Enabling Public Verifiability and Data Dynamics for Storage Security in Cloud Computing

Qian Wang[1], Cong Wang[1], Jin Li[1], Kui Ren[1], and Wenjing Lou[2]

[1] Illinois Institute of Technology, Chicago IL 60616, USA,
{qwang,cwang,jin.li,kren}@ece.iit.edu
[2] Worcester Polytechnic Institute, Worcester MA 01609, USA,
{wjlou}@ece.wpi.edu

Abstract. Cloud Computing has been envisioned as the next-generation architecture of IT Enterprise. It moves the application software and databases to the centralized large data centers, where the management of the data and services may not be fully trustworthy. This unique paradigm brings about many new security challenges, which have not been well understood. This work studies the problem of ensuring the integrity of data storage in Cloud Computing. In particular, we consider the task of allowing a third party auditor (TPA), on behalf of the cloud client, to verify the integrity of the dynamic data stored in the cloud. The introduction of TPA eliminates the involvement of client through the auditing of whether his data stored in the cloud is indeed intact, which can be important in achieving economies of scale for Cloud Computing. The support for data dynamics via the most general forms of data operation, such as block modification, insertion and deletion, is also a significant step toward practicality, since services in Cloud Computing are not limited to archive or backup data only. While prior works on ensuring remote data integrity often lacks the support of either public verifiability or dynamic data operations, this paper achieves both. We first identify the difficulties and potential security problems of direct extensions with fully dynamic data updates from prior works and then show how to construct an elegant verification scheme for seamless integration of these two salient features in our protocol design. In particular, to achieve efficient data dynamics, we improve the Proof of Retrievability model [1] by manipulating the classic Merkle Hash Tree (MHT) construction for block tag authentication. Extensive security and performance analysis show that the proposed scheme is highly efficient and provably secure.

1 Introduction

Several trends are opening up the era of Cloud Computing, which is an Internet-based development and use of computer technology. The ever cheaper and more powerful processors, together with the "software as a service" (SaaS) computing architecture, are transforming data centers into pools of computing service on a huge scale. Meanwhile, the increasing network bandwidth and reliable yet flexible

network connections make it even possible that clients can now subscribe high quality services from data and software that reside solely on remote data centers.

Although envisioned as a promising service platform for the Internet, this new data storage paradigm in "Cloud" brings about many challenging design issues which have profound influence on the security and performance of the overall system. One of the biggest concerns with cloud data storage is that of data integrity verification at untrusted servers. For example, the storage service provider, which experiences Byzantine failures occasionally, may decide to hide the data errors from the clients for the benefit of their own. What is more serious is that for saving money and storage space the service provider might neglect to keep or deliberately delete rarely accessed data files which belong to an ordinary client. Consider the large size of the outsourced electronic data and the client's constrained resource capability, the core of the problem can be generalized as how can the client find an efficient way to perform periodical integrity verifications without the local copy of data files.

In order to solve this problem, many schemes are proposed under different systems and security models [1–10]. In all these works, great efforts are made to design solutions that meet various requirements: high scheme efficiency, stateless verification, unbounded use of queries and retrievability of data, etc. Considering the role of the verifier in the model, all the schemes presented before fall into two categories: private verifiability and public verifiability. Although schemes with private verifiability can achieve higher scheme efficiency, public verifiability allows anyone, not just the client (data owner), to challenge the cloud server for correctness of data storage while keeping no private information. Then, clients are able to delegate the evaluation of the service performance to an independent third party auditor (TPA), without devotion of their computation resources. In the cloud, the clients themselves are unreliable or cannot afford the overhead of performing frequent integrity checks. Thus, for practical use, it seems more rational to equip the verification protocol with public verifiability, which is expected to play a more important role in achieving economies of scale for Cloud Computing. That is, the outsourced data themselves should not be required by the verifier for the verification purpose. In the context of public verification, the importance of blocklessness goes even further because an TPA should not be allowed to possess the original data files for the obvious security concern.

Another major concern among previous designs is that of supporting dynamic data operation for cloud data storage applications. In Cloud Computing, the remotely stored electronic data might not only be accessed but also updated by the clients, e.g., through block modification, deletion and insertion. Unfortunately, the state-of-the-art in the context of remote data storage mainly focus on static data files and the importance of this dynamic data updates has received limited attention in the data possession applications so far [1–4, 6, 9, 11, 12]. Moreover, as will be shown later, the direct extension of the current provable data possession (PDP) [2] or proof of retrievability (PoR) [1, 3] schemes to support data dynamics may lead to security loopholes. Although there are many difficulties faced by researchers, it is well believed that supporting dynamic data operation

can be of vital importance to the practical application of storage outsourcing services. In view of the key role of public verifiability and the supporting of data dynamics for cloud data storage, in this paper we present a framework and an efficient construction for seamless integration of these two components in our protocol design. Our contribution can be summarized as follows: (1) We propose a general formal PoR model with public verifiability for cloud data storage, in which both blockless and stateless verification are achieved simultaneously; (2) We equip the proposed PoR construction with the function of supporting for fully dynamic data operations, especially to support block insertion, which is missing in most existing schemes; (3) We prove the security of our proposed construction and justify the performance of our scheme through concrete implementation and comparisons with the state-of-the-art.

1.1 Related Work

Recently, much of growing interest has been pursued in the context of remotely stored data verification [1–9, 11, 13–15]. Ateniese *et al.* [2] define the "provable data possession" (PDP) model for ensuring possession of files on untrusted storages. In their scheme, they utilize RSA-based homomorphic tags for auditing outsourced data, thus can provide public verifiability. However, Ateniese *et al.* do not consider the case of dynamic data storage, and the direct extension of their scheme from static data storage to dynamic case brings many design and security problems. In their subsequent work [11], Ateniese et al. propose a dynamic version of the prior PDP scheme. However, the system imposes a priori bound on the number of queries and does not support fully dynamic data operations, i.e., it only allows very basic block operations with limited functionality and block insertions cannot be supported. In [13], Wang et al. consider dynamic data storage in distributed scenario, and the proposed challenge-response protocol can both determine the data correctness and locate possible errors. Similar to [11], they only consider partial support for dynamic data operation. Juels *et al.* [3] describe a "proof of retrievability" (PoR) model and give a more rigorous proof of their scheme. In this model, spot-checking and error-correcting codes are used to ensure both "possession" and "retrievability" of data files on archive service systems. Specifically, some special blocks called "sentinels" are randomly embedded into the data file F for detection purpose and F is further encrypted to protect the positions of these special blocks. However, like [11], the number of queries a client can perform is also a fixed priori and the introduction of pre-computed "sentinels" prevents the development of realizing dynamic data updates. In addition, public verifiability is not supported in their scheme. Shacham *et al.* [1] design an improved PoR scheme with full proofs of security in the security model defined in [3]. Like the construction in [2], they use publicly verifiable homomorphic authenticators built from BLS signatures [16] and provably secure in the random oracle model. Based on the BLS construction, public retrievability is achieved and the proofs can be aggregated into a small authenticator value. Still the authors only consider static data files. Erway *et*

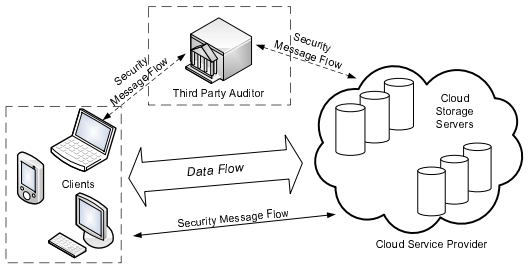

Fig. 1: Cloud data storage architecture

al. [14] was the first to explore constructions for dynamic provable data possession. They extend the PDP model in [2] to support provable updates to stored data files using rank-based authenticated skip lists. This scheme is essentially a fully dynamic version of the PDP solution. In particular, to support updates, especially for block insertion, they try to eliminate the index information in the "tag" computation in Ateniese's PDP model [2]. To achieve this, before the verification procedure, they employ authenticated skip list data structure to authenticate the tag information of challenged or updated blocks first. However, the efficiency of their scheme remains in question. It can be seen that while existing schemes are proposed to aiming at providing integrity verification under different data storage systems, the problem of supporting both public verifiability and data dynamics has not been fully addressed. How to achieve a secure and efficient design to seamlessly integrate these two important components for data storage service remains an open challenging task in cloud computing.

Organization. The rest of the paper is organized as follows. In section 2, we define the system model, security model and our goal. Then, we present our scheme in section 3 and provide security analysis in section 4. We further analyze the experiment results and show the practicality of our schemes in section 5. Finally, we conclude in section 6.

2 Problem Statement

2.1 System Model

A representative network architecture for cloud data storage is illustrated in Fig. 1. Three different network entities can be identified as follows: *Client*: an entity, which has large data files to be stored in the cloud and relies on the cloud for data maintenance and computation, can be either individual consumers or organizations; *Cloud Storage Server* (CSS): an entity, which is managed by Cloud Service Provider (CSP), has significant storage space and computation resource to maintain clients' data; *Third Party Auditor* (TPA): a TPA, which has expertise and capabilities that clients do not have, is trusted to assess and expose risk of cloud storage services on behalf of the clients upon request.

In the cloud paradigm, by putting the large data files on the remote servers, the clients can be relieved of the burden of storage and computation. As clients no longer possess their data locally, it is of critical importance for the clients to ensure that their data are being correctly stored and maintained. That is, clients should be equipped with certain security means so that they can periodically verify the correctness of the remote data even without the existence of local copies. In case that clients do not necessarily have the time, feasibility or resources to monitor their data, they can delegate the monitoring task to a trusted TPA. To protect client data privacy, audits are performed without revealing original data files to TPA. In this paper, we only consider verification schemes with public verifiability: any TPA in possession of the public key can act as a verifier. We assume that TPA is unbiased while the server is untrusted. Note that we don't address the issue of data privacy in this paper, as the topic of data privacy in Cloud Computing is orthogonal to the problem we study here. For application purposes, the clients may interact with the cloud servers via CSP to access or retrieve their pre-stored data. More importantly, in practical scenarios the client may frequently perform block-level operations on the data files. The most general forms of these operations we consider in this paper are modification, insertion, and deletion.

2.2 Security Model

Shacham and Waters propose a security model for PoR system in [1]. Generally, the checking scheme is secure if (i) there exists no polynomial-time algorithm that can cheat the verifier with non-negligible probability; (ii) there exists a polynomial-time extractor that can recover the original data files by carrying out multiple challenges-responses. Under the definition of this PoR system, the client can periodically challenge the storage server to ensure the correctness of the cloud data and the original files can be recovered by interacting with the server. The authors in [1] also define the correctness and soundness of PoR scheme: the scheme is correct if the verification algorithm accepts when interacting with the valid prover (e.g., the server returns a valid response) and it is sound if any cheating server that convinces the client it is storing the data file is actually storing that file. Note that in the "game" between the adversary and the client, the adversary has full access to the information stored in the server, i.e., the adversary can play the part of the prover (server). In the verification process, the adversary's goal is to cheat the client successfully, i.e., trying to generate valid responses and pass the data verification without being detected.

Our security model has subtle but crucial difference from that of the original PoRs in the verification process. Note that the original PoR schemes [1,3,4,15] do not consider dynamic data operations and the block insert cannot be supported at all. This is because the construction of the signatures is involved with the file index information i. Thus, once a file block is inserted, the computation overhead is unacceptable since the signatures of all the following file blocks should be recomputed with the new indexes. To deal with this limitation, we remove the index information i in generating the signatures and use $H(m_i)$ as the tag for block

m_i (see section 3.3) instead of $H(name||i)$ [1] or $h(v||i)$ [3], so individual data operation on any file block will not affect the others. Recall that $H(name||i)$ or $h(v||i)$ should be generated by the client in the verification process [1, 2]. However, in our new construction the client without the data information has no capability to calculate $H(m_i)$. In order to successfully perform the verification while achieving blockless, the server should take over the job of computing $H(m_i)$ and then return it to the prover. The consequence of this variance will lead to a serious problem: it will give the adversary more opportunities to cheat the prover by manipulating $H(m_i)$ or m_i. Due to this construction, our security model differs from that of the original PoR in both the verification and the data updating process. Specifically, in our scheme tags should be authenticated in each protocol execution other than calculated or pre-stored by the verifier (The details will be shown in section 3). Note that we will use server and prover (or client, TPA and verifier) interchangeably in this paper.

2.3 Design Goals

Our design goals can be summarized as the following: (1) Public verification for storage correctness assurance: to allow anyone, not just the clients who originally stored the file on cloud servers, to have the capability to verify the correctness of the stored data on demand; (2) Dynamic data operation support: to allow the clients to perform block-level operations on the data files while maintaining the same level of data correctness assurance. The design should be as efficient as possible so as to ensure the seamless integration of public verifiability and dynamic data operation support; (3) Blockless verification: no challenged file blocks should be retrieved by the verifier (e.g., TPA) during verification process for both efficiency and security concerns. (4) Stateless verification: to eliminate the need for state information maintenance at the verifier side between audits throughout the long term of data storage.

3 The Proposed Scheme

3.1 Notation and Preliminaries

Bilinear Map. A bilinear map is a map $e : G \times G \rightarrow G_T$, where G is a Gap Diffie-Hellman (GDH) group and G_T is another multiplicative cyclic group of prime order p with the following properties [16]: (i) Computable: there exists an efficiently computable algorithm for computing e; (ii) Bilinear: for all $h_1, h_2 \in G$ and $a, b \in \mathbb{Z}_p$, $e(h_1^a, h_2^b) = e(h_1, h_2)^{ab}$; (iii) Non-degenerate: $e(g, g) \neq 1$, where g is a generator of G.

Merkle Hash Tree. A Merkle Hash Tree (MHT) is a well-studied authentication structure [17], which is intended to efficiently and securely prove that a set of elements are undamaged and unaltered. It is constructed as a binary tree where the leaves in the MHT are the hashes of authentic data values. While MHT is commonly used to authenticate the values of data blocks, However, in

this paper we further employ MHT to authenticate both the values and the positions of data blocks. We treat the leaf nodes as the left-to-right sequence, so any leaf node can be uniquely determined by following this sequence and the way of computing the root in MHT.

3.2 Definition

$(pk, sk) \leftarrow KeyGen(1^k)$. *This probabilistic algorithm is run by the client. It takes as input security parameter* 1^k*, and returns public key* pk *and private key* sk*.*

$(\Phi, sig_{sk}(H(R))) \leftarrow SigGen(sk, F)$. *This algorithm is run by the client. It takes as input private key* sk *and a file* F *which is an ordered collection of blocks* $\{m_i\}$*, and outputs the signature set* Φ*, which is an ordered collection of signatures* $\{\sigma_i\}$ *on* $\{m_i\}$*. It also outputs metadata-the signature* $sig_{sk}(H(R))$ *of the root* R *of a Merkle hash tree. In our construction, the leaf nodes of the Merkle hash tree are hashes of* $H(m_i)$*.*

$(P) \leftarrow GenProof(F, \Phi, chal)$. *This algorithm is run by the server. It takes as input a file* F*, its signatures* Φ*, and a challenge* $chal$*. It outputs a data integrity proof* P *for the blocks specified by* $chal$*.*

$\{TRUE, FALSE\} \leftarrow VerifyProof(pk, chal, P)$. *This algorithm can be run by either the client or the third party auditor upon receipt of the proof* P*. It takes as input the public key* pk*, the challenge* $chal$*, and the proof* P *returned from the server, and outputs* $TRUE$ *if the integrity of the file is verified as correct, or* $FALSE$ *otherwise.*

$(F', \Phi', P_{update}) \leftarrow ExecUpdate(F, \Phi, update)$. *This algorithm is run by the server. It takes as input a file* F*, its signatures* Φ*, and a data operation request "update" from client. It outputs an updated file* F'*, updated signatures* Φ' *and a proof* P_{update} *for the operation.*

$\{(TRUE, sig_{sk}(H(R'))), FALSE\} \leftarrow VerifyUpdate(pk, update, P_{update})$. *This algorithm is run by the client. It takes as input public key* pk*, the signature* $sig_{sk}(H(R))$*, an operation request "update", and the proof* P_{update} *from server. If the verification successes, it outputs a signature* $sig_{sk}(H(R'))$ *for the new root* R'*, or* $FALSE$ *otherwise.*

3.3 Our Construction

Given the above discussion, in our construction, we use BLS signature [16] as a basis to design the system with data dynamics support. As will be shown, the schemes designed under BLS construction can also be implemented in RSA construction. In the discussion of section 3.4, we will show that direct extensions of previous work [1,2] have security problems and we believe that protocol design for supporting dynamic data operation is a major challenging task for cloud storage systems.

Now we start to present the main idea behind our scheme. As in the previous PoR systems [1,3], we assume the client encodes the raw data file $\widetilde{F}$ into F using

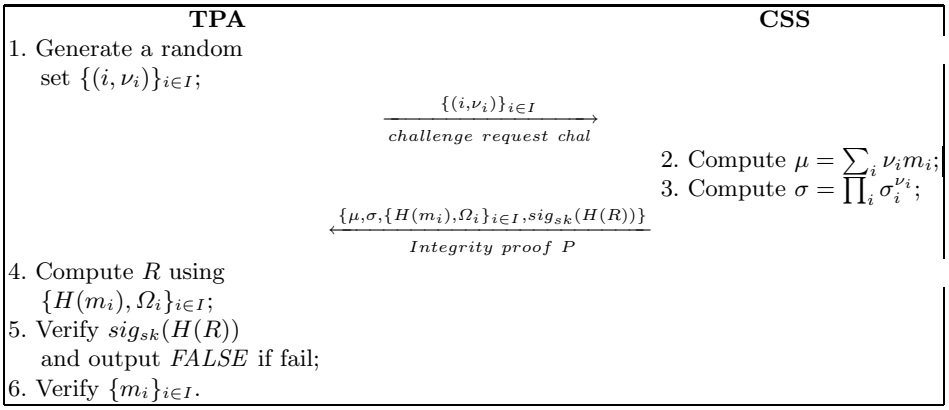

Fig. 2: Protocols for Default Integrity Verification

Reed-Solomon codes and divides the encoded file F into n blocks $m_1, \ldots, m_n$[3], where $m_i \in \mathbb{Z}_p$ and p is a large prime. Let $e : G \times G \to G_T$ be a bilinear map, with a hash function $H : \{0,1\}^* \to G$, viewed as a random oracle [1]. Let g be the generator of G. h is a cryptographic hash function. The procedure of our protocol execution is as follows:

■ **Setup**: The client's public key and private key are generated by invoking $KeyGen(\cdot)$. By running $SigGen(\cdot)$, the raw data file F is pre-processed and the homomorphic authenticators together with metadata are produced.

$KeyGen(1^k)$. The client chooses a random $\alpha \leftarrow \mathbb{Z}_p$ and computes $v \leftarrow g^\alpha$. The secret key is $sk = (\alpha)$ and the public key is $pk = (v)$.

$SigGen(sk, F)$. Given $F = (m_1, \ldots, m_n)$, the client chooses a random element $u \leftarrow G$ and computes signature σ_i for each block m_i $(i = 1, \ldots, n)$ as $\sigma_i \leftarrow (H(m_i) \cdot u^{m_i})^\alpha$. Denote the set of signatures by $\Phi = \{\sigma_i\}$, $1 \le i \le n$. The client then generates a root R based on the construction of Merkle Hash Tree (MHT), where the leave nodes of the tree are an ordered set of BLS hashes of "file tags" $H(m_i)$ $(i = 1, \ldots, n)$. Next, the client signs the root R under the private key α: $sig_{sk}(H(R)) \leftarrow (H(R))^\alpha$. The client sends $\{F, \Phi, sig_{sk}(H(R))\}$ to the server and deletes them from its local storage.

■ **Default Integrity Verification**: The client or the third party, e.g., TPA, can verify the integrity of the outsourced data by challenging the server. To generate the message "*chal*", the TPA (verifier) picks a random c-element subset $I = \{s_1, \ldots, s_c\}$ of set $[1, n]$, where we assume $s_1 \le \cdots \le s_c$. For each $i \in I$ the TPA chooses a random element $\nu_i \leftarrow \mathbb{Z}_p$. The message "*chal*" specifies the positions of the blocks to be checked in this challenge phase. The verifier sends the *chal* $\{(i, \nu_i)\}_{s_1 \le i \le s_c}$ to the prover (server).

$GenProof(F, \Phi, chal)$. Upon receiving the challenge $chal = \{(i, \nu_i)\}_{s_1 \le i \le s_c}$, the

[3] We assume these blocks are distinct with each other.

server computes

$$\mu = \sum_{i=s_1}^{s_c} \nu_i m_i \in \mathbb{Z}_p \quad \text{and} \quad \sigma = \prod_{i=s_1}^{s_c} \sigma_i^{\nu_i} \in G.$$

In addition, the prover will also provide the verifier with a small amount of auxiliary information $\{\Omega_i\}_{s_1 \le i \le s_c}$, which are the node siblings on the path from the leaves $\{h(H(m_i))\}_{s_1 \le i \le s_c}$ to the root R of the MHT. The prover responds the verifier with proof $P = \{\mu, \sigma, \{H(m_i), \Omega_i\}_{s_1 \le i \le s_c}, sig_{sk}(H(R))\}$.

$VerifyProof(pk, chal, P)$. Upon receiving the responses from the prover, the verifier generates root R using $\{H(m_i), \Omega_i\}_{s_1 \le i \le s_c}$ and authenticates it by checking $e(sig_{sk}(H(R)), g) \stackrel{?}{=} e(H(R), g^{\alpha})$. If the authentication fails, the verifier rejects by emitting *FALSE*. Otherwise, the verifier checks

$$e(\sigma, g) \stackrel{?}{=} e(\prod_{i=s_1}^{s_c} H(m_i)^{\nu_i} \cdot u^{\mu}, v).$$

If so, output $TRUE$; otherwise $FALSE$. The protocol is illustrated in Fig. 2.

■ **Dynamic Data Operation with Integrity Assurance**: Now we show how our scheme can explicitly and efficiently handle fully dynamic data operations including data modification ($\mathcal{M}$), data insertion ($\mathcal{I}$) and data deletion ($\mathcal{D}$) for cloud data storage. Note that in the following descriptions for the protocol design of dynamic operation, we assume that the file F and the signature Φ have already been generated and properly stored at server. The root metadata R has been signed by the client and stored at the cloud server, so that anyone who has the client's public key can challenge the correctness of data storage.

-***Data Modification***: We start from data modification, which is one of the most frequently used operations in cloud data storage. A basic data modification operation refers to the replacement of specified blocks with new ones.

Suppose the client wants to modify the i-th block m_i to m_i'. The protocol procedures are described in Fig. 3. At start, based on the new block m_i', the client generates the corresponding signature $\sigma_i' = (H(m_i') \cdot u^{m_i'})^{\alpha}$. Then, he constructs an *update request* message "$update = (\mathcal{M}, i, m_i', \sigma_i')$" and sends to the server, where $\mathcal{M}$ denotes the modification operation. Upon receiving the request, the server runs $ExecUpdate(F, \Phi, update)$. Specifically, the server (i) replaces the block m_i with m_i' and outputs F'; (ii) replaces the σ_i with σ_i' and outputs Φ'; (iii) replaces $H(m_i)$ with $H(m_i')$ in the Merkle hash tree construction and generates the new root R' (see the example in Fig. 4). Finally, the server responses the client with a proof for this operation, $P_{update} = (\Omega_i, H(m_i), sig_{sk}(H(R)), R')$, where Ω_i is the AAI for authentication of m_i. After receiving the proof for modification operation from server, the client first generates root R using $\{\Omega_i, H(m_i)\}$ and authenticates the AAI or R by checking $e(sig_{sk}(H(R)), g) \stackrel{?}{=} e(H(R), g^{\alpha})$. If it is not true, output *FALSE*, otherwise the client can now check whether the server has performed the modification as required or not, by further computing the new root value using $\{\Omega_i, H(m_i')\}$ and comparing it with R'. If it is not true,

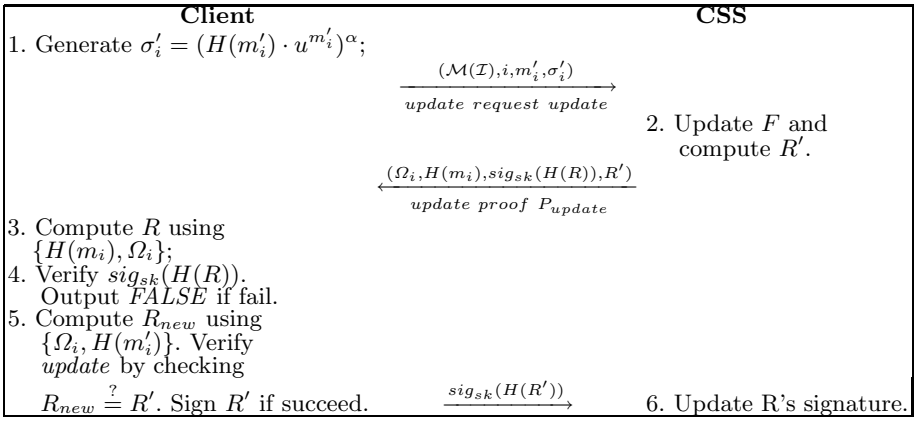

Fig. 3: The protocol for provable data update (Modification and Insertion)

output *FALSE*, otherwise output *TRUE*. Then, the client signs the new root metadata R' by $sig_{sk}(H(R'))$ and sends it to the server for update.

-***Data Insertion***: Compared to data modification, which does not change the logic structure of client's data file, another general form of data operation, data insertion, refers to inserting new blocks after some specified positions in the data file F.

Suppose the client wants to insert block m^* after the i-th block m_i. The protocol procedures are similar to the data modification case (see Fig. 3, now m_i' can be seen as m^*). At start, based on m^* the client generates the corresponding signature $\sigma^* = (H(m^*) \cdot u^{m^*})^\alpha$. Then, he constructs an *update request* message "$update = (\mathcal{I}, i, m^*, \sigma^*)$" and sends to the server, where $\mathcal{I}$ denotes the insertion operation. Upon receiving the request, the server runs $ExecUpdate(F, \Phi, update)$. Specifically, the server (i) stores m^* and adds a leaf $h(H(m^*))$ "after" leaf $h(H(m_i))$ in the Merkle hash tree and outputs F'; (ii) adds the σ^* into the signature set and outputs Φ'; (iii) generates the new root R' based on the updated Merkle hash tree. Finally, the server responses the client with a proof for this operation, $P_{update} = (\Omega_i, H(m_i), sig_{sk}(H(R)), R')$, where Ω_i is the AAI for authentication of m_i in the old tree. An example of block insertion is illustrated in Fig. 5, to insert $h(H(m^*))$ after leaf node $h(H(m_2))$, only node $h(H(m^*))$ and an internal node C is added to the original tree, where $h_c = h(h(H(m_2))||h(H(m^*)))$. After receiving the proof for insert operation from server, the client first generates root R using $\{\Omega_i, H(m_i)\}$ and authenticates the AAI or R by checking if $e(sig_{sk}(H(R)), g) = e(H(R), g^\alpha)$. If it is not true, output *FALSE*, otherwise the client can now check whether the server has performed the insertion as required or not, by further computing the new root value using $\{\Omega_i, H(m_i), H(m^*)\}$ and comparing it with R'. If it is not true, output *FALSE*, otherwise output *TRUE*. Then, the client signs the new root metadata R' by $sig_{sk}(H(R'))$ and sends it to the server for update.

-***Data Deletion***: Data deletion is just the opposite operation of data insertion.

For single block deletion, it refers to deleting the specified block and moving all the latter blocks one block forward. Suppose the server receives the *update* request for deleting block m_i, it will delete m_i from its storage space, delete the leaf node $h(H(m_i))$ in the MHT and generate the new root metadata R' (see the example in Fig. 6). The details of the protocol procedures are similar to that of data modification and insertion, which are thus omitted here.

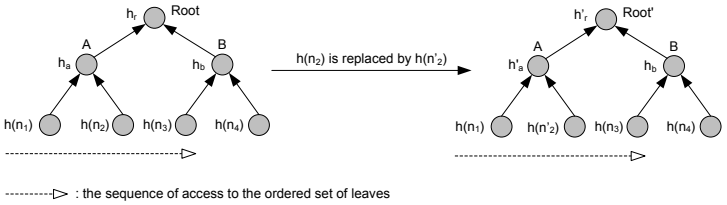

Fig. 4: Example of MHT update under block modification operation. Here, n_i and n_i' are used to denote $H(m_i)$ and $H(m_i')$, respectively.

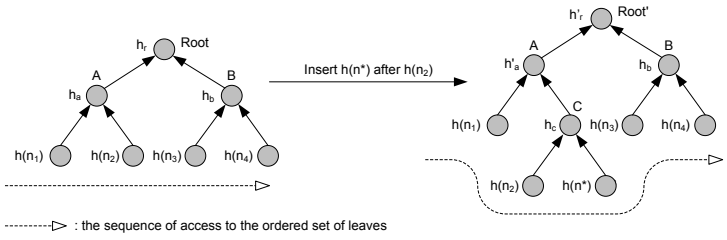

Fig. 5: Example of MHT update under block insertion operation. Here, n_i and n^* are used to denote $H(m_i)$ and $H(m^*)$, respectively.

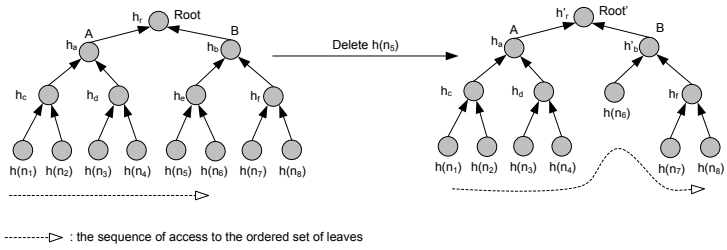

Fig. 6: Example of MHT update under block deletion operation.

3.4 Discussion on Design Considerations

Instantiations based on BLS and RSA. As discussed above, we present a BLS-based construction that offers both public verifiability and data dynamics. In fact, our proposed scheme can also be constructed based on RSA signatures. Compared with RSA construction [2, 14], as a desirable benefit, the BLS construction can offer shorter homomorphic signatures (e.g., 160 bits) than those that use RSA techniques (e.g., 1024 bits). In addition, the BLS construction has the shortest query and response (we does not consider AAI here): 20 bytes and 40 bytes [1]. However, while BLS construction is not suitable to use variable sized blocks (e.g., for security parameter $\lambda = 80$, $m_i \in \mathbb{Z}_p$, where p is a 160-bit prime), the RSA construction can support variable sized blocks. The reason is that in RSA construction the order of QR_N is unknown to the server, so it is impossible to find distinct m_1 and m_2 such that $g^{m_1} \bmod N = g^{m_2} \bmod N$ according to the factoring assumption. But the block size cannot increase without limit, as the verification block $\mu = \sum_{i=s_1}^{s_c} \nu_i m_i$ grows linearly with the block size. Recall that $h(H(m_i))$ are used as the MHT leaves, upon receiving the challenge the server can calculate these tags on-the-fly or pre-store them for fast proof computation. In fact, one can directly use $h(g^{m_i})$ as the MHT leaves instead of $h(H(m_i))$. In this way at the verifier side the job of computing the aggregated signature σ should be accomplished after authentication of g^{m_i}. Now the computation of aggregated signature σ is eliminated at the server side, as a trade-off, additional computation overhead may be introduced at the verifier side.

Support for Data Dynamics. The direct extension of PDP or PoR schemes to support data dynamics may have security problems. We take PoR for example, the scenario in PDP is similar. When m_i is required to be updated, $\sigma_i = [H(name||i)u^{m_i}]^x$ should be updated correspondingly. Moreover, $H(name||i)$ should also be updated, otherwise by dividing σ_i by σ_i', the adversary can obtain $[u^{\Delta m_i}]^x$ and use this information and Δm_i to update any block and its corresponding signature for arbitrary times while keeping σ consistent with μ. This attack cannot be avoided unless $H(name||i)$ is changed for each update operation. Also, because the index information is included in computation of the signature, an insertion operation at any position in F will cause the updating of all following signatures. To eliminate the attack mentioned above and make the insertion efficient, as we have shown, we use $H(m_i)$ instead of $H(name||i)$ as the block tags, and the problem of supporting fully dynamic data operation is remedied in our construction. Note that different from the public information $name||i$, m_i is no longer known to client after the outsourcing of original data files. Since the client or TPA cannot compute $H(m_i)$, this job has to be assigned to the server (prover). However, by leveraging the advantage of computing $H(m_i)$, the prover can cheat the verifier through the manipulation of $H(m_i)$ and m_i. For example, suppose the prover wants to check the integrity of m_1 and m_2 at one time. Upon receiving the challenge, the prover can just compute the pair (σ, μ) using arbitrary combinations of two blocks in the file. Now the response formulated in this way can successfully pass the integrity check. So, to prevent this attack, we should first authenticate the tag information before verification,

i.e., ensuring these tags are corresponding to the blocks to be checked.

Designs for Blockless and Stateless Verification. The naive way of realizing data integrity verification is to make the hashes of the original data blocks as the leaves in MHT, so the data integrity verification can be conducted without tag authentication and signature aggregation steps. However, this construction requires the server to return all the challenged blocks for authentication, and thus is not efficient for verification purpose. Moreover, due to concern for security in the context of public verification, the original data files should not be revealed to TPA during verification process. To overcome these deficiencies, most existing works in remote data checking adopt a blockless strategy for data integrity verification. For the same reason, this paper adopts the blockless approach, and we authenticate the block tags instead of original data blocks in the verification process. As we have described, in the *setup* phase the verifier signs the metadata R and stores it on the server to achieve stateless verification. Making the scheme fully stateless may cause the server to cheat: the server can revert the update operation and keep only old data and its corresponding signatures after completing data updates. Since the signatures and the data are consistent, the client or TPA may not be able to check whether the data is up to date. Actually, one can easily defend this attack by storing the root R on the verifier, i.e., R can be seen as public information. However, this makes the verifier not fully stateless in some sense since TPA will store this information for the rest of time.

4 Security Analysis

Definition 1. (**CDH Assumption**) *The Computational Diffie-Hellman assumption is that, given g, $g^x, g^y \in G$ for unknown $x, y \in \mathbb{Z}_p$, it is hard to compute g^{xy}.*

Theorem 1. *If the signature scheme is existentially unforgeable and the computational Diffie-Hellman problem is hard in bilinear groups, no adversary against the soundness of our public-verification scheme could cause verifier to accept in a proof-of-retrievability protocol instance with non-negligible probability, except by responding with correctly computed values.*

Theorem 2. *Suppose a cheating prover on an n-block file F is well-behaved in the sense above, and that it is ϵ-admissible. Let $\omega = 1/\sharp B + (\rho n)^{\ell}/(n - c + 1)^c$. Then, provided that $\epsilon - \omega$ is positive and non-negligible, it is possible to recover a ρ-fraction of the encoded file blocks in $O(n/(\epsilon - \rho))$ interactions with cheating prover and in $O(n^2 + (1 + \epsilon n^2)(n)/(\epsilon - \omega))$ time overall.*

Theorem 3. *Given a fraction of the n blocks of an encoded file F, it is possible to recover the entire original file F with all but negligible probability.*

Due to space limitations, the detailed proofs of Theorems 1, 2 and 3 are provided in the full version [18].

5 Performance Analysis

We list the features of our proposed scheme in Table 1 and make a comparison of our scheme and state-of-the-art. The scheme in [14] extends the original PDP [2] to support data dynamics using authenticated skip list. Thus, we call it DPDP scheme thereafter. For the sake of completeness, we implemented both our BLS and RSA-based instantiations as well as the state-of-the-art scheme [14] in Linux. Our experiment is conducted using C on a system with an Intel Core 2 processor running at 2.4 GHz, 768 MB RAM, and a 7200 RPM Western Digital 250 GB Serial ATA drive with an 8 MB buffer. Algorithms (pairing, SHA1 etc.) are implemented using the Pairing-Based Cryptography (PBC) library version 0.4.18 and the crypto library of OpenSSL version 0.9.8h. To achieve 80-bit security parameter, the curve group we work on has a 160-bit group order and the size of modulus N is 1024 bits. All results are the averages of 10 trials. Table 2 lists the performance metrics for 1 GB file under various erasure code rate ρ while maintaining high detection probability (99%) of file corruption. In our schemes, rate ρ denotes that any ρ-fraction of the blocks suffices for file recovery as proved in Theorem 3, while in [14], rate ρ denotes the tolerance of file corruption. According to [2], if t fraction of the file is corrupted, by asking proof for a constant c blocks of the file, the verifier can detect this server misbehavior with probability $p = 1 - (1-t)^c$. Let $t = 1 - \rho$ and we get the variant of this relationship $p = 1 - \rho^c$. Under this setting, we quantify the extra cost introduced by the support of dynamic data in our scheme into server computation, verifier computation as well as communication overhead.

From table 2, it can be observed that the overall performance of the three schemes are comparable to each other. Due to the smaller block size (i.e., 20bytes), our BLS-based instantiation is more than 2 times faster than the other two in terms of server computation time. However, its has larger computation cost at the verifier side as the paring operation in BLS scheme consumes more time than RSA techniques. Note that the communication cost of DPDP scheme is the largest among the three in practice. This is because there are 4-tuple values associated with each skip list node for one proof, which results in extra communication cost as compared to our constructions. The communication overhead (server's response to the challenge) of our RSA-based instantiation and DPDP scheme [14] under different block sizes is illustrated in Fig. 7. We can see that the communication cost grows almost linearly as the block size increases, this is mainly caused by the increasing in size of the verification block $\mu = \sum_{i=s_1}^{s_c} \nu_i m_i$. However, at very small block sizes (less than 20KB), both schemes can achieve an optimal point that minimizes the total communication cost.

6 Conclusion

To ensure cloud data storage security, it is critical to enable a third party auditor (TPA) to evaluate the service quality from an objective and independent perspective. Public verifiability also allows clients to delegate the integrity verification tasks to TPA while they themselves can be unreliable or not be able

Scheme / Metric	[2]	[1]	[11]*	[14]	Our Scheme
Data dynamics	**No**		**Yes**		
Public verifiability	**Yes**	**Yes**	**No**	**No**†	**Yes**
Sever comp. complexity	$O(1)$	$O(1)$	$O(1)$	$O(\log n)$	$O(\log n)$
Verifier comp. complexity	$O(1)$	$O(1)$	$O(1)$	$O(\log n)$	$O(\log n)$
Comm. complexity	$O(1)$	$O(1)$	$O(1)$	$O(\log n)$	$O(\log n)$
Verifier storage complexity	$O(1)$	$O(1)$	$O(1)$	$O(1)$	$O(1)$

Table 1: Comparisons of different remote data integrity checking schemes. The security parameter λ is eliminated in the costs estimation for simplicity. $*$ The scheme only supports bounded number of integrity challenges and partially data updates, i.e., data insertion is not supported. † No explicit implementation of public verifiability is given for this scheme.

	Our BLS-based instantiation		Our RSA-based instantiation		[14]
Metric \ Rate-ρ	99%	97%	99%	97%	99%
Sever comp. time (ms)	6.52	2.29	13.42	4.76	13.80
Verifier comp. time (ms)	1154.39	503.88	794.27	208.28	807.90
Comm. cost (KB)	243	80	223	76	280

Table 2: Performance comparison under different tolerance rate ρ of file corruption for 1GB file. The block size for RSA-based instantiation and scheme in [14] is chosen to be 4KB.

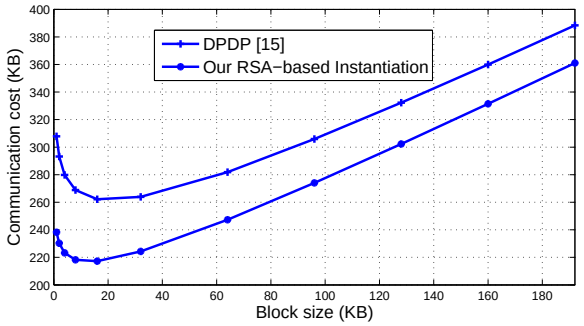

Fig. 7: Comparison of communication complexity between our RSA-based instantiation and DPDP [14], for 1 GB file with variable block sizes. The detection probability is maintained to be 99%.

to commit necessary computation resources performing continuous verifications. Another major concern is how to construct verification protocols that can accommodate *dynamic* data files. In this paper, we explored the problem of providing simultaneous public verifiability and data dynamics for remote data integrity check in Cloud Computing. Our construction is deliberately designed to meet these two important goals while efficiency being kept closely in mind. We extended the PoR model [1] by using an elegant Merkle hash tree construction to

achieve fully dynamic data operation. Experiments show that our construction is efficient in supporting data dynamics with provable verification.

Acknowledgment

This work was supported in part by the US National Science Foundation under grant CNS-0831963, CNS-0626601, CNS-0716306, CNS-0831628 and CNS-0716302.

References

1. H. Shacham and B. Waters, "Compact proofs of retrievability," in *Proc. of ASIACRYPT'08*. Springer-Verlag, 2008, pp. 90–107.
2. G. Ateniese, R. Burns, R. Curtmola, J. Herring, L. Kissner, Z. Peterson, and D. Song, "Provable data possession at untrusted stores," in *Proc. of CCS'07*. New York, NY, USA: ACM, 2007, pp. 598–609.
3. A. Juels and B. S. Kaliski, Jr., "Pors: proofs of retrievability for large files," in *Proc. of CCS'07*. New York, NY, USA: ACM, 2007, pp. 584–597.
4. K. D. Bowers, A. Juels, and A. Oprea, "Proofs of retrievability: Theory and implementation," Cryptology ePrint Archive, Report 2008/175, 2008.
5. M. Naor and G. N. Rothblum, "The complexity of online memory checking," in *Proc. of FOCS'05*, 2005, pp. 573–584.
6. E.-C. Chang and J. Xu, "Remote integrity check with dishonest storage server," in *Proc. of ESORICS'08*. Berlin, Heidelberg: Springer-Verlag, 2008, pp. 223–237.
7. M. A. Shah, R. Swaminathan, and M. Baker, "Privacy-preserving audit and extraction of digital contents," Cryptology ePrint Archive, Report 2008/186, 2008.
8. A. Oprea, M. K. Reiter, and K. Yang, "Space-efficient block storage integrity," in *Proc. of NDSS'05*, 2005.
9. T. Schwarz and E. L. Miller, "Store, forget, and check: Using algebraic signatures to check remotely administered storage," in *Proc. of ICDCS'06*, 2006.
10. Q. Wang, K. Ren, W. Lou, and Y. Zhang, "Dependable and secure sensor data storage with dynamic integrity assurance," in *Proc. of IEEE INFOCOM'09*, Rio de Janeiro, Brazil, Appril 2009.
11. G. Ateniese, R. D. Pietro, L. V. Mancini, and G. Tsudik, "Scalable and efficient provable data possession," in *Proc. of SecureComm'08*, 2008.
12. C. Wang, K. Ren, and W. Lou, "Towards secure cloud data storage," *Proc. of IEEE GLOBECOM'09*, submitted on March 2009.
13. C. Wang, Q. Wang, K. Ren, and W. Lou, "Ensuring data storage security in cloud computing," in *Proc. of IWQoS'09*, Charleston, South Carolina, USA, 2009.
14. C. Erway, A. Kupcu, C. Papamanthou, and R. Tamassia, "Dynamic provable data possession," Cryptology ePrint Archive, Report 2008/432, 2008.
15. K. D. Bowers, A. Juels, and A. Oprea, "Hail: A high-availability and integrity layer for cloud storage," Cryptology ePrint Archive, Report 2008/489, 2008.
16. D. Boneh, B. Lynn, and H. Shacham, "Short signatures from the weil pairing," in *Proc. of ASIACRYPT'01*. London, UK: Springer-Verlag, 2001, pp. 514–532.
17. R. C. Merkle, "Protocols for public key cryptosystems," *Proc. of IEEE Symposium on Security and Privacy'80*, pp. 122–133, 1980.
18. Q. Wang, C. Wang, J. Li, K. Ren, and W. Lou, "Enabling public verifiability and data dynamics for storage security in cloud computing," Cryptology ePrint Archive, Report 2009/281, 2009.