suggestions.

References

[1] G. Ateniese, R. Burns, R. Curtmola, J. Herring, L. Kissner, Z. Peterson and D. Song, "Provable Data Possession at Untrusted Stores", Proceedings of the 14th ACM Conference on Computer and Communications Security, CCS, **(2007)** , pp. 598-609, ACM, New York, NY, USA.
[2] R. Kotla, L. Alvisi and M. Dahlin, "A Durable and Practical Storage System, USENIX Annual Technical Conference", **(2007)**, Santa Clara, CA.
[3] Y. Deswarte, J. Quisquater and A. Saidane, "Remote Integrity Checking, The Sixth Working Conference on Integrity and Internal Control in Information Systems (IICIS)", Springer, **(2007)**, Netherlands.
[4] A. Juels and B. S. Kaliski, "Proofs of Retrievability for Large Files, Proc. 13th ACM Conf. Proceedings of the 14th ACM Conference on Computer and Communications Security, CCS", **(2004)** , New York, NY, USA.
[5] H. Shacham and B. Waters, "Compact Proofs of Retrievability, Proceedings of the 14th International Conference on the Theory and Application of Cryptology and Information Security", Advances in Cryptology, ASIACRYPT, **(2010)**, pp. 90–107, Springer, Berlin, Heidelberg.
[6] F. Sebe, J. Domingo-Ferrer, A. Martinez-Balleste, Y. Deswarte and J. J. Quisquater, "Efficient Remote Data Possession Checking in Critical Information Infrastructures", IEEE Trans, Knowledge Data Engineering, vol. 20, **(2008)**, pp. 1034–1038.
[7] Y. Zhu, H. Wang, Z. Hu, G. Ahn, H. Hu and S. Yau, "Efficient Audit Service Outsourcing for Data Integrity in Clouds", The Journal of Systems and Software, vol. 85, **(2012)**, pp. 1083– 1095.
[8] C. Hanser and D. Slamanig, "Efficient Simultaneous Privately and Publicly Verifiable Robust Provable Data Possession from Elliptic Curves", in proceedings of SECRYPT, **(2013)**.
[9] Q. Wang, C. Wang, J. Li, K. Ren and W. Lou, "Enabling Public Verifiability and Data Dynamics for Storage Security in Cloud Computing", Proceedings of the 14th European conference on Research in Computer Security, ESORICS, **(2009)**, pp. 355–370, Springer, Berlin, Heidelberg.
[10] C. Erway, A. Kupc, C. Papamanthou and R. Tamassia, "Dynamic Provable Data Possession", Proceedings of the 16th ACM Conference on Computer and Communications Security, **(2009)**, pp.213–222.
[11] X. Wang and S. Liu, "Dynamic Provable Data Possession with Batch-Update Verifiability", In IEEE International Conference on Intelligent Control, Automatic Detection and High-End Equipment (ICADE), **(2012)**, Beijing, China.
[12] Y. Zhang and M. Blanton, "Efficient Dynamic Provable Possession of Remote Data via Balanced Update Trees, ASIA CCS", **(2013)**, May 8–10, Hangzhou, China.
[13] Y. Zhu, H. Wang, Z. Hu, G. Ahn, H. Hu and S. Yau, "Dynamic Audit Services for Outsourced Storages in Clouds, CCS", **(2012)** October 16–18.
[14] G. Ateniese, R. Burns, R. Curtmola, J. Herring, L. Kissner, Z. Peterson and D. Song, "Remote Data Checking Using Provable Data Possession", In 13th ACM Symposium on Access Control Models and Technologies, **(2008)**, Estes Pk.
[15] C. Wang, Q. Wang, K. Ren and W. Lou, "Privacy-Preserving Public Auditing for Data Storage Security in Cloud Computing, In Proceedings of the 29th Conference on Information Communications, INFOCOM", **(2010)**, pp. 525–533.
[16] Y. Zhu, H. Hu, G. J. Ahn and M. Yu, "Cooperative Provable Data Possession for Integrity Verification in Multicloud Storage", In IEEE Transactions on Parallel and Distributed Systems, **(2012)**.
[17] C. Xu and X. He, "Cryptanalysis of Auditing Protocol Proposed by Wang et al. for Data Storage Security in Cloud Computing", In 3rd International Conference on Information Computing and Applications (ICICA), **(2012)**, Chengde, China.
[18] H. Wang, and Y. Zhang, "On the Knowledge Soundness of a Cooperative Provable Data Possession Scheme in Multicloud Storage", In IEEE Transactions on Parallel and Distributed Systems, vol. 23, no. 12, **(2012)**, pp. 2231-44.
[19] G. Ateniese, R. Pietro and L. Mancini, "Scalable and Efficient Provable Data Possession, SecureComm", **(2008)**, Istanbul, Turkey.
[20] R. C. Merkle, "Protocols for Public Key Cryptosystems", In Proc. IEEE Symp, Security and Privacy, **(1980)**, pp. 122-133.
[21] E. Stefanov, M. Dijk and A. Oprea, "Iris, A Scalable Cloud File System with Efficient Integity Checks", In ACSAC Proceedings of the 28th Annual Computer Security Applications Conference, **(2012)**, pp. 229-238.
[22] C. Papamanthou, R. Tamassia and N. Triandopoulos, "Authenticated Hash Tables", In CCS, **(2008)**, pp. 437-448.

[23] W. Pugh, "Skip Lists, a Probabilistic Alternative to Balanced Trees", In Commun, ACM, vol. 33, no. 6, **(1990)**, pp. 668-676.
[24] C. Papamanthou, R. Tamassia and A. Schwerin, "Implementation of an Authenticated Dictionary with Skip Lists and Commutative Hashing", In DISCEX II, **(2001)**, pp. 68-82.
[25] R. A. Popa, J. R. Lorch, D. Molnar, H. J. Wdang and L. Zhuang, "Enabling Security in Cloud Storage SLAs with CloudProof", In USENIX Annual Technical Conference, **(2011)**, pp. 355–368.
[26] M. V. Dijk, A. Juels and A. Oprea, "Hourglass Schemes, How to Prove that Cloud Files Are Encrypted", In CCS, **(2012)**.
[27] K. D. Bowers, M. V. Dijk, and A. Juels, "How to Tell if Your Cloud Files Are Vulnerable to Drive Crashes", In CCS '11 Proceedings of the 18th ACM conference on Computer and communications security, **(2011)**, pp. 501-514.
[28] A. Juels and J. Brainard, "Client puzzles, A Cryptographic Countermeasure against Connection Depletion Attacks", In Proc. ISOC NDSS, **(1999)**, pp. 151–165.
[29] L. Chen, L. Han and J. Jing, "A Post-Quantum Provable Data Possession Protocol in Cloud, In Security and Communication networks", **(2013)**.
[30] Q. Zheng and S. Xu, "Fair and Dynamic Proofs of Retrievability, In CODASPY", **(2011)**, San Antonio, Texas, USA.

Authors

Yu Chen, Born in 1988, Master Degree Candidate. Her research interests include security in cloud computing and provable data possession.

Feng Wang, Born in 1977, Ph.D., senior engineer. His research interests include low-power networking, sensor networks and Internet of Things technology and applications.

Liehuang Zhu, Born in 1976, Ph.D., professor. His research interests include security analysis of cryptography protocol, security of internet of things and cloud computing security.

Zijian Zhang, Born in 1984, Ph.D, associate professor. His research interests include security of protocol analysis, cloud computing and cost-friendly differential privacy for smart meters.

