Auditing for Data Integrity and Reliability in Cloud Storage

Bingwei Liu and Yu Chen

1 Introduction

As a new computing paradigm, cloud computing has enhanced the data storage centers with multiple attractive features including on-demand scalability of highly available and reliable pooled computing resources, secure access to metered services from nearly anywhere, and displacement of data and services from inside to outside the organization. Due to the low cost of storage services provided in the cloud, compared with purchasing and maintaining storage infrastructure, it is attractive to companies and individuals to outsource applications and data storage to public cloud computing services.

Outsourcing data to remote data centers that are based on cloud servers is a rapidly growing trend. It alleviates the burden of local data storage and maintenance. Security and privacy, however, have been the major concerns that make potential users reluctant to migrate important and sensitive data to the cloud. The fact that data owners no longer possess their data physically forces service providers and researchers to reconsider data security policies in the storage cloud. On one hand, evidences such as data transmission logs can prevent disputation among users and service providers [8–11]; on the other hand, the service providers need to convince users that their data stored in the cloud is tamper free and crash free, and that their data can be retrieved anytime when needed. Traditional cryptographic methods cannot meet these new challenges in the new paradigm of cloud storage environments. Downloading the entire data set to verify its integrity is not practical due to constraints of the communication network and the massive amount of data.

B. Liu (✉)
Department of Electrical and Computer Engineering, Binghamton University,
State University of New York, Binghamton, NewYork, USA
e-mail: bliu@binghamton.edu

Y. Chen
Department of Electrical and Computer Engineering, Binghamton University,
State University of New York, Binghamton, NewYork, USA
e-mail: ychen@binghamton.edu

© Springer Science+Business Media New York 2015

S. U. Khan, A. Y. Zomaya (eds.), *Handbook on Data Centers,*
DOI 10.1007/978-1-4939-2092-1_17

Integrity and reliability of data in the cloud are not inherently assured. On the one hand, cloud service providers themselves face the same threats that traditional distributed systems need to handle. On the other hand, cloud service providers have incentive to hide data loss or to discard parts of user data without informing the user, since they aim at making profit and need to maintain their reputation. Trusted third party (TTP) based auditing is promising to solve this dilemma. Therefore, a customized auditing scheme is desired, which is expected to keep track of accesses and operations on stored data in the cloud. The recorded information is also essential for digital forensics or disputation resolving. In this chapter, we will discuss the rationale and technologies that are potentially capable of meeting this important challenge in the storage cloud.

There are technologies to verify the retrievability of a large file F in its entirety on a remote server [1, 3, 12]. Juels and Kaliski [12] have developed Proof of Retrievability (POR), a new cryptographic building block. The POR protocol encrypts a large file and randomly embeds randomly-valued check blocks, called sentinels. To protect against corruption by the prover of a small portion of F, they also employed error-correcting codes in the POR scheme. The tradeoff of these sentinel-based schemes is that preprocessing is required before uploading the file to remote storage. Because sentinels must be indistinguishable from regular file blocks, POR can only be applied to encrypted files and has a limited number of queries that are decided prior to outsourcing.

A Provable Data Possession (PDP) scheme [1] allows a user to efficiently, frequently, and securely verify that the server possesses the original data without retrieving the entire data file and provides probabilistic guarantees of possession. The server can only access small portions of the file when generating the proof of its possession of the file. The client stores a small amount of metadata to verify the server's proof.

However, PORs and PDPs mainly focus on static, archival storage. Considering dynamic operations in which the stored data set will be updated, such as inserting, modifying, or deleting, these schemes need to be extended accordingly. Dynamic Provable Data Possession (DPDP) schemes [7] aim to verify file possession under these situations.

Due to constraints at user side such as limited computing resources, researchers also seek solutions that migrate the auditing task to a third party auditor (TPA). This approach will significantly reduce users' computing burden. However, new challenges appear. Privacy protection of users' data against external auditors becomes a major issue. Privacy-preserving public auditing has attracted a lot of attention from the cloud security research community.

The rest of the chapter is organized as follows. The basics of information auditing are introduced in Sect. 2. Section 3 discusses the principles of POR and PDP schemes and illustrates several typical implementations. Section 4 presents recent reported efforts considering privacy-preservation in cloud storage. Section 5 discusses several open questions and indicates potential research directions in the future. Finally, we conclude this chapter in Sect. 6.

2 Information Auditing: Objective and Approaches

The past decades have witnessed the rapid development of information technologies and systems. Such an evolution has made system architecture very complex. Information auditing plays the central role in effective management since it is critical to any organization to obtain a good understanding of information storage, transmission, and manipulation. As more and more components have been introduced, the focus and definition of information auditing are expanded. In this section, a definition of information auditing is given first. Then, three typical approaches are discussed that actually reflect the particular view of an auditor focusing on an organization.

2.1 Definition of Information Auditing

In past 30 years, the application of information auditing has been extended from identifying formal information sources, which emphasizes document management, to monitoring the information manipulations on the organizational level. As an independent, objective assurance and consulting activity, information auditing helps to add value and improve operations. It provides clients and service providers information for internal control, risk management, and so on. Defined by the ASLIB Knowledge & Information Management Group [6], information auditing is:

> A systematic examination of information use, resources and flows, with a verification by reference to both people and existing documents, in order to establish the extent to which they are contributing to an organization's objectives.

According to this definition, information auditing could include one or more of the following objectives:

- Identifying control requirements
- Supporting vender selection
- Reviewing vendor management
- Assessing data migration
- Assessing project management
- Reviewing/assessing/testing control flow
- Logging digital footprints for forensics

Corresponding to the objectives, the following are questions an information auditing system is expected to address:

- **Data**: What information does this system store, transfer, or manipulate?
- **Function**: How does the system work? What has done to the data?
- **Infrastructure**: Where are the system components and how are they connected?

- **User**: Who launches the work? What is the work flow model?
- **Time**: When do events happen? How are they scheduled?
- **Motivation**: Why are functions executed? What are the goals and strategies?

2.2 Three Approaches of Information Auditing

Considering the complexity of today's information systems, an IT manager may be interested in certain components of an organization instead of all components. To allow more dimensions to auditing, an auditor can adopt particular views against an organization and variant approaches can be taken. Three approaches are suggested by researchers: strategic-oriented, process-oriented, and resource-oriented. The strategic-oriented approach focuses on the routines by which an organization achieves its strategic objectives under the constraints of available information resources. The expected output in this dimension would be an information strategy for the organization. Typically a strategic-oriented information auditing system should consider the following questions:

- **Goal**: What is this system for?
- **Approaches**: How can we achieve this goal?
- **Resources**: What information/infrastructure resources do we have/use?
- **Constraints**: Is there any resource/performance gap/constraints?
- **Essential Concerns**: What are the most essential concerns?

The process-oriented approach focuses on a process, which is a sequence of activities the system takes to achieve the expected outcomes. Processes reflect system characteristics and reveal how information flows and how functions cooperate. There are four main types of processes: core processes, support processes, management processes, and business network processes [14]. The key output would include processed-based mapping and information flow/resources analysis. Typical questions [6] a process oriented information auditing system will answer include:

- **Activities**: What do we do?
- **Approaches**: How do we do it?
- **Attestation**: How can we prove we do what we say we will do?
- **Resources**: What information resources do we use and require?
- **Facilities/Tools**: What systems do we use?
- **Concerns**: What problems do we experience?

The resource-oriented approach aims at identifying, classifying, and evaluating information resources. Instead of associating resources with a strategic goal or an operational process, the major purpose of the resource-oriented approach is to allow auditors to manage or categorize resources according to strategic importance or according to their ability to support critical processes. Questions [6] that resource-oriented information auditing systems should address include:

- **Identification**: What are the information resources?
- **Utilization**: How are the information resources used?
- **Management**: How does the system manage and maintain them?
- **Policy**: What are the regulations of utility?
- **Priority**: Which are the most critical information resources? Which are useless?

Considering the properties and security expectations of cloud storage, process-oriented auditing is the most suitable candidate among the three approaches. Information collected in this dimension will provide sufficient evidence for both digital forensics and the reputation estimation of the data storage service provider. However, there is no reported effort that tries to develop such a process-oriented auditing system for cloud computing services. We hope this book chapter can inspire more activities in this important area.

3 Auditing for Data Integrity in Distributed Systems

One of the important applications of distributed storage service systems is to store large research data files that are not frequently accessed but cannot be reproduced because the devices that collected the data are unavailable or because of the expense. A client might choose to store such data in remote a storage system provided by trusted professional services. Usually the data files will be replicated in case one of the storage servers is unavailable because of maintenance or disk damage. For each server that possesses the client's data files, both the client and the service provider need to assure that each file is retrievable in its entirety whenever the client needs. It is not practical to download the entire file to verify its integrity when dealing with large archive file. Users (data owners) need to be assured that their outsourced data is not deleted or modified at the server, without having to download the entire data file.

In this section, we investigate the general categories of strategies for auditing data integrity in distributed systems. Then three popular schemes are discussed in more details.

3.1 Strategies of Auditing Data Integrity

A straightforward way to assure the integrity of our data is to utilize message authentication code (MAC). The client, who wants to use the storage service of a server, calculates a short MAC for each file block, save them in local storage and upload his data to the server. To increase the security of data, the client can choose to encrypt the data before MAC calculation. When the client needs to check the server's possession of the data, he simply asks the server to calculate MACs for all file blocks and send them as a proof of possession. The obvious problem of this simple strategy is the huge overheads of computation and communication. In order to make sure the entirety of

data, the server needs to access all file blocks, executing expense computation when the file is large. The communication cost to transmit all MACs is also unacceptable.

More practical strategies in auditing data integrity can be divide into two categories:

1. **Sentinel Embedding**. The strategy is to utilize sentinels produced by the client to secure data integrity. Sentinels are created by a one-way function. By appending the predefined number of sentinels to the encoded file and permuting the resulting file, the client is able to check fixed number of sentinels during each challenge period to the server. Since the server has no knowledge about the position of these sentinels, it cannot modify any block of the client's data without being detected in one or several challenges that could ask for the entirety of any block.
2. **Random Sampling Authenticators**. The other way to audit the integrity of data is based on authenticators. An authenticator is produced for each file block before uploading data to the server. The client only stores some metadata, such as cryptographic keys and functions, and uploads his data along with authenticators to the server. The key of this strategy is the algorithm that we use to calculate the authenticators. This algorithm should be able to verify the integrity in an aggregating way so that the proof from the server will not be proportional to the number of blocks that we want to check.

Juels and Kaliski [12] proposed the Proof of Retrievability (POR) based on sentinel embedding. Although it is a strong protocol for data integrity, there is one inevitable problem. The number of sentinels is predefined, causing a fix number of challenges. This is unacceptable in some applications. In order to obtain higher confidence of data integrity, the entire file needs to be retrieved to embed more sentinels.

Ateniese et al. [1] on the other hand suggested random sampling in their Provable Data Possession scheme. The rest schemes that we shall discuss in this chapter [17, 18, 20] are all constructed under similar idea, with various choices of authenticator algorithms for specific purposes, such as privacy preservation or dynamic data operations etc.

One problem of random sampling schemes is that they cannot assure in 100 % confidence. Ateniese et al. [1] suggested that checking 460 blocks in each challenge is able to achieve 99 % confidence to detect server misbehavior if 1 % of data is changed. This seems good enough for most applications, but still needs to be improved for more flexible storage services.

In the following subsections, we focus on POR [12], PDP [1] and Compact POR [17] for distributed storage system. Next section further discusses the challenges in Cloud storage services and efforts [18, 20] to solve them.

3.2 Proof of Retrievability

Juels and Kaliski [12] proposed a cryptographic building block known as a proof of retrievability (POR) for archived files. POR enables a user (Verifier) to determine

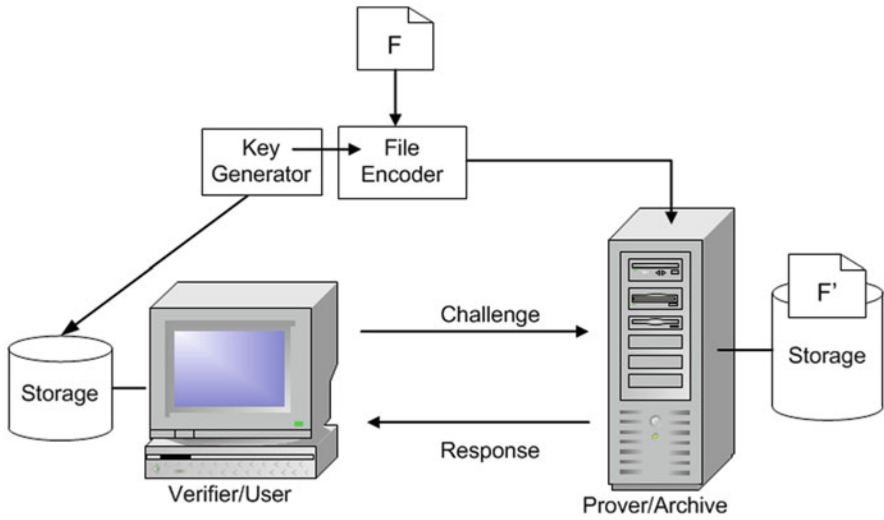

Fig. 1 Schematic of a POR System [12]

that an archive (Prover) “possesses” a file or data object F. A successfully executed POR assures a Verifier that the Prover presents a protocol interface through which the Verifier can retrieve F in its entirety.

Figure 1 shows the schematic of a POR. Two parties are involved in this model: the archive server as the Prover and the owner of the archived file or the user as the Verifier.

At the Verifier side, a key generation algorithm and an encoding algorithm are used to preprocess the file F. The key generation algorithm produces a key to encode the file F. This key should be independent of F and is stored by the Verifier. The encoding algorithm transforms raw file F into encoded file $\tilde{F}$ by randomly embedding a set of randomly-valued check blocks called sentinels.

After storing the encoded file into the Prover, the Verifier challenges the Prover by specifying the positions of a collection of sentinels and asking the Prover to return the associated sentinel values. If the Prover has modified or deleted a substantial portion of F, then with high probability it will also have suppressed a number of sentinels. It is therefore unlikely to respond correctly to the Verifier. To protect against corruption by the Prover of a small portion of F, a POR scheme also employs error-correcting codes.

A POR system (PORSYS) consists of six algorithms: **keygen**, **encode**, **extract**, **challenge**, **verify** and **respond**. Table 1 summarizes inputs, outputs of these algorithms and provides a brief description of each algorithm.

Definition 1 *Algorithm*: An algorithm with n inputs and m outputs is denoted as

$$\mathcal{A}(input_1, \cdots, input_n) \rightarrow (output1, \cdots, output_m)$$

where $\mathcal{A}$ is the name of the algorithm.

Table 1 Six Algorithms of a POR System [12]

Role	Algorithm	Description
Verifier	**keygen**$[\pi] \rightarrow \kappa$	Generate a secret key κ, could be a public/private key pair. For security concern, this key can be decomposed into multiple keys
	encode$(F; \kappa, \alpha)[\pi] \rightarrow \tilde{F}_{\eta})$	Encode the original file with κ into $\tilde{F}_{\eta}$, where η denotes the unique file id (handle) of $\tilde{F}$ in the file system
	extract$(\eta; \kappa, \alpha)[\pi] \rightarrow F$	Extract the original file F by a sequence of challenges to the Prover
	challenge$(\eta; \kappa, \alpha)[\pi] \rightarrow c$	Take as input the file handle η, secret key κ, and state α. Output a challenge value c
	verify$((r, \eta); \kappa, \alpha) \rightarrow b \in \{0, 1\}$	Determine whether the receiver response r is valid to challenge c. If success, output 1, otherwise output 0
Prover	**respond**$(c, \eta) \rightarrow r$	Generate a response to a challenge c

In these algorithms, α denotes a persistent state during a Verifier invocation, and π denotes the full collection of system parameters. π should at least include the security parameter j. In particular, we can also include the length, formatting, encoding of files and challenge/response sizes in π.

The **encode** algorithm is the core of this system since all operations and data for verification are accomplished in this algorithm. The basic steps include error correction, encryption, sentinel creation and permutation.

Figure 2 shows the file structure changes in POR system. Suppose F with a message-authentication code (MAC) has b blocks, denoted as: $F[1], \cdots, F[b]$. It is divided into s chunks, each has k l-bit blocks. Thus we can view it as an $s \times k$ matrix, where each element is a block. For simplicity, the error-correcting code (ECC) also operates over l-bit symbols and sentinels, and l-bit values computed by a one-way function have l-bit length. This basic scheme adopts an efficient (n, k, d)-error correcting code with even-valued d. This code has the ability to correct up to $d/2$ errors. After applying ECC to F, each chunk is expanded to n blocks, resulting in a new file $F' = (F'[1], F'[2], \cdots, F'[b'])$, where the number of blocks is $b' = bn/k$. The encryption step applies a symmetric-key cipher E to F', yielding file F''.

A sentinel is created by a suitable one-way function f, taking as input the key generated by **keygen** and the index of this sentinel. Suppose we have s sentinels. These sentinels are appended to F'', yielding F''' with $b' + s$ blocks. Finally, in the **encode** algorithm, we apply a permutation function to F''', obtaining the output file $\tilde{F}$, where $\tilde{F}[i] = F'''[g(\kappa, i)]$.

The auditing procedure involves **challenge**, **response** and **verify** algorithms.The Verifier use a state α to track the state of each challenge. For simplicity, we let the

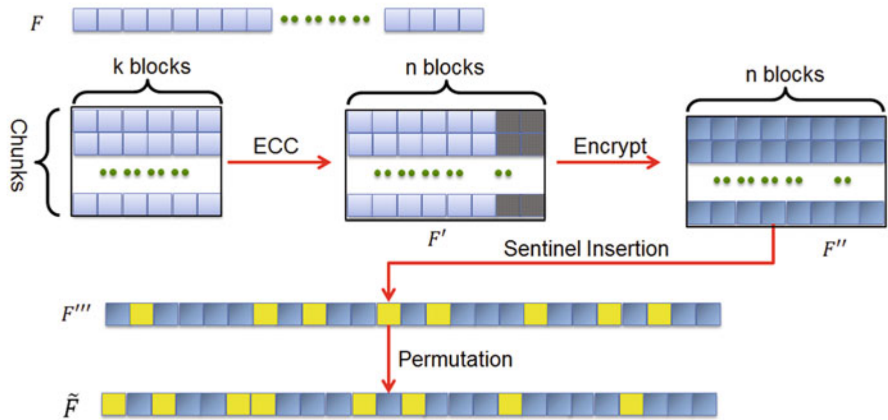

Fig. 2 File blocks changes in the encode step of sentinel POR system

Verifier state α initially be 1, incrementing it by q during each challenge[1]. The current value of α indicates that in last challenge phase the client requested sentinel position from $\alpha - q$ to $\alpha - 1$. The positions of sentinels that the Verifier wants to check are simply generated by the permeation function with two inputs: the secret key and the position of sentinel before applying permutation, that is $b' + \alpha$. The Prover then send the Verifier requested blocks (in the Prover's point of view). The **Verify** uses the one-way function f to calculate all sentinels that are being checked and compare with the Prover's response. In this way, the Verifier can detect the Prover's misbehavior in a relatively low cost of checking a small number of sentinels.

The overhead of POR mainly includes the storage for error-correcting code and sentinels, as well as computation of error-correcting code and permutation operations. Several optimization can be done to improve POR's performance. For example the length of response can be further hashed to a compact fixed length proof and the challenge can also be compressed by passing a seed to the Prover instead of all index of sentinel blocks. However, the major problem of this scheme is the limited number of challenge once the sentinel embedded file is upload to the prover.

3.3 Provable Data Possession

PDP was first proposed by Ateniese et al. [1]. Earlier solutions for verifying a server retaining a file need either expensive redundancy or access to the entire file. The PDP model provides probabilistic proof of the possession of a file with the server accessing small portions of the file when generating the proof. The client only stores fixed size

[1] In [12], the state α is not clearly defined. This interpretation of α is based on the σ in challenge function in Sect. 3.1 of [12].

of metadata and consumes a constant bandwidth. The challenge and the response are also small (168 and 148 bytes respectively). This subsection will introduce the PDP scheme in detail, including the definition and two enhanced versions of PDP algorithms: S-PDP and E-PDP.

3.3.1 Preliminaries

The PDP schemes are based on RSA algorithm. Readers are referred to [16] for more information about RSA algorithm.

First we choose two safe primes p and q that are large enough. Let $N = pq$, all exponentiations are calculated modulo N. We denoted $\mathbb{Z}_N = \{0, 1, \cdots, N-1\}$ and $\mathbb{Z}_N^*$ is the set of all numbers in $\mathbb{Z}_N$ that are relatively prime to N. That is

$$\mathbb{Z}_N^* = \{a \in \mathbb{Z}_N : gcd(a, N) = 1\}$$

Definition 2 *Quadratic Residue*: An integer $a \in \mathbb{Z}_N$ is a quadratic residue (mod N) if $x^2 \equiv a \bmod N$ has a solution. Let QR_N be the set of all quadratic residues of $\mathbb{Z}_N$ and g be a generator of QR_N.

In the RSA algorithm [16], a large integer d is randomly chosen such that it is relatively prime to $(p-1)(q-1)$. The other integer e is computed so that

$$ed \equiv 1 \mod (p-1)(q-1).$$

When using RSA-based algorithm for verification, there is a slight difference in choosing e and d in that

$$ed \equiv 1 \mod \frac{p-1}{2} \cdot \frac{q-1}{2}.$$

Definition 3 *Sets of Binary Numbers*: The set of all binary numbers with length n is denoted by $\{0, 1\}^n$. Specifically, $\{0, 1\}^*$ is the set of arbitrary length of binary numbers.

When we want to randomly choose a number k from a set S, we use the notation $k \xleftarrow{R} S$. For example, $k \xleftarrow{R} \{0, 1\}^\kappa$ means k is a number randomly chosen from the set of all κ-bit binary numbers.

Definition 4 **Homomorphic Verifiable Tags (HVTs)**: An HVT is a pair of values (T_{m_i}, W_i). W_i is a random value obtained from the index i. T_{m_i} will be store on the server.

As building blocks of PDP, HVTs have the properties of unforgeable and blockless verification. The PDP scheme use HVTs as the verification metadata of file blocks.

Finally, we introduce four cryptographic functions:

- $h : \{0, 1\}^* \rightarrow QR_N$ is a secure deterministic hash-and-encode function.
- $H : \{0, 1\}^* \rightarrow \mathbb{Z}_N$ is a cryptographic hash function.

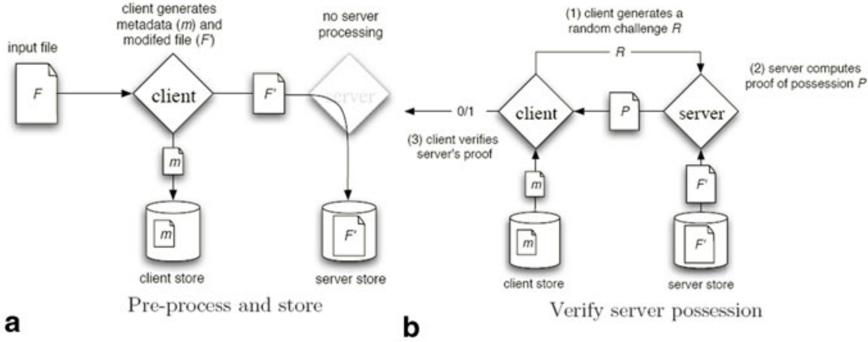

Fig. 3 Protocol for provable data possession [1]

- $f : \{0,1\}^\kappa \times \{0,1\}^{\log_2 n} \rightarrow \{0,1\}^\ell$ is a pseudo-random function (PRF). Specifically, we use $f_k(x)$ to denote $f(k,x)$.
- $\pi : \{0,1\}^\kappa \times \{0,1\}^{\log_2 n} \rightarrow \{0,1\}^{\log_2 n}$ is a pseudo-random permutation (PRP).

A hash function often takes as input an concatenation of two binary strings. We use $s_1||s_2$ to denote the concatenation of s_1 and s_2.

3.3.2 Defining the PDP Protocol

The PDP protocol involves a client, denoted as $\mathcal{C}$, who wants to store a large file in a remote server and a server, denoted as $\mathcal{S}$, who provides storage services. Fig. 3 depicts the PDP protocol in [1].

The PDP protocol consists of four polynomial-time algorithms: **KeyGen**, **TagBlcok**, **GenProof** and **CheckProof**. Table 2 summarizes these algorithms. Among them, **KeyGen**, **TagBlcok** and **CheckProof** are executed on the client side. The server need only to run the **GenProof** algorithm to generate a proof that it is possessing the client's file upon receiving a challenge from the client.

A file **F** is divided into n blocks, that is $\mathbf{F} = (m_1, \cdots, m_n)$. If not explicitly stated, the letter n always means the number of blocks in file **F**. At the beginning of the setup phase, **F** is pre-processed by the client $\mathcal{C}$ into a new file $\mathbf{F}'$. This process could include encrypting the file and generating a tag for each file block. The client then uploads $\mathbf{F}'$ to the server $\mathcal{S}$. To verify whether the server is storing the entire file, the client then periodically generates a challenge and sends it to the server. Upon receiving a challenge, the server computes a proof of possession as a response to this challenge and sends back to the client. Finally, the client can check the server's response and verifies whether the server possesses the correct file.

A PDP protocol consist of two phases: the Setup Phase and the Challenge Phase.

- **Setup Phase**:
  The Setup Phase at the client side includes generation of necessary keys (public key and private key), calculation of a tag for each file block, transmission of

Table 2 Four Algorithms of a PDP System [1]

Role	Function	Description
Client	**KeyGen**$(1^\kappa) \rightarrow (pk, sk)$	Generate a secret key pair (pk, sk), taking as input a secret parameter κ
	TagBlock$(pk, sk, m) \rightarrow T_m$	Generate the verification metadata T_m for the input file block m
	CheckProof$(pk, sk, chal, \mathcal{P})) \rightarrow \{0, 1\}$	Validate a proof of possession $\mathcal{P}$. IF $\mathcal{P}$ is a correct proof of possession, output 1, else output 0
Server	**GenProof**$(pk, \mathbf{F}, chal, \Sigma) \rightarrow \mathcal{P}$	Generate a proof of possession $\mathcal{P}$ for given challenge $chal$

the processed file to the server and finally deletion local copy of the file. These operations are all executed on the client side. In particular, this phase consists of the following steps:

1. **KeyGen**$(1^\kappa) \rightarrow (pk, sk)$ generate secret keys.
2. Apply **TagBlock** to each file block $m_i, i = 1, \cdots, n$, resulting in n tags T_{m_i}.
3. Send $\{pk, \mathbf{F}, \Sigma = (T_{m_1}, \cdots, T_{m_n})\}$ to $\mathcal{S}$.
4. Delete $\mathbf{F}$ and Σ in local storage.

- **Challenge Phase**:
  In the Challenge Phase we use a challenge-response style to verify the integrity of the client's file. The challenge message specifies a predefined number of blocks, with their indices. The server needs to prove it is possessing all these blocks by calculating a proof $\mathcal{P}$ using all block data. The necessary steps of this phase are:
  1. $\mathcal{C}$ generates a challenge **chal**, specifying the set of blocks that it wants $\mathcal{S}$ to prove that it possesses these blocks.
  2. $\mathcal{C}$ sends **chal** to $\mathcal{S}$.
  3. $\mathcal{S}$ runs **GenProof** to get the proof of possession $\mathcal{P}$
  4. $\mathcal{S}$ sends $\mathcal{P}$ to $\mathcal{C}$.

Considering the tradeoff between security and efficiency, Ateniese et al.[1] introduced a secure PDP scheme (S-PDP), which has a strong data possession guarantee, as well as an efficient PDP scheme (E-PDP), providing better efficiency by means of a weaker data possession guarantee. The next two subsections will discuss these two schemes in detail.

3.3.3 The Secure PDP Scheme (S-PDP)

This section provides the construction of the Secure PDP Scheme (S-PDP) [1], including implementation of each algorithm and the two phases. The S-PDP is able to assure that the server possesses all blocks that are specified in the challenge message.

The key generation algorithm produce the public key $pk = (N, g)$ and the secret key $sk = (e, d, v)$. The RSA modulus N is the product of two distinct large primes p and q. Let g be a generator of QR_N. The public key pk is then formed by N

and g. Among the three integers in the secret key v is randomly chosen from $\{0, 1\}^\kappa$. d is used to generate tags (authenticators) in **TagBlock** algorithm and e is used in **CheckProof** algorithm. pk and sk should be stored on the client side.

For each block of data, m_i, the **TagBlock** algorithm calculates a tag using the data as a number and its index. An index related number W_i is first generated by concatenating v with the index i, denoted as $W_i = v||i$. The tag of this block T_{m_i} is then computed as

$$T_{m_i} = (h(W_i) \cdot g^m)^d \mod N.$$

After getting all tags $\Sigma = (T_{m_1}, \cdots, T_{m_n})$, the client sends them to the server together with the original file F and the public key pk. That is $\{pk, \mathbf{F}, \Sigma\}$ are sent to the server. The client then deletes F and Σ on its local storage. This finishes the Setup Phase.

In the Challenge Phase, a challenge **chal** $= (c, k_1, k_2, g_s)$ is generated as follows. First of all, we randomly choose three integers: k_1,k_2 and s. k_1 and k_2 are selected from $\{0, 1\}^\kappa$, serving as keys for the pseudo-random permutation π and the pseudo-random function f respectively. The last number s belongs to $\mathbb{Z}_N^*$ and is used to mask the generator g. The challenge message is then formed as (c, k_1, k_2, g_s), where c is the number of blocks that each challenge will pick and $g_s = g^s \mod N$.

At the server side, there is only one algorithm **GenProof** that is executed upon receiving a challenge requested by the client. For each number j from 1 to c, an index $i_j = \pi_{k_1}(j)$ and a mask $a_j = f_{k_2}(j)$ are calculated. The file blocks that the client want to check are then indicated by $\{i_1, i_2, \cdots, i_c\}$. Finally, two numbers T and ρ are computed as the proof for this challenge:

$$T = \prod_{j=1}^{c} T_{m_{i_j}}^{a_j}, \rho = H(g_s^{a_1 m_{i_1} + \cdots + a_c m_{i_c}} \mod N).$$

The motivation for putting the coefficients a_j in the challenge phase is to strengthen the guarantee that $\mathcal{S}$ possesses each block queried by the client. In each challenge phase, there is a randomly chosen key for calculation of these coefficients. $\mathcal{S}$ cannot store combinations of the original blocks to save storage cost. Since the proof of possession has a constant length regardless the number of blocks being requested, this scheme can maintain constant communication cost in the challenge phase.

Once T and ρ are ready, the server response the client with its proof to **chal**, $\mathcal{P} = (T, \rho)$. The client runs **GenProof** to check the correctness of the proof. First, it computes i_j, a_j and W_{i_j} as the server did. Then $\tau = T^e$ is divided by $h(W_{i_j})^{a_j}$ for each j from 1 to c. This actually results in

$$\tau = g^{a_1 m_{i_1} + \cdots + a_c m_{i_c}} \mod N.$$

If the proof is valid, the following equation should be true:

$$H(\tau^s \mod N) = \rho.$$

Table 3 Comparison between S-PDP and E-PDP [1]

Algorithm	S-PDP	E-PDP
GenProof	$a_j = f_{k_2}(j)$	delete
	$T = \left(\prod_{j=1}^{c} T_{i_j}^{a_j}\right) \bmod N$	$T = \prod_{j=1}^{c} T_{i_j} \bmod N$
	$\rho = H(g_s^{a_1 m_{i_1} + \cdots + a_c m_{i_c}} \bmod N)$	$\rho = H(g_s^{m_{i_1} + \cdots + m_{i_c}} \bmod N)$
CheckProof	$i_j = \pi_{k_1}(j), W_{i_j} = v \Vert i_j, a_j = f_{k_2}(j)$	$i_j = \pi_{k_1}(j), W_{i_j} = v \Vert i_j$
	$\tau = (\tau / h(W_{i_j})^{a_j}) \bmod N$	$\tau = [T^e/(h(w_{i_1}) \cdots h(w_{i_c}))] \bmod \text{N}$

3.3.4 The Efficient PDP Scheme (E-PDP)

The Efficient Provable Data Possession (E-PDP) [1] scheme achieved a higher performance at the cost of a weaker guarantee by eliminating all coefficients a_j in the GenProof and CheckProof algorithms. Table 3 shows a comparison between S-PDP and E-PDP. As all coefficients $a_j = 1$, the E-PDP scheme reduces the expensive exponential computation. However, the server $\mathcal{S}$ can only possess the sum of the blocks $m_{i_1}, \cdots, m_{i_c}$ for a challenge. In order to completely pass the challenge phase every time, the server $\mathcal{S}$ needs to compute every combination of c blocks out of n blocks, that is $\binom{n}{c}$. The client can choose values of n and c such that make it impractical for the server to simply store all sums.

3.4 Compact Proof of Retrievability

Because of the predefined number of sentinels, the sentinel-based POR scheme has a limited number of possible challenges. Based on Juels and Kaliski's work [12], Shacham and Waters [17] introduced two new schemes that achieved public and private verifiability, with compete proof of security.

Shacham and Waters [17] proposed two new proof of retrievability schemes, with private and public verifiability respectively. The main advantage of SW PORs is the unlimited number of queries. The private verification scheme, based on pseudo-random functions (PRFs), has the shortest response of any POR scheme (20 bytes) with the cost of a longer query. The second scheme used short signatures introduced by Boneh, Lynn and Shacham (BLS) [5] to verify the authentication of data in remote servers, hence assuring public verifiability is secure. At an 80-bit security level, this scheme has the shortest query (20 bytes) and response (40 bytes)of any POR scheme.

An important contribution of [17] is that it provided a complete security proof for both schemes. Interested readers can consult this paper for more details.

3.4.1 System Model

Shacham and Waters's system model has similar functions with Juels and Kaliski's POR description in [12], but modules were redefined and more details were added. In

this model, key generation and verification procedures no longer maintain any state. In addition, Shacham and Waters's protocols [17] allow challenge and response to be arbitrary.

There are two parties in this system, the Verifier and the Prover. The Verifier could be the data owner itself or a third party auditor. The prover is the storage server. Similar to POR, a file M with size b is divided into n blocks, each further split into s sectors. Thus, we can refer to a sector as m_{ij}, $1 \leq i \leq n$, $1 \leq j \leq s$. M can also be treated as an $n \times s$ matrix $\{m_{ij}\}$. Each sector is an element of $\mathbb{Z}_p = \{0, 1, \cdots, p-1\}$, where p is a large enough prime number.

Our description of these algorithms in the following sections about public and private verification will be slightly different with [12] in input and output parameters. We redefine part of these algorithms to maintain internal consistency. We hereby ignore all input and output parameters since they vary in the public scheme and the private scheme and focus on the functionalities of these algorithms. There are four algorithms in the system model:

- **KeyGen**. The key generation algorithm. The Verifier runs this algorithm to generate necessary private keys (for private verification) or key pairs (for public verification) for other algorithms.
- **Store**. The file processing algorithm. The Verifier runs this algorithm to produce the processed file M^* and a file tag t. The processed file M^* is stored in the server. The file tag t is saved in the data owner's local storage or at the server side depends on the desire of service agreement.
- **Prove**. The proving algorithm. The Prover runs this algorithm to generate a proof of retrievability according the index indicated in the challenge message sent by the Verifier.
- **Verify**. The verifying algorithm. If it is a third party auditor who is running this algorithm, the file tag need to be retrieved and verified first. Then the challenge message is produced. When the prover sends the response message back, the Verify algorithm continues to verify the response. After running this algorithm, the Verifier will know whether the file is being stored on the server and can be retrieved as needed. If the algorithm fails, it outputs 0. The Verifier can then use an extractor algorithm to attempt to recover the file.

This scheme still works in two phases. In the setup phase the data owner runs **KeyGen**and **Store** to process the file and upload the resulting file to the server (Prover). The challenge phase involves the Verifier (data owner or TPA), running **Verify**, and the Prover, running **Prove**.

3.4.2 Private Verification Construction

The construction of a private verification consists of the implementation of the four functions that are discussed in Sect. 3.4.1.

Since this scheme is for private verification purposes, we only need a secret key $sk = (k_{enc}, k_{mac})$ in the **KeyGen** algorithm, where k_{enc} is an encryption key and k_{mac} is an MAC key.

In the **Store** algorithm, the file M is preprocessed with an erasure code before applying the following operations. This erasure code should be able to recover the file even the Prover erases some portion of the file. The processed file is denoted as M^* as a part of output. This file is divided into n blocks, each has s sections. Hence we can write $M^* = \{m_{ij}\}, (i = 1, \cdots, n; j = 1, \cdots, s)$. To detect any modification to the file by the Prover, the Verifier (data owner) calculate an authenticators

$$\sigma_i = f_{k_{prf}}(i) + \sum_{j=1}^{s} \alpha_j m_{ij}, i = 1, \cdots, n$$

for each block, where α_j are randomly chosen from $\mathbb{Z}_p$. These authenticators provide strong assurance that the Prover cannot forge any one of them since it has no knowledge of the PRF as well as the PRF key. A file tag t is also computed to include PRF key and α_j. These numbers are concatenated and encrypted first. Then the encrpyted bit string is appended to the number of blocks n, forming an initial tag t_0. Finally, the file tag is produced by appending the MAC of t_0, keyed with k_{mac}, with itself. The Verifier only stores sk and outsources the processed file M^*, authenticators $\{\sigma_i\}_{i=1}^{n}$ and the file tag t to the Prover.

In the **Verify** algorithm, the Verifier sends an l-element query $Q = \{(i, v_i)\}$, $1 \leq i \leq n$, specifying the index of blocks that are to be verified, to the Prover. In each pair of (i, v_i), i is a random index of file block and v_i is randomly chosen from B, a subset of $\mathbb{Z}_p$. For simplicity, we can let $B = \mathbb{Z}_p$. There are totally l pairs in Q, suggesting that the set of all i, $I = \{i : (i, v_i) \in Q\}$, has the size l.

In the **Prove** algorithm, the Prover uses v_i as coefficients when calculating a proof for a specific Q. This also prevents the Prover from using a previously calculated proof. The response $r = \{\mu_1, \cdots, \mu_s, \sigma\}$ is produced as follows. For each $1 \leq j \leq s$, we let $\mu_j = \sum_{i \in I} v_i m_{ij}$. The last number σ is simply the sum of all products of v_i and σ_i, that is $\sigma = \sum_{i \in I} v_i \sigma_i$. Now the Prover have all it needs for the response r and send it to the Verifier.

Back to the **Verify** algorithm, upon receiving r, the Verifier checks if

$$\sigma = \sum_{i \in I} v_i f_{k_{prf}}(i) + \sum_{j=1}^{s} \alpha_j \mu_j$$

is true. If it is, there is a high probability that M is retrievable.

As shown in the protocol, this private verification scheme has less computation overhead than PDP since multiplication is the only operation except addition.

3.4.3 Public Verification Construction

A public verifiable POR scheme allows anyone who has the public key of the data owner to query the Prover and verify the return response. With this protocol, user

offloads the verification task to a trusted third party auditor. The public verification scheme in [17] used BLS signatures [5] for authentication values instead of utilizing PRF.

In this public verification construction, Shacham and Waters employ bilinear map in the verify algorithm V. We briefly introduce bilinear map here. Interested readers are referred to [4, 5] for more details. Let G_1, G_2 and G_T be multiplicative cyclic groups of the same prime order p. g_1 is a generator of G_1 and g_2 is a generator of G_2. A bilinear map is a map $e : G_1 \times G_2 \rightarrow G_T$ with the following properties:

1. Bilinear: for all $u, v \in G$ and $a, b \in \mathbb{Z}$, $e(u^a, v^b) = e(u, v)^{ab}$.
2. Non-degenerate: $e(g_1, g_2) \neq 1$

In this scheme, we let $G = G_1 = G_2$.

The **KeyGen** algorithm now needs a public and private key pair. At first a signing key pair (spk, ssk) is generated. The public key $pk = (v, spk)$ and the secret key $sk = (\alpha, ssk)$, where α is randomly chosen from $\mathbb{Z}_p$ and $v = g^{\alpha}$.

The file tag t contains a $name \in \mathbb{Z}_p$ and s randomly chosen $u_k \in G, k = 1, \cdots, s$ as well as the number of blocks n. These data are concatenated, resulting $t_0 = name||n||u_1|| \cdots ||u_s$, and appended by its signature keyed with ssk. The final file tag is $t = t_0||SSig_{ssk}(t_0)$. u_k is also used for authenticator calculation. For each i from 1 to n, the authenticator of block m_i is $\sigma_i = (H(name||i) \cdot \prod_{j=1}^{s} u_j^{m_{ij}})^{\alpha}$, where $H : \{0, 1\}^* \rightarrow G$ be the BLS hash [5]. After calculating the above information, the user then sends the erasure coded file M^* together with $\{\sigma_i\}$ to the Prover.

The challenge phase is almost the same as the private verification scheme except that σ in the response message is changed to the sum of $\sigma_i^{v_i}$ instead of $v_i \sigma_i$ before. Of course, the verification equation need to be modified since bilinear group is used here. The user now check whether the following condition is held:

$$e(\sigma, g) = e\Big(\prod_{(i,v_i) \in Q} H(name||i)^{v_i} \cdot \prod_{j=1}^{s} u_j^{\mu_j}, v \Big).$$

4 Auditing in Cloud Storage Platform

Cloud computing is migrating traditional computing services to remote cloud service providers. Cloud storage has advantages such as high flexibility, ultimately low price and relatively high data security for a wide spectrum of users. However, not all problems with traditional distributed storage are solved by cloud computing. Security is still the major concern, even though the cloud providers all claim they can protect our data in more secure way than the users can do themselves.

This section analyzes changes brought by cloud computing in data storage and introduces researchers' attempt to solve these problems.

4.1 Challenges

Although there are reported efforts in information auditing for distributed systems, the special features in the cloud storage platforms necessitate customized design due to new challenges. This subsection briefly lists the problems that need to be considered.

1. **Dynamic Data Operations**. Clients in cloud services might not have files of large size such as the original PDP and PoR schemes assume, but the number of files is greater, and the flexibility requirements are stronger. Files in the cloud storage will be changed more frequently. Modification, deletion and insertion need to be considered in the design of storage system.
2. **Public Verifiability**. Computing devices that cloud clients have might not be powerful enough to accomplish the computational task of integrity auditing of their own data in cloud storage. Meanwhile, these clients' end devices might have multiple tasks to do, which cannot allow limited computing resources consumed by this single task. It is desired to offload the verification procedure to a third party auditor (TPA), which has sufficient computing resources and expertise in data auditing. It is expected to make the verification protocol a public verifiable one.
3. **Privacy Preserving**. This seems to conflict with the public verifiability requirement at first glance. How can the TPA execute auditing protocol and yet not be trusted? Studies in preserving privacy in using TPAs for auditing purposes showed that it is feasible and practical to design a verification protocol for untrusted TPAs. Using this protocol, file blocks should not be retrieved in order to verify the integrity of files.
4. **Computational Efficiency**. A cloud client can be a portable device like PDA or smartphone which usually has weaker computation ability and limited communication bandwidth. Data auditing protocols in cloud storage should try to reduce both computation and bandwidth as much as possible.
5. **Multiple Files**. A cloud client's storage request could consist of large number of files instead of a single large file.
6. **Batch auditing**. A cloud server can be audited by a TPA for thousands of users' files. In this case, if aggregating multiple proofs as a single message to the TPA is applicable, the communication burden of the protocol could be significantly reduced to an acceptable level.

In the following subsections, we shall discuss more schemes trying to tackle some of these problems.

4.2 Public Verifiability

There is a variant of PDP scheme [1] that can support public verifiability. A PDP scheme with public verifiability property allows anyone to challenge the server for the possession of the specific file as long as they have the client's public key.

To support public verifiability, the following changes are made to the S-PDP protocol:

1. Besides N and g, the Client should make e public.
2. A PRF $\omega : \{0,1\}^{\kappa} \times \{0,1\}^{\log_2 n} \rightarrow \{0,1\}^{\ell}$ is used to generate W_i by randomly choose a v from $\{0,1\}^{\kappa}$ as a key. That is, $W_i = \omega_v(i)$.
3. The client makes v public after the Setup phase.
4. The challenge **chal** in GenProof and CheckProof no longer contains g_s or s.
5. In GenProof, the server computes $M = a_1 m_{i_1} + \cdots + a_c m_{i_c}$ instead of ρ and returns $\mathcal{V} = (T, M)$.
6. In CheckProof, the client checks $g^M = \tau$ and $|M| < \lambda/2$.

In sect. 3.4, we also saw a public verifiable POR scheme.

4.3 Dynamic Data Operations Support

The PDP scheme [1] did not employ dynamic data operations like modification, deletion and insertion due to the original motivation to verify integrity of archive files, which will not involve many dynamic operations. Similarly, the POR scheme [12] cannot support data dynamics due to the verification mechanism of embedding pre-computed sentinels. However, these operations are vital features for cloud storage services.

Ateniese et al. [2] propose a dynamic version of PDP scheme . The extended scheme achieved higher efficiency because it only relied on symmetry-key cryptography. But the number of queries was limited, hence, the scheme cannot support fully dynamic data operations. Erway et al. [7] introduced a formal framework for dynamic provable data possession (DPDP) . Their first scheme utilized authenticated skip list data structure to authenticate tag information of blocks, thereby eliminating the index information in tags. They also provided an alternative RSA tree based construction, which improved the detection probability at the cost of an increased Server computation burden.

Wang et al.[20] extended the Compact POR in Sect. 3.4 to support both public verifiability and data dynamics in cloud storage. We'll focus on this model to discuss dynamic data operation support. Table 4 shows the six algorithms in [20].

In cloud data storage, Clients could be portable devices that have limited computation ability. A third party auditor is necessary for the verification procedure. The system we shall consider in this section includes three entities: Client, Cloud Storage Server and the prover, a Third Part Auditor (TPA). The TPA is trusted and unbiased while the Server is untrusted. Privacy preserving is not considered in [20].

Table 4 Algorithms of Extended POR System [20]

Role	Algorithm	Description
Client	**KeyGen**$(1^\kappa) \rightarrow (pk, sk)$	This algorithm is the same as in SW's model in sect. 3.4. It generates a secret key pair (pk, sk), taking as input a secret parameter κ
	SigGen$(sk, F) \rightarrow (\Phi, sig_{sk}(H(R)))$	Generates the signature set $\Phi = \{\sigma_i\}$ on file blocks $\{m_i\}$ and sign the root R of a Merkel hash tree $sig_{sk}(H(R))$
	VerifyUpdate$(pk, update, \mathcal{P}_{update}) \rightarrow \{(1, sig_{sk}(H(R'))), 0\}$	Verifies the update operation
Client/TPA	**VerifyProof**$(pk, chal, \mathcal{P}) \rightarrow \{0, 1\}$	Validate a proof $\mathcal{P}$. IF $\mathcal{P}$ is correct, output 1, else output 0
Server	**GenProof**$(pk, \mathbf{F}, \mathbf{chal}, \Sigma) \rightarrow \mathcal{P}$	Generates a proof $\mathcal{P}$ for given challenge **chal**
	ExecUpdate$(F, \Phi, update) \rightarrow (F', \Phi', \mathcal{P}_{update})$	According to the type of "update" request from the Client, this algorithm executes the corresponding update operation and outputs the updated file F', signatures Φ' and proof $\mathcal{P}$

The Client encodes the raw file $\tilde{F}$ into **F** using Reed-Solomon codes. The file **F** consists of n blocks $m_1, \cdots, m_n$, $m_i \in \mathbb{Z}_p$, where p is a large prime. $e : G \times G \rightarrow G_T$ is a bilinear map, with a hash function $H : \{0, 1\}^* \rightarrow G$ serving as a random oracle. g is the the generator of G. h is a cryptographic hash function.

In order to accomplish dynamic dada operation, the well studied Merkle hash tree (MHT) [13] is a good choice to assure the value and positions of data blocks. A MHT is a binary tree with all data blocks as leaf nodes. A parent node is the hash of the concatenation of its two children. This procedure continues until reach a common root node. All dynamic operations will result in a update of MHT by recalculating every node that is in the path from affected blocks to the root. The sibling data that is needed for a recalculation is called auxiliary information.

The verification of data relies on BLS hash [5] and bilinear map as in [17]. The differences here reside in file tag, authenticators calculation, components of a proof and the verification of a proof. The file tag is shorter in that it only include the concatenated name, number of blocks and a random value for authentication purpose. A proof includes four parts: a block data related value, a authenticator related value, the auxiliary information set and the signature of the MHT root's BLS hash. When computing an authenticator, the **SigGen** algorithm no longer take into account the index or name as in [1] or the public scheme of [17]. It is simply $\sigma_i = (H(m_i) \cdot u^{m_i})^\alpha$, where u is a random value and α is a part of the secret key. The bilinear map e is used twice in **VerifyProof**, one to authenticate MHT root and the other to verify the rest of the proof.

We now consider three types of dynamic data operations: Modification, Insertion and Deletion. The advantage of MHT lies in the convenience in modifying the structure of the tree, hence embedding dynamic data operations into the scheme.

- Modification. The Client wants to replace a block block. First, it computes the signature for new block and a update message is sent to the Server. The Server runs **ExecUpdate** to update the block. The update procedure includes replacement of new block, new authenticator and the leaf node in MHT. The root is then updated. A proof of update message must be sent to the Client so that he can know whether the update is valid. This message includes all auxiliary information, the old signature of the hash of the root and the new root. The Client generate old root using auxiliary information and uses bilinear map to check whether the signature is valid. If it is true, new MHT root is computed and compared with the one transmitted back by the Server. The modification is valid if and only if it passes all these tests.
- Insertion. The Client wants to insert block m^* after block m_i. It generates signature σ^* and sends the Server a update message. The Server runs **ExecUpdate** to execute a insertion operation, storing new block, inserting new authenticator, generating new root, and sends a update message to the Client like it did in the modification operation. The Client also need to verify this operation according to the message it receive.
- Deletion. Inverse operation to insertion. Similar to modification and insertion.

This scheme efficiently solves the dynamic operation problem but has the drawback that the messages exchanged between the Client and the Server is proportional to the number of file blocks.

4.4 Privacy Preserving

Although auditing storage data through a third party auditor, who has expertise in auditing and powerful computing capabilities, has many advantages to the client, the auditing procedure has the possibility to reveal user data to the TPA. Previous schemes [1, 17, 20] for data verification do not consider the privacy protection issue when offloading the verification job to the TPA. They all assumed that the TPA is trusted and will not try to look into user's data when verifying the integrity of data. A privacy-preserving public auditing scheme was proposed for cloud storage in [19]. Based on a homomorphic linear authenticator, integrated with random masking, the proposed scheme is able to preserve data privacy when TPA audits the stored data in the server.

There are three entities in this system: The user, the Cloud Server and the TPA. Since dynamic data operations were not considered in this scheme, only four algorithms (**KeyGen**, **SigGen**, **GenProof** and **VerifyProof**) are needed in this protocol, without the two algorithms for update purpose in Sect. 4.3. Still, we have two phases in the system: Setup Phase and Audit Phase. The mathematical integrity assurance technique is still a bilinear map $e : G_1 \times G_2 \rightarrow G_T$ as in Sect. 3.4.3. These groups

should be different groups but has the same order. The server has knowledge about G_1, G_T and $\mathbb{Z}_p$ (all file blocks are elements of this group).

Like all other public key cryptosystem, the **KeyGen** algorithm needs a public-private key pair. A pair of signing keys (spk, ssk) is generated for the verification of file tag, which includes the identifier of the file.

The secrete key sk includes the secret signing key ssk and a random integer chosen from $\mathbb{Z}_p$. The public key $pk = (spk, g, g^x, u, e(u, v))$ on the other hand includes more values. g is a generator of G_2, u is an element of G_1 and $e(u, v) \in G_T$ is the image of u and v under the bilinear map e.

The **SigGen** algorithm calculates the file tag and authenticators in a different way. The file tag in [19] is shorter than previous schemes [17, 20], only the identifier of the file is included. This identifier, denoted as $name$, is also an element of $\mathbb{Z}_p$. The signing key pair is generated just for verification of $name$. An authenticator the block m_i is $\sigma_i = (H(name||i) \cdot u^{m_i})^x$. The hash function $H : \{0, 1\}^* \rightarrow G_1$ maps a bit string into G_1, which means all authenticators will fall into G_1. The set of the authenticators and the file tag are sent to the server. This finishes the setup phase.

During the audit phase, the file tag is retrieved and verified by the TPA. If t is valid, the file name is recovered. The challenge **chal** is generated in the same way as in [17]. Upon receiving **chal**, the server runs **GenProof** to calculate the proof that it possesses the requested file blocks. There are three components in the response to **chal**: μ, σ, and R. σ is the aggregation of all authenticators that are indicated by **chal**. Each authenticator σ_i is raised to the power v_i and their product is the value of σ. The other two values are related to a random number r from $\mathbb{Z}_p$. R is the result of raising the image of u, v to the power r. A number μ' is directly calculated from all indicated blocks. This value is highly related to the file. To hide it from the TPA, the server uses r and the hash value of R. The final component $\mu = r + h(R)\mu'$ is obtained. The TPA runs **VerifyProof** to validate the response. If $R \cdot e(\sigma^\gamma g) = e((\prod_{i \in I} H(W_i)^{v_i})^\gamma \cdot u^\mu, v)$ is true, the response is a valid one. The audit procedure is then accomplished.

Data dynamic operations can also be supported by adapting this scheme using MHT as in [20].

4.5 Multiple Verifications

Since the cloud server is accessed by multiple users, the possibility that many clients request verification for different files or one client requests verifying multiple files. These requests should be treated in different way and hence need different auditing schemes. For example, multiple clients have different key pairs, whereas one single client requesting multiple verifications has the same key pair. Most schemes that claim to be able to support batch auditing belong to the first category.

Both [19] and [20] have the extension to support multiple verifications thanks to the aggregation property of bilinear signature schemes [4]. [20] uses auxiliary information in a proof, hence has relatively long proof message for multiple clients

batch auditing. Only σ in each proof can be aggregated in one value. [19] aggregates by multiply all R's.

Batch auditing can reduce the computation cost on TPA since the K responses are aggregated into one. But the practical efficiency still needs to be verified by further experiments.

5 Open Questions

In this chapter, we provided an overview of general issues on information auditing to clarify the major goal of information auditing. Then we discussed two popular protocols that audits for data integrity in distributed data storage: PDP and POR. They were proposed almost at the same time to address different security concerns. PDP provides a high probability guarantee that a system possesses a file with high efficiency in computation and communication. POR and Compact POR allow a stronger guarantee of retrievability with the cost of more complex algorithms. Most schemes discussed in this chapter came with security proof in the original research papers, in which interested readers can find proof details and mathematical analysis. However, there is not sufficient study on efficiency and performance.

Since it is still a new research area in Cloud storage, we anticipate more new schemas will come out in the academic community, trying to resolve different challenges from various perspective. When evaluating a scheme, it usually includes the following metrics:

- Server computation overhead for a proof in each storage node.
- Server communication overhead when transmit computing results to form the final proof.
- Client computation overhead for authenticators, error-correcting code and verification algorithm.
- Communication cost between any two parties of the client, the server and TPA.
- Client storage for necessary metadata.
- Server misbehavior detection probability.

Compared to traditional information auditing, there are still numerous open problems in cloud data security auditing. The number one impending issue is the lack of standardization and consistency in auditing development efforts due to the heterogenity in infrastructure, platforms, software, and policy. While a "silver bullet" is highly desired, the diversity in auditing and assurance practices in cloud computing makes it extremely challenging to find a one-for-all solution. Essentially, in terms of data security oriented auditing, a thorough study is expected on balancing the tradeoffs among confidentiality, integrity, availability and usability.

From the cloud service providers' point of view, allowing external auditing implies more components such as transparency, responsibility, assurance, and remediation [15]. To accommodate these central components, a cloud service provider is required to:

- Set up policies that are consistent with external auditing criteria.
- Provide transparency to clients/users.
- Allow external auditing.
- Support remediation, such as accident management and compliant handling.
- Enable legal mechanisms that support prospective and retrospective accountability.

6 Conclusions

An efficient auditing system is critical to establish accountability for cloud users who do not have physical possession of their data. Existance of a trustworthy third party audits enable users to check the data integrity, track suspicious activities, obtain evidence for forensics, and evaluate service providers' behaviors when needed. This chapter provides our readers fundamental understanding of cloud auditing technologies. We expect to witness development of standard framework for cloud auditing and efforts at cloud service providers to make their policies and mechanisms more auditable and accountable.

References

1. Ateniese, G., Burns, R., Curtmola, R., Herring, J., Kissner, L., Peterson, Z., Song, D.: Provable data possession at untrusted stores. In: Proceedings of the 14th ACM conference on Computer and communications security, CCS '07, pp. 598–609. ACM, New York, NY, USA (2007). DOI 10.1145/1315245.1315318. URL http://doi.acm.org/10.1145/1315245.1315318
2. Ateniese, G., Di Pietro, R., Mancini, L.V., Tsudik, G.: Scalable and efficient provable data possession. In: Proceedings of the 4th international conference on Security and privacy in communication networks, SecureComm '08, pp. 9:1–9:10. ACM, New York, NY, USA (2008). DOI 10.1145/1460877.1460889. URL http://doi.acm.org/10.1145/1460877.1460889
3. Ateniese, G., Kamara, S., Katz, J.: Proofs of Storage from Homomorphic Identification Protocols. In: M. Matsui (ed.) Advances in Cryptology - ASIACRYPT 2009, *Lecture Notes in Computer Science*, vol. 5912, chap. 19, pp. 319–333. Springer Berlin / Heidelberg, Berlin, Heidelberg (2009). DOI 10.1007/978-3-642-10366-7_19. URL http://dx.doi.org/10.1007/978-3-642-10366-7_19
4. Boneh, D., Gentry, C., Lynn, B., Shacham, H.: Aggregate and verifiably encrypted signatures from bilinear maps. Advances in Cryptology-EUROCRYPT 2003 pp. 641–641 (2003)
5. Boneh, D., Lynn, B., Shacham, H.: Short signatures from the weil pairing. Journal of Cryptology **17**, 297–319 (2004). URL http://dx.doi.org/10.1007/s00145-004-0314-9. 10.1007/s00145-004-0314–9
6. Buchanan, S., Gibb, F.: The information audit: Role and scope. International journal of information management **27**(3), 159–172 (2007)
7. Erway, C., Küpçü, A., Papamanthou, C., Tamassia, R.: Dynamic provable data possession. In: Proceedings of the 16th ACM conference on Computer and communications security, CCS '09, pp. 213–222. ACM, New York, NY, USA (2009). DOI 10.1145/1653662.1653688. URL http://doi.acm.org/10.1145/1653662.1653688
8. Feng, J., Chen, Y.: A fair non–repudiation framework for data integrity in cloud storage services. International Journal of Cloud Computing **2**(1), 20–47 (2013)

9. Feng, J., Chen, Y., Liu, P.: Bridging the missing link of cloud data storage security in aws. In: Consumer Communications and Networking Conference (CCNC), 2010 7th IEEE, pp. 1–2. IEEE (2010)
10. Feng, J., Chen, Y., Summerville, D., Ku, W.S., Su, Z.: Enhancing cloud storage security against roll-back attacks with a new fair multi-party non-repudiation protocol. In: Consumer Communications and Networking Conference (CCNC), 2011 IEEE, pp. 521–522. IEEE (2011)
11. Feng, J., Chen, Y., Summerville, D.H.: A fair multi-party non-repudiation scheme for storage clouds. In: Collaboration Technologies and Systems (CTS), 2011 International Conference on, pp. 457–465. IEEE (2011)
12. Juels, A., Kaliski Jr., B.S.: Pors: proofs of retrievability for large files. In: Proceedings of the 14th ACM conference on Computer and communications security, CCS ’07, pp. 584–597. ACM, New York, NY, USA (2007). DOI 10.1145/1315245.1315317. URL http://doi.acm.org/10.1145/1315245.1315317
13. Merkle, R.: Protocols for public key cryptosystems. In: IEEE Symposium on Security and privacy, vol. 1109, pp. 122–134 (1980)
14. Ould, M.A.: Business Processes: Modeling and Analysis for Re-engineering and Improvement. Wiley, Chichester (1995)
15. Pearson, S.: Toward accountability in the cloud. Internet Computing, IEEE **15**(4), 64 –69 (2011). DOI 10.1109/MIC.2011.98
16. Rivest, R.L., Shamir, A., Adleman, L.: A method for obtaining digital signatures and public-key cryptosystems. Commun. ACM **21**(2), 120–126 (1978). DOI 10.1145/359340.359342. URL http://doi.acm.org/10.1145/359340.359342
17. Shacham, H., Waters, B.: Compact Proofs of Retrievability Advances in Cryptology - ASIACRYPT 2008. In: J. Pieprzyk (ed.) Advances in Cryptology - ASIACRYPT 2008, *Lecture Notes in Computer Science*, vol. 5350, chap. 7, pp. 90–107. Springer Berlin / Heidelberg, Berlin, Heidelberg (2008). DOI 10.1007/978-3-540-89255-7_7. URL http://dx.doi.org/10.1007/978-3-540-89255-7_7
18. Wang, C., Chow, S., Wang, Q., Ren, K., Lou, W.: Privacy-preserving public auditing for secure cloud storage. Computers, IEEE Transactions on **PP**(99), 1 (2011). DOI 10.1109/TC.2011.245
19. Wang, C., Wang, Q., Ren, K., Lou, W.: Privacy-preserving public auditing for data storage security in cloud computing. In: INFOCOM, 2010 Proceedings IEEE, pp. 1–9 (2010). DOI 10.1109/INFCOM.2010.5462173
20. Wang, Q., Wang, C., Ren, K., Lou, W., Li, J.: Enabling public auditability and data dynamics for storage security in cloud computing. Parallel and Distributed Systems, IEEE Transactions on **22**(5), 847 –859 (2011). DOI 10.1109/TPDS.2010.183