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

Qian Wang, *Student Member, IEEE,* Cong Wang, *Student Member, IEEE,* Kui Ren, *Member, IEEE,* Wenjing Lou, *Senior Member, IEEE,* and Jin Li

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 the 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 auditability 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 the seamless integration of these two salient features in our protocol design. In particular, to achieve efficient data dynamics, we improve the existing proof of storage models by manipulating the classic Merkle Hash Tree construction for block tag authentication. To support efficient handling of multiple auditing tasks, we further explore the technique of bilinear aggregate signature to extend our main result into a multi-user setting, where TPA can perform multiple auditing tasks simultaneously. Extensive security and performance analysis show that the proposed schemes are highly efficient and provably secure.

Index Terms—Data storage, public auditability, data dynamics, cloud computing.

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 the problem of data integrity checking, many schemes are proposed under different systems and security models [2]–[11]. 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 auditability and public auditability. Although schemes with private auditability can achieve higher scheme efficiency, public auditability 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

- *Qian Wang, Cong Wang, and Kui Ren are with the Department of Electrical and Computer Engineering, Illinois Institute of Technology, 3301 South Dearborn Street, Suite 103 Siegel Hall, Chicago, IL, 60616 USA. E-mail: {qian,cong,kren}@ece.iit.edu.*
- *Wenjing Lou is with the Department of Electrical and Computer Engineering, Worcester Polytechnic Institute, 100 Institute Road, Worcester, MA 01609 USA. E-mail: wjlou@ece.wpi.edu.*
- *Jin Li is with the School of Computer Science and Educational Software, Guangzhou University, Guangzhou 510006, China. E-mail: jli25@iit.edu.*

A preliminary version [1] of this paper was presented at the 14th European Symposium on Research in Computer Security (ESORICS'09).

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 may not be able to afford the overhead of performing frequent integrity checks. Thus, for practical use, it seems more rational to equip the verification protocol with public auditability, which is expected to play a more important role in achieving economies of scale for Cloud Computing. Moreover, for efficiency consideration, the outsourced data themselves should not be required by the verifier for the verification purpose.

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, etc. 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 so far [2]–[5], [7], [10], [12]. Moreover, as will be shown later, the direct extension of the current provable data possession (PDP) [2] or proof of retrievability (PoR) [3], [4] 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 auditability and data dynamics for cloud data storage, we propose an efficient construction for the seamless integration of these two components in the protocol design. Our contribution can be summarized as follows:

1) We motivate the public auditing system of data storage security in Cloud Computing, and propose a protocol supporting for fully dynamic data operations, especially to support block insertion, which is missing in most existing schemes;
2) We extend our scheme to support scalable and efficient public auditing in Cloud Computing. In particular, our scheme achieves batch auditing where multiple delegated auditing tasks from different users can be performed simultaneously by the TPA.
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 [2]–[10], [12]–[15]. Ateniese *et al.* [2] are the first to consider public auditability in their defined "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 public auditability is achieved. 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 may suffer design and security problems. In their subsequent work [12], 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 a distributed scenario, and the proposed challenge-response protocol can both determine the data correctness and locate possible errors. Similar to [12], they only consider partial support for dynamic data operation. Juels *et al.* [3] describe a "proof of retrievability" (PoR) model, where 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 [12], 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 auditability is not supported in their scheme. Shacham *et al.* [4] design an improved PoR scheme with full proofs of security in the security model defined in [3]. They use publicly verifiable homomorphic authenticators built from BLS signatures [16], based on which the proofs can be aggregated into a small authenticator value, and public retrievability is achieved. Still, the authors only consider static data files. Erway *et 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. To support updates, especially for block insertion, they eliminate the index information in the "tag" computation in Ateniese's PDP model [2] and employ authenticated skip list data structure to authenticate the tag information of challenged or updated blocks first before the verification procedure. However, the efficiency of their scheme remains unclear.

Although the existing schemes aim at providing integrity verification for different data storage systems, the problem of supporting both public auditability 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.

Portions of the work presented in this article have previously appeared as an extended abstract [1]. We revise the article a lot and add more technical details as compared to [1]. Firstly, in Section III-C, before the introduction of our proposed construction we present

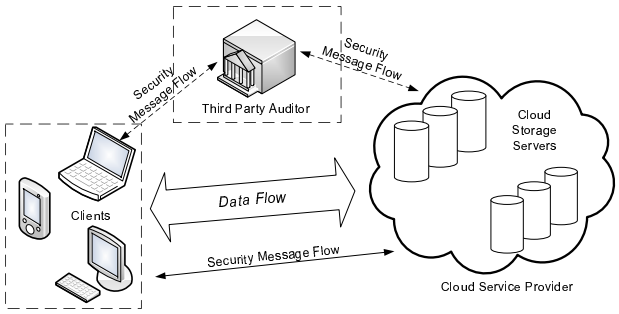


Fig. 1: Cloud data storage architecture

two basic solutions (*i.e.*, the MAC-based and signature-based schemes) for realizing data auditability and discuss their demerits in supporting public auditability and data dynamics. Secondly, we generalize the support of data dynamics to both proof of retrievability (PoR) and provable data possession (PDP) models and discuss the impact of dynamic data operations on the overall system efficiency both. In particular, we emphasize that while dynamic data updates can be performed efficiently in PDP models more efficient protocols need to be designed for the update of the encoded files in PoR models. For completeness, the designs for distributed data storage security are also discussed in Section III-E. Thirdly, in Section III-D, we extend our data auditing scheme for the single client and explicitly include a concrete description of the multi-client data auditing scheme. We also redo the whole experiments and present the performance comparison between the multi-client data auditing scheme and the individual auditing scheme in Section V. Finally, for the proposed theorems in this article, we provide formal security proofs under the random oracle model, which are lacking in [1].

Organization. The rest of the paper is organized as follows. In section II, we define the system model, security model and our design goals. Then, we present our scheme in section III and provide security analysis in section IV, respectively. We further analyze the experiment results and show the practicality of our schemes in section V. Finally, we conclude in section VI.

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 the clients' data;
- *Third Party Auditor* (TPA): an entity, 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. In this paper, we only consider verification schemes with public auditability: any TPA in possession of the public key can act as a verifier. We assume that TPA is unbiased while the server is untrusted. 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*. 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.

2.2 Security Model

Following the security model defined in [4], we say that 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. The client or TPA 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 [4] also define the correctness and soundness of their 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). The goal of the adversary is to cheat the verifier 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 existing PDP or PoR models [2]–[4] in the verification process. As mentioned above, these schemes do not consider dynamic data operations, and

the block insertion cannot be supported at all. This is because the construction of the signatures is involved with the file index information i. Therefore, once a file block is inserted, the computation overhead is unacceptable since the signatures of all the following file blocks should be re-computed with the new indexes. To deal with this limitation, we remove the index information i in the computation of signatures and use $H(m_i)$ as the tag for block m_i instead of $H(name||i)$ [4] or $h(v||i)$ [3], so individual data operation on any file block will not affect the others. Recall that in existing PDP or PoR models [2], [4], $H(name||i)$ or $h(v||i)$ should be generated by the client in the verification process. However, in our new construction the client has no capability to calculate $H(m_i)$ without the data information. In order to achieve this *blockless* verification, 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 PDP or PoR models in both the verification and the data updating process. Specifically, the tags in our scheme should be authenticated in each protocol execution other than calculated or pre-stored by the verifier (The details will be shown in section III). In the following descriptions, we will use server and prover (or client, TPA and verifier) interchangeably.

2.3 Design Goals

Our design goals can be summarized as the following:

1) Public auditability 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 auditability 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 efficiency concern.

3 The Proposed Scheme

In this section, we present our security protocols for cloud data storage service with the aforementioned research goals in mind. We start with some basic solutions aiming to provide integrity assurance of the cloud data and discuss their demerits. Then we present our protocol which supports public auditability and data dynamics. We also show how to extent our main scheme to support batch auditing for TPA upon delegations from multi-users.

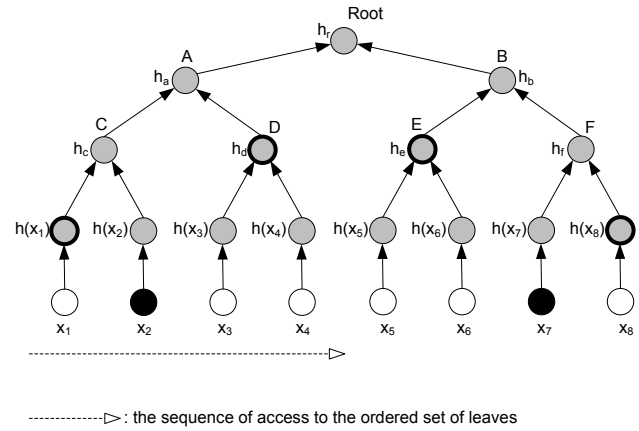


Fig. 2: Merkle hash tree authentication of data elements. We treat the leaf nodes $h(x_1), \ldots, h(x_n)$ as the left-to-right sequence.

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. Fig. 2 depicts an example of authentication. The verifier with the authentic h_r requests for $\{x_2, x_7\}$ and requires the authentication of the received blocks. The prover provides the verifier with the auxiliary authentication information (AAI) $\Omega_2 =< h(x_1), h_d >$ and $\Omega_7 =< h(x_8), h_e >$. The verifier can then verify x_2 and x_7 by first computing $h(x_2)$, $h(x_7)$, $h_c = h(h(x_1)||h(x_2)))$, $h_f = h(h(x_7)||h(x_8)))$, $h_a = h(h_c||h_d)$, $h_b = h(h_e||h_f)$ and $h_r = h(h_a||h_b)$, and then checking if the calculated h_r is the same as the authentic one. 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.*

$\{(TRUE, FALSE, sig_{sk}(H(R')))\} \leftarrow VerifyUpdate(pk, update, P_{update})$. *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 Basic Solutions

Assume the outsourced data file F consists of a finite ordered set of blocks $m_1, m_2, \ldots, m_n$. One straightforward way to ensure the data integrity is to pre-compute MACs for the entire data file. Specifically, before data outsourcing, the data owner pre-computes MACs of F with a set of secret keys and stores them locally. During the auditing process, the data owner each time reveals a secret key to the cloud server and asks for a fresh keyed MAC for verification. This approach provides deterministic data integrity assurance straightforwardly as the verification covers all the data blocks. However, the number of verifications allowed to be performed in this solution is limited by the number of secret keys. Once the keys are exhausted, the data owner has to retrieve the entire file of F from the server in order to compute new MACs, which is usually impractical due to the huge communication overhead. Moreover, public auditability is not supported as the private keys are required for verification.

Another basic solution is to use signatures instead of MACs to obtain public auditability. The data owner pre-computes the signature of each block m_i ($i \in [1, n]$) and sends both F and the signatures to the cloud server for storage. To verify the correctness of F, the data owner can adopt a spot-checking approach, *i.e.*, requesting a number of randomly selected blocks and their corresponding signatures to be returned. This basic solution can provide probabilistic assurance of the data correctness and support public auditability. However, it also severely suffers from the fact that a considerable number of original data blocks should be retrieved to ensure a reasonable detection probability, which again could result in a large communication overhead and greatly affects system efficiency. Notice that the above solutions can only support the case of static data, and none of them can deal with dynamic data updates.

3.4 Our Construction

To effectively support public auditability without having to retrieve the data blocks themselves, we resort to the homomorphic authenticator technique [2], [4]. Homomorphic authenticators are unforgeable metadata generated from individual data blocks, which can be securely aggregated in such a way to assure a verifier that a linear combination of data blocks is correctly computed by verifying only the aggregated authenticator. In our design, we propose to use PKC based homomorphic authenticator (*e.g.*, BLS signature [4] or RSA signature based authenticator [2]) to equip the verification protocol with public auditability. In the following description, we present the BLS-based scheme to illustrate our design 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 III-D, we show that direct extensions of previous work [2], [4] 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. We assume that file F (potentially encoded using Reed-Solomon codes [18]) is divided into n blocks $m_1, m_2, \ldots, m_n$[1], where $m_i \in \mathbb{Z}_p$ and p is a large prime. Let $e : G \times G \rightarrow G_T$ be a bilinear map, with a hash function $H : \{0,1\}^* \rightarrow G$, viewed as a random oracle [19]. 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 data file F is pre-processed, and the homomorphic authenticators together with metadata are produced.

$KeyGen(1^k)$. The client generates a random signing key pair (spk, ssk). Choose a random $\alpha \leftarrow \mathbb{Z}_p$ and compute $v \leftarrow g^\alpha$. The secret key is $sk = (\alpha, ssk)$ and the public key is $pk = (v, spk)$.

$SigGen(sk, F)$. Given $F = (m_1, m_2 \ldots, m_n)$, the client chooses a random element $u \leftarrow G$. Let $t = name||n||u||\mathsf{SSig}_{ssk}(name||n||u)$ be the file tag for F. Then the client computes signature σ_i for each block m_i ($i = 1, 2, \ldots, n$) as $\sigma_i \leftarrow (H(m_i) \cdot u^{m_i})^\alpha$. Denote the set of signatures by $\Phi = \{\sigma_i\}$, $1 \leq i \leq n$. The client then generates a root R based on the construction

1. We assume these blocks are distinct with each other and a systematic code is used for encoding.

TPA		CSS
1. Generate a random set $\{(i, \nu_i)\}_{i \in I}$;		
	$\xrightarrow[\text{challenge request chal}]{\{(i,\nu_i)\}_{i \in I}}$	
		2. Compute $\mu = \sum_i \nu_i m_i$;
		3. Compute $\sigma = \prod_i \sigma_i^{\nu_i}$;
	$\xleftarrow[\text{Integrity proof } P]{\{\mu,\sigma,\{H(m_i),\Omega_i\}_{i \in I}, sig_{sk}(H(R))\}}$	
4. Compute R using $\{H(m_i), \Omega_i\}_{i \in I}$;		
5. Verify $sig_{sk}(H(R))$ and output *FALSE* if fail;		
6. Verify $\{m_i\}_{i \in I}$.		

TABLE 1: Protocols for Default Integrity Verification

of Merkle Hash Tree (MHT), where the leave nodes of the tree are an ordered set of hashes of "file tags" $H(m_i)$ $(i = 1, 2, \ldots, n)$. Next, the client signs the root R under the private key α: $sig_{sk}(H(R)) \leftarrow (H(R))^{\alpha}$. The client sends $\{F, t, \Phi, sig_{sk}(H(R))\}$ to the server and deletes $\{F, \Phi, sig_{sk}(H(R))\}$ from its local storage (See Section III-D for the discussion of blockless and stateless verification).

■ **Default Integrity Verification**: The client or TPA can verify the integrity of the outsourced data by challenging the server. Before challenging, the TPA first use spk to verify the signature on t. If the verification fails, reject by emitting *FALSE*; otherwise, recover u. To generate the message "$chal$", the TPA (verifier) picks a random c-element subset $I = \{s_1, s_2, \ldots, s_c\}$ of set $[1, n]$, where we assume $s_1 \leq \cdots \leq s_c$. For each $i \in I$ the TPA chooses a random element $\nu_i \leftarrow B \subseteq \mathbb{Z}_p$. The message "$chal$" specifies the positions of the blocks to be checked in this challenge phase. The verifier sends the $chal$ $\{(i, \nu_i)\}_{s_1 \leq i \leq s_c}$ to the prover (server).

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

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

where both the data blocks and the corresponding signature blocks are aggregated into a single block, respectively. In addition, the prover will also provide the verifier with a small amount of auxiliary information $\{\Omega_i\}_{s_1 \leq i \leq s_c}$, which are the node siblings on the path from the leaves $\{h(H(m_i))\}_{s_1 \leq i \leq s_c}$ to the root R of the MHT. The prover responds the verifier with proof $P = \{\mu, \sigma, \{H(m_i), \Omega_i\}_{s_1 \leq i \leq s_c}, sig_{sk}(H(R))\}$.

$VerifyProof(pk, chal, P)$. Upon receiving the responses from the prover, the verifier generates root R using $\{H(m_i), \Omega_i\}_{s_1 \leq i \leq s_c}$ 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 Table 1.

■ **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, 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 Table 2. 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. 3). 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

Client		CSS
1. Generate $\sigma'_i = (H(m'_i) \cdot u^{m'_i})^\alpha$;		
	$\xrightarrow[update\ request\ update]{(\mathcal{M}(\mathcal{I}), i, m'_i, \sigma'_i)}$	
		2. Update F and compute R'.
	$\xleftarrow[update\ proof\ P_{update}]{(\Omega_i, H(m_i), sig_{sk}(H(R)), R')}$	
3. Compute R using $\{H(m_i), \Omega_i\}$;		
4. Verify $sig_{sk}(H(R))$. Output *FALSE* if fail.		
5. Compute R_{new} using $\{\Omega_i, H(m'_i)\}$. Verify *update* by checking $R_{new} \stackrel{?}{=} R'$. Sign R' if succeed.	$\xrightarrow{sig_{sk}(H(R'))}$	6. Update R's signature.

TABLE 2: The protocol for provable data update (Modification and Insertion)

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, 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. Finally, the client executes the default integrity verification protocol. If the output is *TRUE*, delete $sig_{sk}(H(R'))$, P_{update} and m'_i from its local storage.

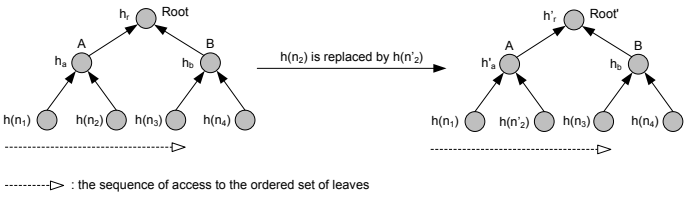


Fig. 3: Example of MHT update under block modification operation. Here, n_i and n'_i are used to denote $H(m_i)$ and $H(m'_i)$, respectively.

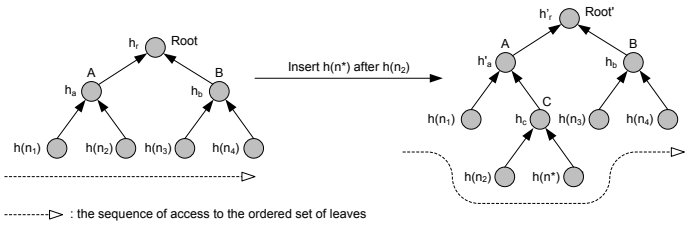


Fig. 4: Example of MHT update under block insertion operation. Here, n_i and n^* are used to denote $H(m_i)$ and $H(m^*)$, respectively.

-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.

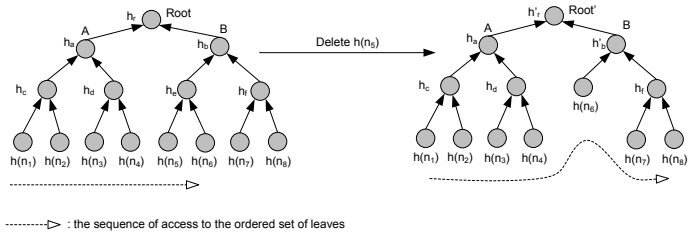


Fig. 5: Example of MHT update under block deletion operation.

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 Table 2, 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. 4, 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 then 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. Finally, the client executes the default integrity verification protocol. If the output is *TRUE*, delete $sig_{sk}(H(R'))$, P_{update} and m^* from its local storage.

-***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. 5). The details of the protocol procedures are similar to that of data modification and insertion, which are thus omitted here.

■ **Batch Auditing for Multi-client Data:** As cloud servers may concurrently handle multiple verification sessions from different clients, given K signatures on K distinct data files from K clients, it is more advantageous to aggregate all these signatures into a single short one and verify it at one time. To achieve this goal, we extend our scheme to allow for provable data updates and verification in a multi-client system. The key idea is to use the bilinear aggregate signature scheme [20], which has the following property: for any $u_1, u_2, v \in G$, $e(u_1u_2, v) = e(u_1, v) \cdot e(u_2, v)$ and for any $u, v \in G$, $e(\psi(u), v) = e(\psi(v), u)$. As in the BLS based construction, the aggregate signature scheme allows the creation of signatures on arbitrary distinct messages. Moreover, it supports the aggregation of multiple signatures by distinct signers on distinct messages into a single short signature, and thus greatly reduces the communication cost while providing efficient verification for the authenticity of all messages.

Assume there are K clients in the system, and each client k has data files $F_i = (m_{k,1}, \ldots, m_{k,n})$, where $k \in \{1, \ldots, K\}$. The protocol is executed as follows. For a particular client k, pick random $x_k \leftarrow \mathbb{Z}_p$, and compute $v_k = g^{x_k}$. The client's public key is $v_k \in G$ and the public key is $v_k \in \mathbb{Z}_p$. In the *SigGen* phase, given the file $F_k = (m_{k,1}, \ldots, m_{k,n})$, client k chooses a random element $u_k \leftarrow G$ and computes signature $\sigma_{k,i} \leftarrow [H(m_{k,i}) \cdot u_k^{m_{k,i}}]^{x_k} \in G$. In the *challenge* phase, the verifier sends the query $Q = \{(i, \nu_i)\}_{s_1 \le i \le s_c}$ to the prover (server) for verification of all K clients. In the *GenProof* phase, upon receiving the *chal*, for each client k ($k \in \{1, \ldots, K\}$), the prover computes $\mu_k = \sum_{\{(i,\nu_i)\}_{s_1 \le i \le s_c}} \nu_i m_{k,i} \in \mathbb{Z}_p$ and $\sigma = \prod_{k=1}^{K} (\prod_{\{(i,\nu_i)\}_{s_1 \le i \le s_c}} \sigma_{k,i}^{\nu_i}) = \prod_{k=1}^{K} (\prod_{\{(i,\nu_i)\}_{s_1 \le i \le s_c}} [H(m_{k,i}) \cdot u_k^{m_{k,i}}]^{x_k \nu_i})$. The prover then responses the verifier with $\{\sigma, \{\mu_k\}_{1 \le k \le K}, \{\Omega_{k,i}\}, \{H(m_{k,i})\}\}$. In the *VerifyProof* phase, similar as the single client case, the verifier first authenticates tags $H(m_{k,i})$ by verifying signatures on the roots (for each client's file). If the authentication succeeds, then, using the properties of the bilinear map, the verifier can check if the following equation holds:

$$e(\sigma, g) = \prod_{k=1}^{K} e(\prod_{\{(i,\nu_i)\}_{s_1 \le i \le s_c}} [H(m_{k,i})]^{\nu_i} \cdot (u_k)^{\mu_k}, v_k).$$

The above equation is similar to the checking equation in the single-client case, and it holds because:

$$\begin{aligned} e(\sigma, g) &= e(\prod_{k=1}^{K} (\prod_{\{(i,\nu_i)\}_{s_1 \le i \le s_c}} \sigma_{k,i}^{\nu_i}), g) \\ &= e(\prod_{k=1}^{K} (\prod_{\{(i,\nu_i)\}_{s_1 \le i \le s_c}} [H(m_{k,i}) \cdot u_k^{m_{k,i}}]^{x_k \nu_i}), g) \\ &= \prod_{k=1}^{K} e([\prod_{\{(i,\nu_i)\}_{s_1 \le i \le s_c}} [H(m_{k,i})]^{\nu_i} \cdot (u_k)^{\mu_k}]^{x_k}, g) \\ &= \prod_{k=1}^{K} e(\prod_{\{(i,\nu_i)\}_{s_1 \le i \le s_c}} [H(m_{k,i})]^{\nu_i} \cdot (u_k)^{\mu_k}, g^{x_k}). \end{aligned}$$

3.5 Discussion on Design Considerations

Instantiations based on BLS and RSA. As discussed above, we present a BLS-based construction that offers both public auditability 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 [4]. 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 $H(m_i)$ (or g^{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.

In basic PDP constructions, the system stores static files (*e.g.*, archival or backup) without error correction capability. Thus, file updates can be performed efficiently. In a PoR system, as all or part of data files are encoded, frequent or small data updates requires the updates of all related (encoded) files. In this paper, we do not constrain ourselves in a specific model, and our scheme can be applied in both PDP and PoR models. However, the design of protocols for supporting efficient data operation in PoR systems still remains an open problem.

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. For this 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. However, a rational cheating server would not do this unless the reversion of data updates benefits it much. 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.

Designs for Distributed Data Storage Security. To further enhance the availability of the data storage security, individual user's data can be redundantly stored in multiple physical locations. That is, besides being exploited at individual servers, data redundancy can also be employed across multiple servers to tolerate faults or server crashes as user's data grows in size and importance. It is well known that erasure-correcting code can be used to tolerate multiple failures in distributed storage systems. In cloud data storage, we can rely on this technique to disperse the data file F redundantly across a set of $n = m+k$ distributed servers. A $[m+k, k]$-Reed-Solomon code is used to create k redundancy parity vectors from m data vectors in such a way that the original m data vectors can be reconstructed from any m out of the $m+k$ data and parity vectors. By placing each of the $m+k$ vectors on a different server, the original data file can survive the failure of any k of the $m+k$ servers without any data loss. Such a distributed cryptographic system allows a set of servers to prove to a client that a stored file is intact and retrievable.

4 Security Analysis

In this section, we evaluate the security of the proposed scheme under the security model defined in Section 2.2. Following [4], we consider a file F after Reed-Solomon coding.

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

We say that the (t, ϵ)-CDH assumption holds in G if no t-time algorithm has the non-negligible probability ϵ in solving the CDH problem. A proof-of-retrievability protocol is sound if any cheating prover that convinces the verification algorithm that it is storing a file F is actually storing that file, which we define in saying that it yields up the file F to an extractor algorithm which interacts with it using the proof-of-retrievability protocol. We say that the adversary (cheating server) is *ϵ-admissible* if it convincingly answers an ϵ-fraction of verification

Metric \ Scheme	[2]	[4]	[12]*	[14]	Our Scheme
Data dynamics	**No**		**Yes**		
Public auditability	**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 3: 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 auditability 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)	6.45	2.11	13.81	4.55	14.13
Verifier comp. time (ms)	806.01	284.17	779.10	210.47	782.56
Comm. cost (KB)	239	80	223	76	280

TABLE 4: 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.

challenges. We formalize the notion of an extractor and then give a precise definition for soundness.

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.

Proof: See Appendix. □

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 $\omega = 1/\#B + (\rho n)^{\ell}/(n-c+1)^{c}$. 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.

Proof: The verification of the proof-of-retrievability is similar to [4], we omit the details of the proof here. The difference in our work is to replace $H(i)$ with $H(m_i)$ such that secure update can still be realized without including the index information. These two types of tags are used for the same purpose (*i.e.*, to prevent potential attacks), so this change will not affect the extraction algorithm defined in the proof-of-retrievability. We can also prove that extraction always succeeds against a well-behaved cheating prover, with the same probability analysis given in [4]. □

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.

Proof: Based on the rate-ρ Reed-Solomon codes, this result can be easily derived, since any ρ-fraction of encoded file blocks suffices for decoding.

□

The security proof for the multi-client batch auditing is similar to the single client case, thus omitted here.

5 Performance Analysis

We list the features of our proposed scheme in Table 3 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 4 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 4, 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) compared to RSA-based instantiation, our BLS-based instantiation is more than 2 times faster than the other two in terms of server computation time. However, its has

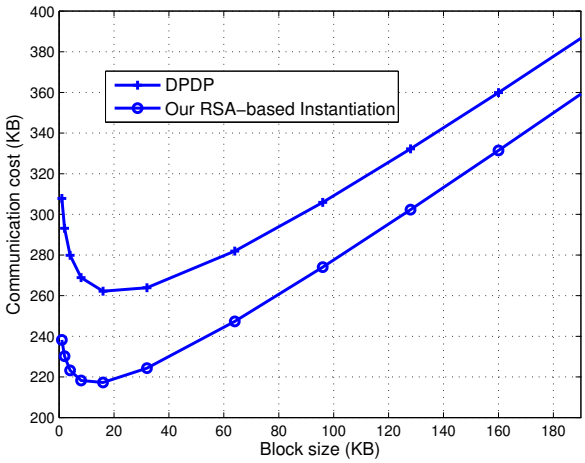


Fig. 6: Comparison of communication complexity between our RSA-based instantiation and DPDP [14], for 1 GB file with variable block sizes. The detection probability is maintained to be 99%.

larger computation cost at the verifier side as the pairing 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. 6. We can see that the communication cost grows almost linearly as the block size increases, which is mainly caused by the increasing in size of the verification block $\mu = \sum_{i=s_1}^{s_c} \nu_i m_i$. However, the experiments suggest that when block size is chosen around 16KB, both schemes can achieve an optimal point that minimizes the total communication cost.

We also conduct experiments for multi-client batch auditing and demonstrate its efficiency in Figure 7, where the number of clients in the system is increased from 1 to approximately 100 with intervals of 4. As we can see, batch auditing not only enables simultaneously verification from multiple-client, but also reduces the computation cost on the TPA side. Given total K clients in the system, the batch auditing equation helps reduce the number of expensive pairing operations from $2K$, as required in the individual auditing, to $K + 1$. Thus, a certain amount of auditing time is expected to be saved. Specifically, following the same experiment setting as ρ = 99% and 97%, batch auditing indeed saves TPA's computation overhead for about 5% and 14%, respectively. Note that in order to maintain detection probability of 99%, the random sample size in TPA's challenge for $\rho = 99\%$ is quite larger than $\rho = 97\%$: 460 versus 152. As this sample size is also a dominant factor of auditing time, this explains why batch auditing for $\rho = 99\%$ is not as efficient as for $\rho = 97\%$.

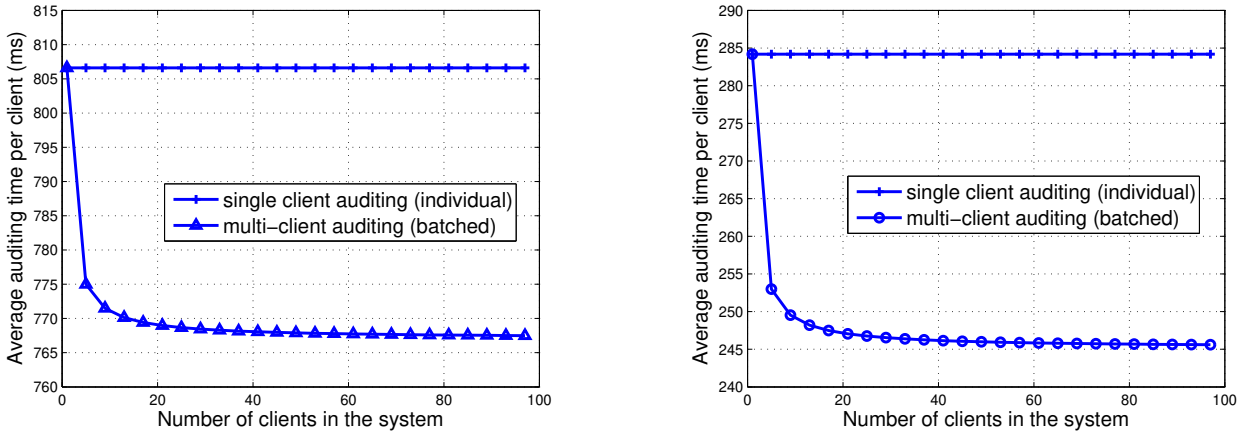
6 Conclusion

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 auditability also allows clients to delegate the integrity verification tasks to TPA while they themselves can be unreliable or not be able 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 auditability 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. To achieve efficient data dynamics, we improve the existing proof of storage models by manipulating the classic Merkle Hash Tree (MHT) construction for block tag authentication. To support efficient handling of multiple auditing tasks, we further explore the technique of bilinear aggregate signature to extend our main result into a multi-user setting, where TPA can perform multiple auditing tasks simultaneously. Extensive security and performance analysis show that the proposed scheme is highly efficient and provably secure.

Appendix A
Proof of the Theorem 1

Proof: It is easy to prove that the signature scheme is existentially unforgeable with the assumption that BLS [16] short signature scheme is secure. In concrete, assume there is a secure BLS signature scheme, with public key $y = g^{\alpha}$ and a map-to-point hash function H. If there is an adversary that can break our signature scheme, we show how to use this adversary to forge a BLS signature as follows: Set $u = g^{x_0}$ by choosing x_0 from Z_p. For any signature query on message m, we can submit this message to BLS signing oracle and get $\sigma = H(m)^{\alpha}$. Therefore, the signing oracle of this new signature scheme can be simulated as $\sigma' = \sigma y^{mx_0} = (H(m)g^{mx_0})^{\alpha}$. Finally, if there is any adversary can forge a new signature $\sigma' = (H(m')u^{m'})^{\alpha}$ on a message m' that has never been queried, we can get a forged BLS signature on the message m' as $\sigma = \sigma'/y^{m'x_0} = H(m')^{\alpha}$. This completes the proof of the new signature scheme that the BLS signature scheme is secure.

We then prove the theorem by using a sequence of games as defined in [4]. The first game, Game 0, is simply the challenge game, which is similar to [4], with the changes for public auditability sketched. Game 1 is the same as Game 0, with one difference. The challenger keeps a list of all signed tags ever issued as part of a store-protocol query. If the adversary ever submits a tag either in initiating a proof-of-retrievability protocol or as



(a) Tolerance rate ρ is 99%.

(b) Tolerance rate ρ is 97%.

Fig. 7: Performance comparison between individual auditing and batch auditing. The average per client auditing time is computed by dividing total auditing time by the number of clients in the system. For both tolerance rate $\rho = 99\%$ and $\rho = 97\%$, the detection probability is maintained to be 99%.

the challenge tag, the challenger will abort if it is a valid tag that has never been signed by the challenger. Based on the definition of Game 0 and Game 1, it is obviously that we can use the adversary to construct a forger against the signature scheme, if there is a difference in the adversary's success probability between Games 0 and 1.

Game 2 is the same as Game 1, except that in Game 2, the challenger keeps a list of its responses to queries from the adversary. Now the challenger observes each instance of the proof-of-retrievability protocol with the adversary. Let $P = \{\mu, \sigma, \{H(m_i), \Omega_i\}_{s_1 \le i \le s_c}, sig_{sk}(H(R))\}$ be the expected response that would have been obtained from an honest prover. The correctness of $H(m_i)$ can be verified through $\{H(m_i), \Omega_i\}_{s_1 \le i \le s_c}$ and $sig_{sk}(H(R))$. The correctness of the proof can be verified based on the following equation $e(\sigma, g) = e(\prod_{\{(i,\nu_i)\}_{s_1 \le i \le s_c}} H(m_i)^{\nu_i} \cdot u^{\mu}, v)$. Assume the adversary's response is P'. Because of the authentication in MHT, the second part in P' should be the same with $\{H(m_i), \Omega_i\}_{s_1 \le i \le s_c}$ and $sig_{sk}(H(R))$. Suppose $P' = \{\mu', \sigma', \{H(m_i), \Omega_i\}_{s_1 \le i \le s_c}, sig_{sk}(H(R))\}$ is the adversary's response. The verification of (μ', σ') is $e(\sigma', g) = e(\prod_{\{(i,\nu_i)\}_{s_1 \le i \le s_c}} H(m_i)^{\nu_i} \cdot u^{\mu'}, v)$. Obviously, $\mu' \neq \mu$, otherwise, $\sigma' = \sigma$, which contradicts our assumption in this game. Define $\Delta\mu = \mu' - \mu$. With this adversary, the simulator could break the challenge CDH instance as follows: Given $(g, g^{\alpha}, h) \in G$, the simulator is asked to output h^{α}. The simulator sets $v = g^{\alpha}$ and $u = g^a h^b$ for $a, b \in Z_p^*$. The simulator could answer the signature query with similar method as described in [4], by letting $H(m_i) = g^{r_i} h^{-m_i}$. Finally, the adversary outputs $P' = \{\mu', \sigma', \{H(m_i), \Omega_i\}_{s_1 \le i \le s_c}, sig_{sk}(H(R))\}$. We obtain $e(\sigma'/\sigma, g) = e(u^{\Delta\mu}, v) = e((g^a h^b)^{\Delta\mu}, g^{\alpha})$. From this equation, we have $e(\sigma'\sigma^{-1} v^{-a\Delta\mu}, g) = e(h, v)^{b\Delta\mu}$. Therefore, $h^{\alpha} = (\sigma'\sigma^{-1} v^{a\Delta\mu})^{\frac{1}{b\Delta\mu}}$ because $v = g^{\alpha}$. To analyze the probability that the challenger aborts in the game, we only need to compute the probability that $b\Delta\mu = 0 \bmod p$. Because b is chosen by the challenger and hidden from the adversary, the probability that $b\Delta\mu = 0 \bmod p$ will be only $1/p$, which is negligible.

Game 3 is the same as Game 2, with the following difference: As before, the challenger observes proof-of-retrievability protocol instances. Suppose the file that causes the abort is that the signatures are $\{\sigma_i\}$. Suppose $Q = (i, v_i)_{s_1 \le i \le s_c}$ is the query that causes the challenger to abort, and that the adversary's response to that query was $P' = \{\mu', \sigma', \{H(m_i), \Omega_i\}_{s_1 \le i \le s_c}, sig_{sk}(H(R))\}$. Let $P = \{\mu, \sigma, \{H(m_i), \Omega_i\}_{s_1 \le i \le s_c}, sig_{sk}(H(R))\}$ be the expected response obtained from an honest prover. We have proved in Game 2 that $\sigma = \sigma'$. It is only the values μ and μ' that can differ. Define $\Delta\mu = \mu' - \mu$. The simulator answers the adversary's queries. Finally, The adversary outputs a forged signature $P' = \{\mu', \sigma', \{H(m_i), \Omega_i\}_{s_1 \le i \le s_c}, sig_{sk}(H(R))\}$. Now we have $e(\sigma', g) = e(\prod_{\{(i,\nu_i)\}_{s_1 \le i \le s_c}} H(m_i)^{\nu_i} \cdot u^{\mu'}, v) = e(\sigma, g) = e(\prod_{\{(i,\nu_i)\}_{s_1 \le i \le s_c}} H(m_i)^{\nu_i} \cdot u^{\mu}, v)$. From this equation, we have $1 = u^{\Delta\mu}$. In this case, $\Delta\mu = 0 \bmod p$. Therefore, we have $\mu = \mu' \bmod p$.

As we analyzed above, there is only negligible difference probability between these games. This completes the proof.

□

ACKNOWLEDGMENTS

The authors would like to thank the editor and anonymous reviewers for their valuable suggestions that significantly improved the quality of this paper. 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] Q. Wang, C. Wang, J. Li, K. Ren, and W. Lou, "Enabling public verifiability and data dynamics for storage security in cloud computing," in *Proc. of ESORICS'09*. Saint Malo, France: Springer-Verlag, 2009, pp. 355–370.

[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] H. Shacham and B. Waters, "Compact proofs of retrievability," in *Proc. of ASIACRYPT'08*. Melbourne, Australia: Springer-Verlag, 2008, pp. 90–107.

[5] K. D. Bowers, A. Juels, and A. Oprea, "Proofs of retrievability: Theory and implementation," Cryptology ePrint Archive, Report 2008/175, 2008.

[6] M. Naor and G. N. Rothblum, "The complexity of online memory checking," in *Proc. of FOCS'05*, Pittsburgh, PA, USA, 2005, pp. 573–584.

[7] 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.

[8] M. A. Shah, R. Swaminathan, and M. Baker, "Privacy-preserving audit and extraction of digital contents," Cryptology ePrint Archive, Report 2008/186, 2008.

[9] A. Oprea, M. K. Reiter, and K. Yang, "Space-efficient block storage integrity," in *Proc. of NDSS'05*, San Diego, CA, USA, 2005.

[10] T. Schwarz and E. L. Miller, "Store, forget, and check: Using algebraic signatures to check remotely administered storage," in *Proc. of ICDCS'06*, Lisboa, Portugal, 2006, pp. 12–12.

[11] 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, pp. 954–962.

[12] G. Ateniese, R. D. Pietro, L. V. Mancini, and G. Tsudik, "Scalable and efficient provable data possession," in *Proc. of SecureComm'08*. New York, NY, USA: ACM, 2008, pp. 1–10.

[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," in *Proc. of CCS'09*. Chicago, IL, USA: ACM, 2009.

[15] K. D. Bowers, A. Juels, and A. Oprea, "Hail: A high-availability and integrity layer for cloud storage," in *Proc. of CCS'09*. Chicago, IL, USA: ACM, 2009, pp. 187–198.

[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.

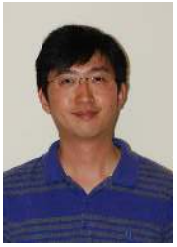
[17] R. C. Merkle, "Protocols for public key cryptosystems," *Proc. of IEEE Symposium on Security and Privacy'80*, pp. 122–133, 1980.

[18] S. Lin and D. J. Costello, *Error Control Coding, Second Edition*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 2004.

[19] M. Bellare and P. Rogaway, "Random oracles are practical: A paradigm for designing efficient protocols," in *Proc. of CCS'93*, 1993, pp. 62–73.

[20] D. Boneh, C. Gentry, B. Lynn, and H. Shacham, "Aggregate and verifiably encrypted signatures from bilinear maps," in *Proc. of Eurocrypt'03*. Warsaw, Poland: Springer-Verlag, 2003, pp. 416–432.

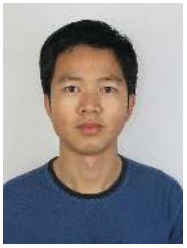
Qian Wang received the B.S. degree from Wuhan University, China, in 2003 and the M.S. degree from Shanghai Institute of Microsystem and Information Technology, Chinese Academy of Sciences, China, in 2006, both in Electrical Engineering. He is currently working towards the Ph.D. degree in the Electrical and Computer Engineering Department at Illinois Institute of Technology. His research interests include wireless network security and privacy, and Cloud Computing security.



Cong Wang received his B.E and M.E degrees from Wuhan University, China, in 2004 and 2007, respectively. He is currently a Ph.D student in the Electrical and Computer Engineering Department at Illinois Institute of Technology. His research interests are in the areas of applied cryptography and network security, with current focus on secure data services in Cloud Computing.

Kui Ren is an assistant professor in the Electrical and Computer Engineering Department at Illinois Institute of Technology. He obtained his PhD degree in Electrical and Computer Engineering from Worcester Polytechnic Institute in 2007. He received his B. Eng and M. Eng both from Zhejiang University in 1998 and 2001, respectively. In the past, he has worked as a research assistant at Shanghai Institute of Microsystem and Information Technology, Chinese Academy of Sciences, at Institute for Infocomm Research, Singapore, and at Information and Communications University, South Korea. His research interests include network security & privacy and applied cryptography with current focus on security & privacy in cloud computing, lower-layer attack & defense mechanisms for wireless networks, and sensor network security. His research is sponsored by US National Science Foundation. He is a member of IEEE and ACM.

Wenjing Lou earned a BE and an ME in Computer Science and Engineering at Xi'an Jiaotong University in China, an MASc in Computer Communications at the Nanyang Technological University in Singapore, and a PhD in Electrical and Computer Engineering at the University of Florida. From December 1997 to July 1999, she worked as a Research Engineer at Network Technology Research Center, Nanyang Technological University. She joined the Electrical and Computer Engineering department at Worcester Polytechnic Institute as an assistant professor in 2003, where she is now an associate professor. Her current research interests are in the areas of ad hoc, sensor, and mesh networks, with emphases on network security and routing issues. She has been an editor for IEEE Transactions on Wireless Communications since 2007. She was named Joseph Samuel Satin Distinguished fellow in 2006 by WPI. She is a recipient of the U.S. National Science Foundation Faculty Early Career Development (CAREER) award in 2008.



Jin Li received his B.S. degree in mathematics from the Southwest University, China, in 2002, and his Ph.D. degree from the University of Sun Yat-sen University, China, in 2007. From 2009 to 2010 he was an associate senior researcher in School of Electrical and Computer Engineering at Illinois Institute of Technology. He joined the School of Computer Science and Educational Software at Guangzhou University as a professor in 2010. His current research interests include the design of cryptographic protocols and secure protocols in Cloud Computing with focus on privacy and anonymity.