Towards Secure and Dependable Storage Services in Cloud Computing

Cong Wang, *Student Member, IEEE*, Qian Wang, *Student Member, IEEE*, Kui Ren, *Member, IEEE*, Ning Cao, *Student Member, IEEE*, and Wenjing Lou, *Senior Member, IEEE*

Abstract—Cloud storage enables users to remotely store their data and enjoy the on-demand high quality cloud applications without the burden of local hardware and software management. Though the benefits are clear, such a service is also relinquishing users' physical possession of their outsourced data, which inevitably poses new security risks towards the correctness of the data in cloud. In order to address this new problem and further achieve a secure and dependable cloud storage service, we propose in this paper a flexible distributed storage integrity auditing mechanism, utilizing the homomorphic token and distributed erasure-coded data. The proposed design allows users to audit the cloud storage with very lightweight communication and computation cost. The auditing result not only ensures strong cloud storage correctness guarantee, but also simultaneously achieves fast data error localization, i.e., the identification of misbehaving server. Considering the cloud data are dynamic in nature, the proposed design further supports secure and efficient dynamic operations on outsourced data, including block modification, deletion, and append. Analysis shows the proposed scheme is highly efficient and resilient against Byzantine failure, malicious data modification attack, and even server colluding attacks.

Index Terms—Data integrity, dependable distributed storage, error localization, 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. The increasing network bandwidth and reliable yet flexible network connections make it even possible that users can now subscribe high quality services from data and software that reside solely on remote data centers.

Moving data into the cloud offers great convenience to users since they don't have to care about the complexities of direct hardware management. The pioneer of Cloud Computing vendors, Amazon Simple Storage Service (S3) and Amazon Elastic Compute Cloud (EC2) [2] are both well known examples. While these internet-based online services do provide huge amounts of storage space and customizable computing resources, this computing platform shift, however, is eliminating the responsibility of local machines for data maintenance at the same time. As a result, users are at the mercy of their cloud service providers for the availability and integrity of their data [3]. On the one hand, although the cloud infrastructures are much more powerful and reliable than personal computing devices, broad range of both internal and external threats for data integrity still exist. Examples of outages and data loss incidents of noteworthy cloud storage services appear from time to time [4]–[8]. On the other hand, since users may not retain a local copy of outsourced data, there exist various incentives for cloud service providers (CSP) to behave unfaithfully towards the cloud users regarding the status of their outsourced data. For example, to increase the profit margin by reducing cost, it is possible for CSP to discard rarely accessed data without being detected in a timely fashion [9]. Similarly, CSP may even attempt to hide data loss incidents so as to maintain a reputation [10]–[12]. Therefore, although outsourcing data into the cloud is economically attractive for the cost and complexity of long-term large-scale data storage, its lacking of offering strong assurance of data integrity and availability may impede its wide adoption by both enterprise and individual cloud users.

In order to achieve the assurances of cloud data integrity and availability and enforce the quality of cloud storage service, efficient methods that enable on-demand data correctness verification on behalf of cloud users have to be designed. However, the fact that users no longer have physical possession of data in the cloud prohibits the direct adoption of traditional cryptographic primitives for the purpose of data integrity protection. Hence, the verification of cloud storage correctness must be conducted without explicit knowledge of the whole data files [9]–[12]. Meanwhile, cloud storage is not just a third party data warehouse. The data stored in the cloud may not only be accessed but also be frequently updated by the users [13]–[15], including insertion, deletion, modification, appending, etc. Thus, it is also imperative to support the integration of this dynamic feature into the

- *Cong Wang, Qian Wang, and Kui Ren are with the Department of Electrical and Computer Engineering, Illinois Institute of Technology, Chicago, IL 60616. E-mail: {cong,qian,kren}@ece.iit.edu.*
- *Ning Cao and Wenjing Lou are with the Department of Electrical and Computer Engineering, Worcester Polytechnic Institute, Worcester, MA 01609. E-mail: {ncao,wjlou}@ece.wpi.edu.*

A preliminary version [1] of this paper was presented at the 17th IEEE International Workshop on Quality of Service (IWQoS'09).

cloud storage correctness assurance, which makes the system design even more challenging. Last but not the least, the deployment of Cloud Computing is powered by data centers running in a simultaneous, cooperated and distributed manner [3]. It is more advantages for individual users to store their data redundantly across multiple physical servers so as to reduce the data integrity and availability threats. Thus, distributed protocols for storage correctness assurance will be of most importance in achieving robust and secure cloud storage systems. However, such important area remains to be fully explored in the literature.

Recently, the importance of ensuring the remote data integrity has been highlighted by the following research works under different system and security models [9]–[19]. These techniques, while can be useful to ensure the storage correctness without having users possessing local data, are all focusing on single server scenario. They may be useful for quality-of-service testing [20], but does not guarantee the data availability in case of server failures. Although direct applying these techniques to distributed storage (multiple servers) could be straightforward, the resulted storage verification overhead would be linear to the number of servers. As an complementary approach, researchers have also proposed distributed protocols [20]–[22] for ensuring storage correctness across multiple servers or peers. However, while providing efficient cross server storage verification and data availability insurance, these schemes are all focusing on static or archival data. As a result, their capabilities of handling dynamic data remains unclear, which inevitably limits their full applicability in cloud storage scenarios.

In this paper, we propose an effective and flexible distributed storage verification scheme with explicit dynamic data support to ensure the correctness and availability of users' data in the cloud. We rely on erasure-correcting code in the file distribution preparation to provide redundancies and guarantee the data dependability against Byzantine servers [23], where a storage server may fail in arbitrary ways. This construction drastically reduces the communication and storage overhead as compared to the traditional replication-based file distribution techniques. By utilizing the homomorphic token with distributed verification of erasure-coded data, our scheme achieves the storage correctness insurance as well as data error localization: whenever data corruption has been detected during the storage correctness verification, our scheme can almost guarantee the simultaneous localization of data errors, i.e., the identification of the misbehaving server(s). In order to strike a good balance between error resilience and data dynamics, we further explore the algebraic property of our token computation and erasure-coded data, and demonstrate how to efficiently support dynamic operation on data blocks, while maintaining the same level of storage correctness assurance. In order to save the time, computation resources, and even the related online burden of users, we also provide the extension of the proposed main scheme to support third-party auditing, where users can safely delegate the integrity checking tasks to third-party auditors and be worry-free to use the cloud storage services. Our work is among the first few ones in this field to consider distributed data storage security in Cloud Computing. Our contribution can be summarized as the following three aspects:

1) Compared to many of its predecessors, which only provide binary results about the storage status across the distributed servers, the proposed scheme achieves the integration of storage correctness insurance and data error localization, i.e., the identification of misbehaving server(s).

2) Unlike most prior works for ensuring remote data integrity, the new scheme further supports secure and efficient dynamic operations on data blocks, including: update, delete and append.

3) The experiment results demonstrate the proposed scheme is highly efficient. Extensive security analysis shows our scheme is resilient against Byzantine failure, malicious data modification attack, and even server colluding attacks.

The rest of the paper is organized as follows. Section II introduces the system model, adversary model, our design goal and notations. Then we provide the detailed description of our scheme in Section III and IV. Section V gives the security analysis and performance evaluations, followed by Section VI which overviews the related work. Finally, Section VII concludes the whole paper.

2 Problem Statement

2.1 System Model

A representative network architecture for cloud storage service architecture is illustrated in Figure 1. Three different network entities can be identified as follows:

- User: an entity, who has data to be stored in the cloud and relies on the cloud for data storage and computation, can be either enterprise or individual customers.
- Cloud Server (CS): an entity, which is managed by *cloud service provider* (CSP) to provide data storage service and has significant storage space and computation resources (we will not differentiate CS and CSP hereafter.).
- Third Party Auditor (TPA): an optional TPA, who has expertise and capabilities that users may not have, is trusted to assess and expose risk of cloud storage services on behalf of the users upon request.

In cloud data storage, a user stores his data through a CSP into a set of cloud servers, which are running in a simultaneous, cooperated and distributed manner. Data redundancy can be employed with technique of erasure-correcting code to further tolerate faults or server crash as user's data grows in size and importance. Thereafter, for application purposes, the user interacts with the

cloud servers via CSP to access or retrieve his data. In some cases, the user may need to perform block level operations on his data. The most general forms of these operations we are considering are block update, delete, insert and append. Note that in this paper, we put more focus on the support of file-oriented cloud applications other than non-file application data, such as social networking data. In other words, the cloud data we are considering is not expected to be rapidly changing in a relative short period.

As users no longer possess their data locally, it is of critical importance to ensure users that their data are being correctly stored and maintained. That is, users should be equipped with security means so that they can make continuous correctness assurance (to enforce cloud storage service-level agreement) of their stored data even without the existence of local copies. In case that users do not necessarily have the time, feasibility or resources to monitor their data online, they can delegate the data auditing tasks to an optional trusted TPA of their respective choices. However, to securely introduce such a TPA, any possible leakage of user's outsourced data towards TPA through the auditing protocol should be prohibited.

In our model, we assume that the point-to-point communication channels between each cloud server and the user is authenticated and reliable, which can be achieved in practice with little overhead. These authentication handshakes are omitted in the following presentation.

2.2 Adversary Model

From user's perspective, the adversary model has to capture all kinds of threats towards his cloud data integrity. Because cloud data do not reside at user's local site but at CSP's address domain, these threats can come from two different sources: internal and external attacks. For internal attacks, a CSP can be self-interested, untrusted and possibly malicious. Not only does it desire to move data that has not been or is rarely accessed to a lower tier of storage than agreed for monetary reasons, but it may also attempt to hide a data loss incident due to management errors, Byzantine failures and so on. For external attacks, data integrity threats may come from outsiders who are beyond the control domain of CSP, for example, the economically motivated attackers. They may compromise a number of cloud data storage servers in different time intervals and subsequently be able to modify or delete users' data while remaining undetected by CSP.

Therefore, we consider the adversary in our model has the following capabilities, which captures both external and internal threats towards the cloud data integrity. Specifically, the adversary is interested in continuously corrupting the user's data files stored on individual servers. Once a server is comprised, an adversary can pollute the original data files by modifying or introducing its own fraudulent data to prevent the original data from being retrieved by the user. This corresponds to the threats from external attacks. In the worst case scenario, the adversary can compromise all the storage servers so that he can intentionally modify the data files as long as they are internally consistent. In fact, this is equivalent to internal attack case where all servers are assumed colluding together from the early stages of application or service deployment to hide a data loss or corruption incident.

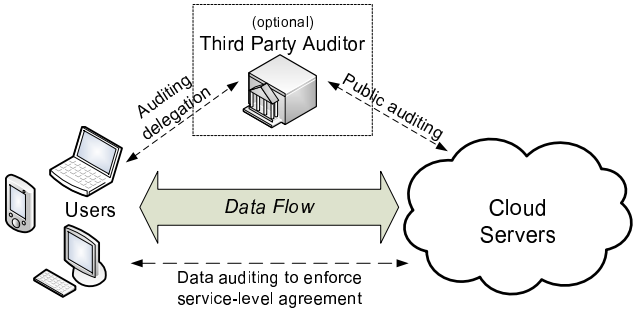


Fig. 1: Cloud storage service architecture

2.3 Design Goals

To ensure the security and dependability for cloud data storage under the aforementioned adversary model, we aim to design efficient mechanisms for dynamic data verification and operation and achieve the following goals: (1) Storage correctness: to ensure users that their data are indeed stored appropriately and kept intact all the time in the cloud. (2) Fast localization of data error: to effectively locate the malfunctioning server when data corruption has been detected. (3) Dynamic data support: to maintain the same level of storage correctness assurance even if users modify, delete or append their data files in the cloud. (4) Dependability: to enhance data availability against Byzantine failures, malicious data modification and server colluding attacks, i.e. minimizing the effect brought by data errors or server failures. (5) Lightweight: to enable users to perform storage correctness checks with minimum overhead.

2.4 Notation and Preliminaries

- **F** – the data file to be stored. We assume that **F** can be denoted as a matrix of m equal-sized data vectors, each consisting of l blocks. Data blocks are all well represented as elements in Galois Field $GF(2^p)$ for $p = 8$ or 16.
- **A** – The dispersal matrix used for Reed-Solomon coding.
- **G** – The encoded file matrix, which includes a set of $n = m + k$ vectors, each consisting of l blocks.
- $f_{key}(\cdot)$ – pseudorandom function (PRF), which is defined as $f : \{0,1\}^* \times key \rightarrow GF(2^p)$.
- $\phi_{key}(\cdot)$ – pseudorandom permutation (PRP), which is defined as $\phi : \{0,1\}^{\log_2(\ell)} \times key \rightarrow \{0,1\}^{\log_2(\ell)}$.

- ver – a version number bound with the index for individual blocks, which records the times the block has been modified. Initially we assume ver is 0 for all data blocks.
- s_{ij}^{ver} – the seed for PRF, which depends on the file name, block index i, the server position j as well as the optional block version number ver.

3 Ensuring Cloud Data Storage

In cloud data storage system, users store their data in the cloud and no longer possess the data locally. Thus, the correctness and availability of the data files being stored on the distributed cloud servers must be guaranteed. One of the key issues is to effectively detect any unauthorized data modification and corruption, possibly due to server compromise and/or random Byzantine failures. Besides, in the distributed case when such inconsistencies are successfully detected, to find which server the data error lies in is also of great significance, since it can always be the first step to fast recover the storage errors and/or identifying potential threats of external attacks.

To address these problems, our main scheme for ensuring cloud data storage is presented in this section. The first part of the section is devoted to a review of basic tools from coding theory that is needed in our scheme for file distribution across cloud servers. Then, the homomorphic token is introduced. The token computation function we are considering belongs to a family of universal hash function [24], chosen to preserve the homomorphic properties, which can be perfectly integrated with the verification of erasure-coded data [21] [25]. Subsequently, it is shown how to derive a challenge-response protocol for verifying the storage correctness as well as identifying misbehaving servers. The procedure for file retrieval and error recovery based on erasure-correcting code is also outlined. Finally, we describe how to extend our scheme to third party auditing with only slight modification of the main design.

3.1 File Distribution Preparation

It is well known that erasure-correcting code may be used to tolerate multiple failures in distributed storage systems. In cloud data storage, we rely on this technique to disperse the data file $\mathbf{F}$ redundantly across a set of $n = m + k$ distributed servers. An (m, k) Reed-Solomon erasure-correcting 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, with a space overhead of k/m. For support of efficient sequential I/O to the original file, our file layout is systematic, i.e., the unmodified m data file vectors together with k parity vectors is distributed across $m + k$ different servers.

Let $\mathbf{F} = (F_1, F_2, \ldots, F_m)$ and $F_i = (f_{1i}, f_{2i}, \ldots, f_{li})^T$ $(i \in \{1, \ldots, m\})$. Here T (shorthand for transpose) denotes that each F_i is represented as a column vector, and l denotes data vector size in blocks. All these blocks are elements of $GF(2^p)$. The systematic layout with parity vectors is achieved with the information dispersal matrix $\mathbf{A}$, derived from an $m\times(m+k)$ Vandermonde matrix [26]:

$$\begin{pmatrix} 1 & 1 & \ldots & 1 & 1 & \ldots & 1 \\ \beta_1 & \beta_2 & \ldots & \beta_m & \beta_{m+1} & \ldots & \beta_n \\ \vdots & \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ \beta_1^{m-1} & \beta_2^{m-1} & \ldots & \beta_m^{m-1} & \beta_{m+1}^{m-1} & \ldots & \beta_n^{m-1} \end{pmatrix},$$

where β_j $(j \in \{1, \ldots, n\})$ are distinct elements randomly picked from $GF(2^p)$.

After a sequence of elementary row transformations, the desired matrix $\mathbf{A}$ can be written as

$$\mathbf{A} = (\mathbf{I}|\mathbf{P}) = \begin{pmatrix} 1 & 0 & \ldots & 0 & p_{11} & p_{12} & \ldots & p_{1k} \\ 0 & 1 & \ldots & 0 & p_{21} & p_{22} & \ldots & p_{2k} \\ \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \ldots & 1 & p_{m1} & p_{m2} & \ldots & p_{mk} \end{pmatrix},$$

where $\mathbf{I}$ is a $m \times m$ identity matrix and $\mathbf{P}$ is the secret parity generation matrix with size $m \times k$. Note that $\mathbf{A}$ is derived from a Vandermonde matrix, thus it has the property that any m out of the $m + k$ columns form an invertible matrix.

By multiplying $\mathbf{F}$ by $\mathbf{A}$, the user obtains the encoded file:

$$\begin{aligned} \mathbf{G} = \mathbf{F} \cdot \mathbf{A} &= (G^{(1)}, G^{(2)}, \ldots, G^{(m)}, G^{(m+1)}, \ldots, G^{(n)}) \\ &= (F_1, F_2, \ldots, F_m, G^{(m+1)}, \ldots, G^{(n)}), \end{aligned}$$

where $G^{(j)} = (g_1^{(j)}, g_2^{(j)}, \ldots, g_l^{(j)})^T$ $(j \in \{1, \ldots, n\})$. As noticed, the multiplication reproduces the original data file vectors of $\mathbf{F}$ and the remaining part $(G^{(m+1)}, \ldots, G^{(n)})$ are k parity vectors generated based on $\mathbf{F}$.

3.2 Challenge Token Pre-computation

In order to achieve assurance of data storage correctness and data error localization simultaneously, our scheme entirely relies on the pre-computed verification tokens. The main idea is as follows: before file distribution the user pre-computes a certain number of short verification tokens on individual vector $G^{(j)}$ $(j \in \{1, \ldots, n\})$, each token covering a random subset of data blocks. Later, when the user wants to make sure the storage correctness for the data in the cloud, he challenges the cloud servers with a set of randomly generated block indices. Upon receiving challenge, each cloud server computes a short "signature" over the specified blocks and returns them to the user. The values of these signatures should match the corresponding tokens pre-computed by the user. Meanwhile, as all servers operate over the same subset of the indices, the requested response values for integrity check must also be a valid codeword determined by secret matrix $\mathbf{P}$.

Algorithm 1 Token Pre-computation

procedure
  Choose parameters l, n and function f, ϕ;
  Choose the number t of tokens;
  Choose the number r of indices per verification;
  Generate master key K_{PRP} and challenge key k_{chal};
  for vector $G^{(j)}$, $j \leftarrow 1$, n **do**
    for round $i \leftarrow 1$, t **do**
      Derive $\alpha_i = f_{k_{chal}}(i)$ and $k_{prp}^{(i)}$ from K_{PRP}.
      Compute $v_i^{(j)} = \sum_{q=1}^{r} \alpha_i^q * G^{(j)}[\phi_{k_{prp}^{(i)}}(q)]$
    end for
  end for
  Store all the v_i's locally.
end procedure

Suppose the user wants to challenge the cloud servers t times to ensure the correctness of data storage. Then, he must pre-compute t verification tokens for each $G^{(j)}$ ($j \in \{1, \ldots, n\}$), using a PRF $f(\cdot)$, a PRP $\phi(\cdot)$, a challenge key k_{chal} and a master permutation key K_{PRP}. Specifically, to generate the i^{th} token for server j, the user acts as follows:

1) Derive a random challenge value α_i of $GF(2^p)$ by $\alpha_i = f_{k_{chal}}(i)$ and a permutation key $k_{prp}^{(i)}$ based on K_{PRP}.
2) Compute the set of r randomly-chosen indices:
$$\{I_q \in [1, ..., l] | 1 \le q \le r\}, \text{where } I_q = \phi_{k_{prp}^{(i)}}(q).$$
3) Calculate the token as:
$$v_i^{(j)} = \sum_{q=1}^{r} \alpha_i^q * G^{(j)}[I_q], \text{where } G^{(j)}[I_q] = g_{I_q}^{(j)}.$$

Note that $v_i^{(j)}$, which is an element of $GF(2^p)$ with small size, is the response the user expects to receive from server j when he challenges it on the specified data blocks.

After token generation, the user has the choice of either keeping the pre-computed tokens locally or storing them in encrypted form on the cloud servers. In our case here, the user stores them locally to obviate the need for encryption and lower the bandwidth overhead during dynamic data operation which will be discussed shortly. The details of token generation are shown in Algorithm 1.

Once all tokens are computed, the final step before file distribution is to blind each parity block $g_i^{(j)}$ in $(G^{(m+1)}, \ldots, G^{(n)})$ by
$$g_i^{(j)} \leftarrow g_i^{(j)} + f_{k_j}(s_{ij}), i \in \{1, \ldots, l\},$$
where k_j is the secret key for parity vector $G^{(j)}$ ($j \in \{m+1, \ldots, n\}$). This is for protection of the secret matrix $\mathbf{P}$. We will discuss the necessity of using blinded parities in detail in Section 5.2. After blinding the parity information, the user disperses all the n encoded vectors $G^{(j)}$ ($j \in \{1, \ldots, n\}$) across the cloud servers $S_1, S_2, \ldots, S_n$.

3.3 Correctness Verification and Error Localization

Error localization is a key prerequisite for eliminating errors in storage systems. It is also of critical importance to identify potential threats from external attacks. However, many previous schemes [20], [21] do not explicitly consider the problem of data error localization, thus only providing binary results for the storage verification. Our scheme outperforms those by integrating the correctness verification and error localization (misbehaving server identification) in our challenge-response protocol: the response values from servers for each challenge not only determine the correctness of the distributed storage, but also contain information to locate potential data error(s).

Specifically, the procedure of the i-th challenge-response for a cross-check over the n servers is described as follows:

1) The user reveals the α_i as well as the i-th permutation key $k_{prp}^{(i)}$ to each servers.
2) The server storing vector $G^{(j)}$ ($j \in \{1, \ldots, n\}$) aggregates those r rows specified by index $k_{prp}^{(i)}$ into a linear combination
$$R_i^{(j)} = \sum_{q=1}^{r} \alpha_i^q * G^{(j)}[\phi_{k_{prp}^{(i)}}(q)].$$
and send back $R_i^{(j)}$ ($j \in \{1, \ldots, n\}$).
3) Upon receiving $R_i^{(j)}$'s from all the servers, the user takes away blind values in $R^{(j)}$ ($j \in \{m+1, \ldots, n\}$) by
$$R_i^{(j)} \leftarrow R_i^{(j)} - \sum_{q=1}^{r} f_{k_j}(s_{I_q,j}) \cdot \alpha_i^q, \text{where } I_q = \phi_{k_{prp}^{(i)}}(q).$$
4) Then the user verifies whether the received values remain a valid codeword determined by secret matrix $\mathbf{P}$:
$$(R_i^{(1)}, \ldots, R_i^{(m)}) \cdot \mathbf{P} \stackrel{?}{=} (R_i^{(m+1)}, \ldots, R_i^{(n)}).$$

Because all the servers operate over the same subset of indices, the linear aggregation of these r specified rows $(R_i^{(1)}, \ldots, R_i^{(n)})$ has to be a codeword in the encoded file matrix (See Section 5.1 for the correctness analysis). If the above equation holds, the challenge is passed. Otherwise, it indicates that among those specified rows, there exist file block corruptions.

Once the inconsistency among the storage has been successfully detected, we can rely on the pre-computed verification tokens to further determine where the potential data error(s) lies in. Note that each response $R_i^{(j)}$ is computed exactly in the same way as token $v_i^{(j)}$, thus the user can simply find which server is misbehaving by verifying the following n equations:
$$R_i^{(j)} \stackrel{?}{=} v_i^{(j)}, j \in \{1, \ldots, n\}.$$

Algorithm 2 Correctness Verification and Error Localization

procedure CHALLENGE(i)
  Recompute $\alpha_i = f_{k_{chal}}(i)$ and $k_{prp}^{(i)}$ from K_{PRP};
  Send $\{\alpha_i, k_{prp}^{(i)}\}$ to all the cloud servers;
  Receive from servers:
  $\{R_i^{(j)} = \sum_{q=1}^{r} \alpha_i^q * G^{(j)}[\phi_{k_{prp}^{(i)}}(q)] | 1 \le j \le n\}$
  for ($j \leftarrow m+1$, n) **do**
    $R^{(j)} \leftarrow R^{(j)} - \sum_{q=1}^{r} f_{k_j}(s_{I_q,j}) \cdot \alpha_i^q$, $I_q = \phi_{k_{prp}^{(i)}}(q)$
  end for
  if $((R_i^{(1)}, \ldots, R_i^{(m)}) \cdot \mathbf{P} == (R_i^{(m+1)}, \ldots, R_i^{(n)}))$ **then**
    Accept and ready for the next challenge.
  else
    for ($j \leftarrow 1$, n) **do**
      if $(R_i^{(j)} != v_i^{(j)})$ **then**
        return server j is misbehaving.
      end if
    end for
  end if
end procedure

Algorithm 2 gives the details of correctness verification and error localization.

Discussion. Previous work [20], [21] has suggested using the decoding capability of error-correction code to treat data errors. But such approach imposes a bound on the number of misbehaving servers b by $b \le \lfloor k/2 \rfloor$. Namely, they cannot identify misbehaving servers when $b > \lfloor k/2 \rfloor$[1]. However, our token based approach, while allowing efficient storage correctness validation, does not have this limitation on the number of misbehaving servers. That is, our approach can identify any number of misbehaving servers for $b \le (m+k)$. Also note that, for every challenge, each server only needs to send back an aggregated value over the specified set of blocks. Thus the bandwidth cost of our approach is much less than the straightforward approaches that require downloading all the challenged data.

3.4 File Retrieval and Error Recovery

Since our layout of file matrix is systematic, the user can reconstruct the original file by downloading the data vectors from the first m servers, assuming that they return the correct response values. Notice that our verification scheme is based on random spot-checking, so the storage correctness assurance is a probabilistic one. However, by choosing system parameters $(e.g., r, l, t)$ appropriately and conducting enough times of verification, we can guarantee the successful file retrieval with high probability. On the other hand, whenever the data corruption is detected, the comparison of pre-computed tokens and received response values can guarantee the identification of misbehaving server(s) (again with high probability), which will be discussed shortly. Therefore, the user can always ask servers to send back blocks of the r rows specified in the challenge and regenerate the correct blocks by erasure correction, shown in Algorithm 3, as long as the number of identified misbehaving servers is less than k. (otherwise, there is no way to recover the corrupted blocks due to lack of redundancy, even if we know the position of misbehaving servers.) The newly recovered blocks can then be redistributed to the misbehaving servers to maintain the correctness of storage.

1. In [20], the authors also suggest using brute-force decoding when their dispersal code is an erasure code. However, such brute-force method is asymptotically inefficient, and still cannot guarantee identification of all misbehaving servers.

Algorithm 3 Error Recovery

procedure
  % Assume the block corruptions have been detected among
  % the specified r rows;
  % Assume $s \le k$ servers have been identified misbehaving
  Download r rows of blocks from servers;
  Treat s servers as erasures and recover the blocks.
  Resend the recovered blocks to corresponding servers.
end procedure

3.5 Towards Third Party Auditing

As discussed in our architecture, in case the user does not have the time, feasibility or resources to perform the storage correctness verification, he can optionally delegate this task to an independent third party auditor, making the cloud storage publicly verifiable. However, as pointed out by the recent work [27], [28], to securely introduce an effective TPA, the auditing process should bring in no new vulnerabilities towards user data privacy. Namely, TPA should not learn user's data content through the delegated data auditing. Now we show that with only slight modification, our protocol can support privacy-preserving third party auditing.

The new design is based on the observation of linear property of the parity vector blinding process. Recall that the reason of blinding process is for protection of the secret matrix $\mathbf{P}$ against cloud servers. However, this can be achieved either by blinding the parity vector or by blinding the data vector (we assume $k < m$). Thus, if we blind data vector before file distribution encoding, then the storage verification task can be successfully delegated to third party auditing in a privacy-preserving manner. Specifically, the new protocol is described as follows:

1) Before file distribution, the user blinds each file block data $g_i^{(j)}$ in $(G^{(1)}, \ldots, G^{(m)})$ by $g_i^{(j)} \leftarrow g_i^{(j)} + f_{k_j}(s_{ij}), i \in \{1, \ldots, l\}$, where k_j is the secret key for data vector $G^{(j)}$ $(j \in \{1, \ldots, m\})$.
2) Based on the blinded data vector $(G^{(1)}, \ldots, G^{(m)})$, the user generates k parity vectors

$(G^{(m+1)}, \ldots, G^{(n)})$ via the secret matrix $\mathbf{P}$.

3) The user calculates the i^{th} token for server j as previous scheme: $v_i^{(j)} = \sum_{q=1}^{r} \alpha_i^q * G^{(j)}[I_q]$, where $G^{(j)}[I_q] = g_{I_q}^{(j)}$ and $\alpha_i = f_{k_{chal}}(i) \in GF(2^p)$.
4) The user sends the token set $\{v_i^{(j)}\}_{\{1 \le i \le t, 1 \le j \le n\}}$, secret matrix $\mathbf{P}$, permutation and challenge key K_{PRP} and k_{chal} to TPA for auditing delegation.

The correctness validation and misbehaving server identification for TPA is just similar to the previous scheme. The only difference is that TPA does not have to take away the blinding values in the servers' response $R^{(j)}$ $(j \in \{1, \ldots, n\})$ but verifies directly. As TPA does not know the secret blinding key k_j $(j \in \{1, \ldots, m\})$, there is no way for TPA to learn the data content information during auditing process. Therefore, the privacy-preserving third party auditing is achieved. Note that compared to previous scheme, we only change the sequence of file encoding, token pre-computation, and blinding. Thus, the overall computation overhead and communication overhead remains roughly the same.

4 Providing Dynamic Data Operation Support

So far, we assumed that $\mathbf{F}$ represents static or archived data. This model may fit some application scenarios, such as libraries and scientific datasets. However, in cloud data storage, there are many potential scenarios where data stored in the cloud is dynamic, like electronic documents, photos, or log files etc. Therefore, it is crucial to consider the dynamic case, where a user may wish to perform various block-level operations of update, delete and append to modify the data file while maintaining the storage correctness assurance.

Since data do not reside at users' local site but at cloud service provider's address domain, supporting dynamic data operation can be quite challenging. On the one hand, CSP needs to process the data dynamics request without knowing the secret keying material. On the other hand, users need to ensure that all the dynamic data operation request has been faithfully processed by CSP. To address this problem, we briefly explain our approach methodology here and provide the details later. For any data dynamic operation, the user must first generate the corresponding resulted file blocks and parities. This part of operation has to be carried out by the user, since only he knows the secret matrix $\mathbf{P}$. Besides, to ensure the changes of data blocks correctly reflected in the cloud address domain, the user also needs to modify the corresponding storage verification tokens to accommodate the changes on data blocks. Only with the accordingly changed storage verification tokens, the previously discussed challenge-response protocol can be carried on successfully even after data dynamics. In other words, these verification tokens help ensure that CSP would correctly execute the processing of any dynamic data operation request. Otherwise, CSP would be caught cheating with high probability in the protocol execution later on. Given this design methodology, the straightforward and trivial way to support these operations is for user to download all the data from the cloud servers and re-compute the whole parity blocks as well as verification tokens. This would clearly be highly inefficient. In this section, we will show how our scheme can explicitly and efficiently handle dynamic data operations for cloud data storage, by utilizing the linear property of Reed-Solomon code and verification token construction.

4.1 Update Operation

In cloud data storage, a user may need to modify some data block(s) stored in the cloud, from its current value f_{ij} to a new one, $f_{ij} + \Delta f_{ij}$. We refer this operation as data update. Figure 2 gives the high level logical representation of data block update. Due to the linear property of Reed-Solomon code, a user can perform the update operation and generate the updated parity blocks by using Δf_{ij} only, without involving any other unchanged blocks. Specifically, the user can construct a general update matrix $\Delta\mathbf{F}$ as

$$\begin{aligned} \Delta\mathbf{F} &= \begin{pmatrix} \Delta f_{11} & \Delta f_{12} & \ldots & \Delta f_{1m} \\ \Delta f_{21} & \Delta f_{22} & \ldots & \Delta f_{2m} \\ \vdots & \vdots & \ddots & \vdots \\ \Delta f_{l1} & \Delta f_{l2} & \ldots & \Delta f_{lm} \end{pmatrix} \\ &= (\Delta F_1, \Delta F_2, \ldots, \Delta F_m). \end{aligned}$$

Note that we use zero elements in $\Delta\mathbf{F}$ to denote the unchanged blocks and thus $\Delta\mathbf{F}$ should only be a sparse matrix most of the time (we assume for certain time epoch, the user only updates a relatively small part of file $\mathbf{F}$). To maintain the corresponding parity vectors as well as be consistent with the original file layout, the user can multiply $\Delta\mathbf{F}$ by $\mathbf{A}$ and thus generate the update information for both the data vectors and parity vectors as follows:

$$\begin{aligned} \Delta\mathbf{F} \cdot \mathbf{A} &= (\Delta G^{(1)}, \ldots, \Delta G^{(m)}, \Delta G^{(m+1)}, \ldots, \Delta G^{(n)}) \\ &= (\Delta F_1, \ldots, \Delta F_m, \Delta G^{(m+1)}, \ldots, \Delta G^{(n)}), \end{aligned}$$

where $\Delta G^{(j)}$ $(j \in \{m+1, \ldots, n\})$ denotes the update information for the parity vector $G^{(j)}$.

Because the data update operation inevitably affects some or all of the remaining verification tokens, after preparation of update information, the user has to amend those unused tokens for each vector $G^{(j)}$ to maintain the same storage correctness assurance. In other words, for all the unused tokens, the user needs to exclude every occurrence of the old data block and replace it with the new one. Thanks to the homomorphic construction of our verification token, the user can perform the token update efficiently. To give more details, suppose a block $G^{(j)}[I_s]$, which is covered by the specific token $v_i^{(j)}$, has been changed to $G^{(j)}[I_s] + \Delta G^{(j)}[I_s]$, where $I_s = \phi_{k_{prp}^{(i)}}(s)$. To maintain the usability of token

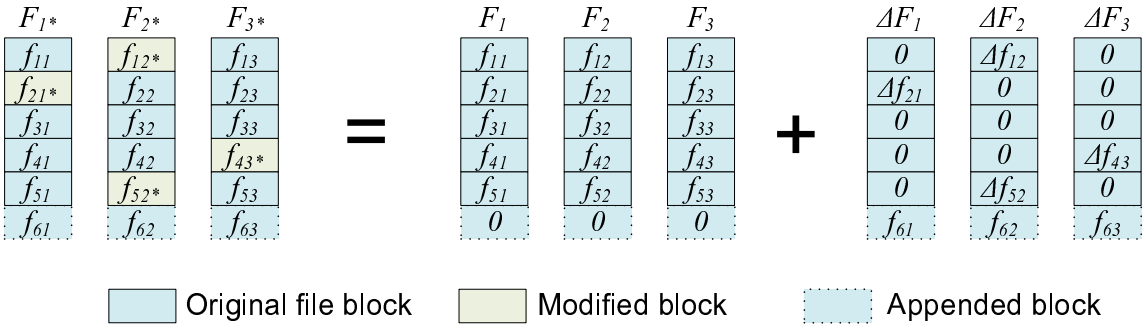


Fig. 2: Logical representation of data dynamics, including block update, append and delete.

$v_i^{(j)}$, it is not hard to verify that the user can simply update it by $v_i^{(j)} \leftarrow v_i^{(j)} + \alpha_i^s * \Delta G^{(j)}[I_s]$, without retrieving other $r-1$ blocks required in the pre-computation of $v_i^{(j)}$.

After the amendment to the affected tokens[2], the user needs to blind the update information $\Delta g_i^{(j)}$ for each parity block in $(\Delta G^{(m+1)}, \ldots, \Delta G^{(n)})$ to hide the secret matrix $\mathbf{P}$ by $\Delta g_i^{(j)} \leftarrow \Delta g_i^{(j)} + f_{k_j}(s_{ij}^{ver}), i \in \{1, \ldots, l\}$. Here we use a new seed s_{ij}^{ver} for the PRF. The version number ver functions like a counter which helps the user to keep track of the blind information on the specific parity blocks. After blinding, the user sends update information to the cloud servers, which perform the update operation as $G^{(j)} \leftarrow G^{(j)} + \Delta G^{(j)}, \ (j \in \{1, \ldots, n\})$.

Discussion. Note that by using the new seed s_{ij}^{ver} for the PRF functions every time (for a block update operation), we can ensure the freshness of the random value embedded into parity blocks. In other words, the cloud servers cannot simply abstract away the random blinding information on parity blocks by subtracting the old and newly updated parity blocks. As a result, the secret matrix $\mathbf{P}$ is still being well protected, and the guarantee of storage correctness remains.

4.2 Delete Operation

Sometimes, after being stored in the cloud, certain data blocks may need to be deleted. The delete operation we are considering is a general one, in which user replaces the data block with zero or some special reserved data symbol. From this point of view, the delete operation is actually a special case of the data update operation, where the original data blocks can be replaced with zeros or some predetermined special blocks. Therefore, we can rely on the update procedure to support delete operation, i.e., by setting Δf_{ij} in ΔF to be $-\Delta f_{ij}$. Also, all the affected tokens have to be modified and the updated parity information has to be blinded using the same method specified in update operation.

2. In practice, it is possible that only a fraction of tokens need amendment, since the updated blocks may not be covered by all the tokens.

4.3 Append Operation

In some cases, the user may want to increase the size of his stored data by adding blocks at the end of the data file, which we refer as data append. We anticipate that the most frequent append operation in cloud data storage is bulk append, in which the user needs to upload a large number of blocks (not a single block) at one time.

Given the file matrix $\mathbf{F}$ illustrated in file distribution preparation, appending blocks towards the end of a data file is equivalent to concatenate corresponding rows at the bottom of the matrix layout for file $\mathbf{F}$ (See Figure 2). In the beginning, there are only l rows in the file matrix. To simplify the presentation, we suppose the user wants to append m blocks at the end of file $\mathbf{F}$, denoted as $(f_{l+1,1}, f_{l+1,2}, ..., f_{l+1,m})$ (We can always use zero-padding to make a row of m elements.). With the secret matrix $\mathbf{P}$, the user can directly calculate the append blocks for each parity server as $(f_{l+1,1}, ..., f_{l+1,m}) \cdot \mathbf{P} = (g_{l+1}^{(m+1)}, ..., g_{l+1}^{(n)})$.

To ensure the newly appended blocks are covered by our challenge tokens, we need a slight modification to our token pre-computation. Specifically, we require the user to expect the maximum size in blocks, denoted as l_{max}, for each of his data vector. This idea of supporting block append was first suggested by [13] in a single server setting, and it relies on both the initial budget for the maximum anticipated data size l_{max} in each encoded data vector and the system parameter $r_{max} = \lceil r * (l_{max}/l) \rceil$ for each pre-computed challenge-response token. The pre-computation of the i-th token on server j is modified as follows: $v_i = \sum_{q=1}^{r_{max}} \alpha_i^q * G^{(j)}[I_q]$, where

$$G^{(j)}[I_q] = \begin{cases} G^{(j)}[\phi_{k_{prp}^{(i)}}(q)] & , [\phi_{k_{prp}^{(i)}}(q)] \leq l \\ 0 & , [\phi_{k_{prp}^{(i)}}(q)] > l \end{cases},$$

and the PRP $\phi(\cdot)$ is modified as: $\phi(\cdot) : \{0,1\}^{\log_2(l_{max})} \times key \rightarrow \{0,1\}^{\log_2(l_{max})}$. This formula guarantees that on average, there will be r indices falling into the range of existing l blocks. Because the cloud servers and the user have the agreement on the number of existing blocks in each vector $G^{(j)}$, servers will follow exactly the above procedure when re-computing the token values upon receiving user's challenge request.

Now when the user is ready to append new blocks, i.e., both the file blocks and the corresponding parity blocks are generated, the total length of each vector $G^{(j)}$ will be increased and fall into the range $[l, l_{max}]$. Therefore, the user will update those affected tokens by adding $\alpha_i^s * G^{(j)}[I_s]$ to the old v_i whenever $G^{(j)}[I_s] \neq 0$ for $I_s > l$, where $I_s = \phi_{k_{prp}^{(i)}}(s)$. The parity blinding is similar as introduced in update operation, and thus is omitted here.

4.4 Insert Operation

An insert operation to the data file refers to an append operation at the desired index position while maintaining the same data block structure for the whole data file, i.e., inserting a block $F[j]$ corresponds to shifting all blocks starting with index $j+1$ by one slot. Thus, an insert operation may affect many rows in the logical data file matrix $\mathbf{F}$, and a substantial number of computations are required to renumber all the subsequent blocks as well as re-compute the challenge-response tokens. Hence, a direct insert operation is difficult to support.

In order to fully support block insertion operation, recent work [14], [15] suggests utilizing additional data structure (for example, Merkle Hash Tree [29]) to maintain and enforce the block index information. Following this line of research, we can circumvent the dilemma of our block insertion by viewing each insertion as a logical append operation at the end of file $\mathbf{F}$. Specifically, if we also use additional data structure to maintain such logical to physical block index mapping information, then all block insertion can be treated via logical append operation, which can be efficiently supported. On the other hand, using the block index mapping information, the user can still access or retrieve the file as it is. Note that as a tradeoff, the extra data structure information has to be maintained locally on the user side.

5 Security Analysis and Performance Evaluation

In this section, we analyze our proposed scheme in terms of correctness, security and efficiency. Our security analysis focuses on the adversary model defined in Section II. We also evaluate the efficiency of our scheme via implementation of both file distribution preparation and verification token pre-computation.

5.1 Correctness Analysis

First, we analyze the correctness of the verification procedure. Upon obtaining all the response $R_i^{(j)}$s from servers and taking away the random blind values from $R_i^{(j)}$ ($j \in \{m+1, \ldots, n\}$), the user relies on the equation $(R_i^{(1)}, \ldots, R_i^{(m)}) \cdot \mathbf{P} \stackrel{?}{=} (R_i^{(m+1)}, \ldots, R_i^{(n)})$ to ensure the storage correctness. To see why this is true, we can rewrite the equation according to the token computation: $(\sum_{q=1}^{r} \alpha_i^q * g_{I_q}^{(1)}, ..., \sum_{q=1}^{r} \alpha_i^q * g_{I_q}^{(m)}) \cdot \mathbf{P} = (\sum_{q=1}^{r} \alpha_i^q * g_{I_q}^{(m+1)}, ..., \sum_{q=1}^{r} \alpha_i^q * g_{I_q}^{(n)})$, and hence the left hand side (LHS) of the equation expands as:

$$\text{LHS} = (\alpha_i, \alpha_i^2, \ldots, \alpha_i^r) \begin{pmatrix} g_{I_1}^{(1)} & g_{I_1}^{(2)} & \cdots & g_{I_1}^{(m)} \\ g_{I_2}^{(1)} & g_{I_2}^{(2)} & \cdots & g_{I_2}^{(m)} \\ \vdots & \vdots & \ddots & \vdots \\ g_{I_r}^{(1)} & g_{I_r}^{(2)} & \cdots & g_{I_r}^{(m)} \end{pmatrix} \cdot \mathbf{P}$$

$$= (\alpha_i, \alpha_i^2, \ldots, \alpha_i^r) \begin{pmatrix} g_{I_1}^{(m+1)} & g_{I_1}^{(m+2)} & \cdots & g_{I_1}^{(n)} \\ g_{I_2}^{(m+1)} & g_{I_2}^{(m+2)} & \cdots & g_{I_2}^{(n)} \\ \vdots & \vdots & \ddots & \vdots \\ g_{I_r}^{(m+1)} & g_{I_r}^{(m+2)} & \cdots & g_{I_r}^{(n)} \end{pmatrix},$$

which equals the right hand side as required. Thus, it is clear to show that as long as each server operates on the same specified subset of rows, the above checking equation will always hold.

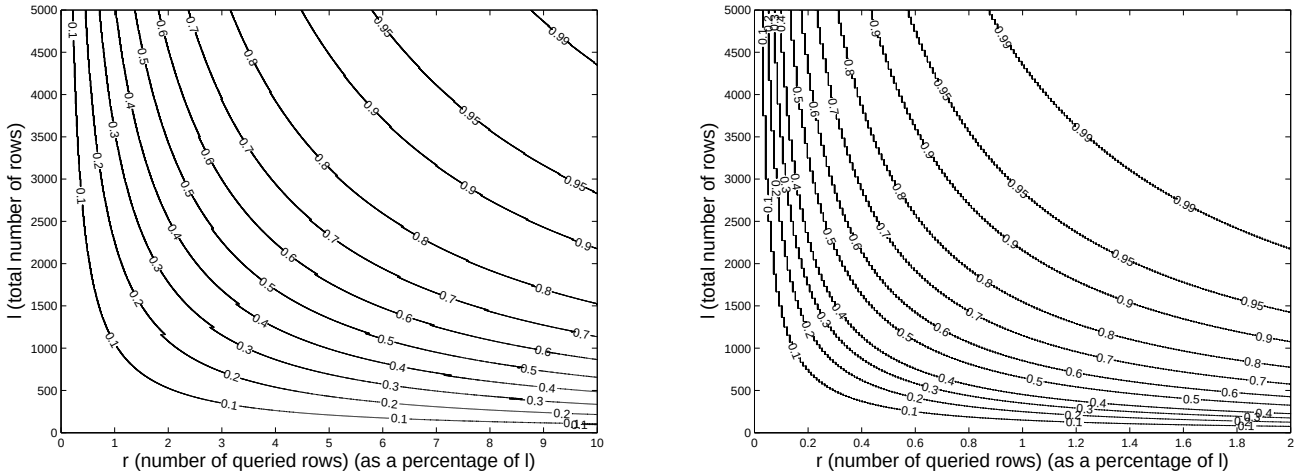
5.2 Security Strength

5.2.1 Detection Probability against Data Modification

In our scheme, servers are required to operate only on specified rows in each challenge-response protocol execution. We will show that this "sampling" strategy on selected rows instead of all can greatly reduce the computational overhead on the server, while maintaining high detection probability for data corruption.

Suppose n_c servers are misbehaving due to the possible compromise or Byzantine failure. In the following analysis, we do not limit the value of n_c, i.e., $n_c \leq n$. Thus, all the analysis results hold even if all the servers are compromised. We will leave the explanation on collusion resistance of our scheme against this worst case scenario in a later subsection. Assume the adversary modifies the data blocks in z rows out of the l rows in the encoded file matrix. Let r be the number of different rows for which the user asks for checking in a challenge. Let X be a discrete random variable that is defined to be the number of rows chosen by the user that matches the rows modified by the adversary. We first analyze the matching probability that at least one of the rows picked by the user matches one of the rows modified by the adversary: $P_m^r = 1 - P\{X = 0\} = 1 - \prod_{i=0}^{r-1}(1 - \min\{\frac{z}{l-i}, 1\}) \geq 1 - (\frac{l-z}{l})^r$. If none of the specified r rows in the i-th verification process are deleted or modified, the adversary avoids the detection.

Next, we study the probability of a false negative result that there exists at least one invalid response calculated from those specified r rows, but the checking equation still holds. Consider the responses $R_i^{(1)}, \ldots, R_i^{(n)}$ returned from the data storage servers for the i-th challenge, each response value $R_i^{(j)}$, calculated within $GF(2^p)$, is based on r blocks on server j. The number of responses $R^{(m+1)}, \ldots, R^{(n)}$ from parity servers is $k = n - m$. Thus, according to the proposition 2 of our previous work in [30], the false negative probability



(a) $z = 1\%$ of l. (b) $z = 10\%$ of l.

Fig. 3: The detection probability P_d against data modification. We show P_d as a function of l (the number of blocks on each cloud storage server) and r (the number of rows queried by the user, shown as a percentage of l) for two values of z (the number of rows modified by the adversary). Both graphs are plotted under $p = 16$, $n_c = 10$ and $k = 5$, but with different scale.

is $P_f^r = Pr_1 + Pr_2$, where $Pr_1 = \frac{(1+2^{-p})^{n_c}-1}{2^{n_c}-1}$ and $Pr_2 = (1 - \mathrm{Pr}_1)(2^{-p})^k$.

Based on above discussion, it follows that the probability of data modification detection across all storage servers is $P_d = P_m^r \cdot (1 - P_f^r)$. Figure 3 plots P_d for different values of l, r, z while we set $p = 16, n_c = 10$ and $k = 5$ [3]. From the figure we can see that if more than a fraction of the data file is corrupted, then it suffices to challenge for a small constant number of rows in order to achieve detection with high probability. For example, if $z = 1\%$ of l, every token only needs to cover 460 indices in order to achieve the detection probability of at least 99%.

5.2.2 Identification Probability for Misbehaving Servers

We have shown that, if the adversary modifies the data blocks among any of the data storage servers, our sampling checking scheme can successfully detect the attack with high probability. As long as the data modification is caught, the user will further determine which server is malfunctioning. This can be achieved by comparing the response values $R_i^{(j)}$ with the pre-stored tokens $v_i^{(j)}$, where $j \in \{1, \ldots, n\}$. The probability for error localization or identifying misbehaving server(s) can be computed in a similar way. It is the product of the matching probability for sampling check and the probability of complementary event for the false negative result. Obviously, the matching probability is $\widehat{P}_m^r = 1 - \prod_{i=0}^{r-1}(1 - \min\{\frac{\hat{z}}{l-i}, 1\})$, where $\hat{z} \leq z$.

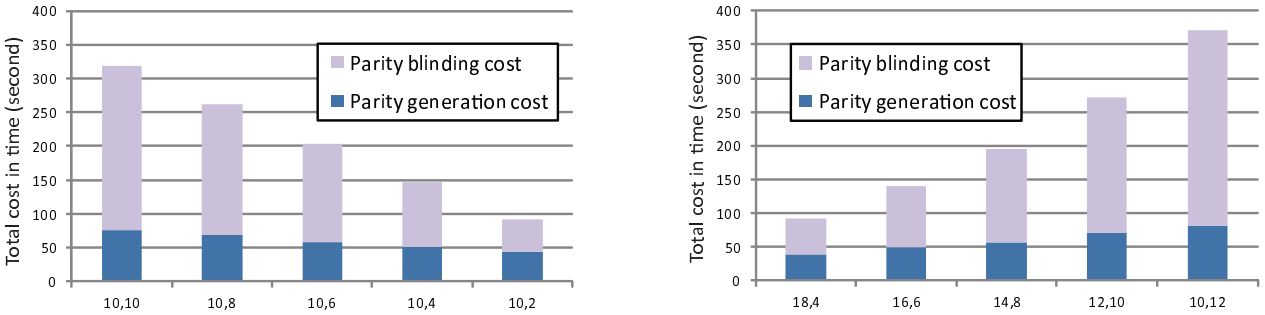
Next, we consider the false negative probability that $R_i^{(j)} = v_i^{(j)}$ when at least one of $\hat{z}$ blocks is modified. According to proposition 1 of [30], tokens calculated in $GF(2^p)$ for two different data vectors collide with probability $\widehat{P}_f^r = 2^{-p}$. Thus, the identification probability for misbehaving server(s) is $\widehat{P}_d = \widehat{P}_m^r \cdot (1 - \widehat{P}_f^r)$. Along with the analysis in detection probability, if $z = 1\%$ of l and each token covers 460 indices, the identification probability for misbehaving servers is at least 99%. Note that if the number of detected misbehaving servers is less than the parity vectors, we can use erasure-correcting code to recover the corrupted data, achieving storage dependability as shown at Section 3.4 and Algorithm 3.

5.2.3 Security Strength Against Worst Case Scenario

We now explain why it is a must to blind the parity blocks and how our proposed schemes achieve collusion resistance against the worst case scenario in the adversary model.

Recall that in the file distribution preparation, the redundancy parity vectors are calculated via multiplying the file matrix $\mathbf{F}$ by $\mathbf{P}$, where $\mathbf{P}$ is the secret parity generation matrix we later rely on for storage correctness assurance. If we disperse all the generated vectors directly after token pre-computation, i.e., without blinding, malicious servers that collaborate can reconstruct the secret $\mathbf{P}$ matrix easily: they can pick blocks from the same rows among the data and parity vectors to establish a set of $m \cdot k$ linear equations and solve for the $m \cdot k$ entries of the parity generation matrix $\mathbf{P}$. Once they have the knowledge of $\mathbf{P}$, those malicious servers can consequently modify any part of the data blocks and calculate the corresponding parity blocks, and vice versa, making their codeword relationship always consistent. Therefore, our storage correctness challenge scheme would be undermined—even if those modified blocks are covered by the specified rows, the storage correctness check equation would always hold.

3. Note that n_c and k only affect the false negative probability P_f^r. However in our scheme, since $p = 16$ almost dominates the negligibility of P_f^r, the value of n_c and k have little effect in the plot of P_d.



(a) m is fixed, and k is decreasing. (b) $m + k$ is fixed.

Fig. 4: Performance comparison between two different parameter settings for 1 GB file distribution preparation. The (m, k) denotes the chosen parameters for the underlying Reed-Solomon coding. For example, (10,2) means we divide file into 10 data vectors and then generate 2 redundant parity vectors.

To prevent colluding servers from recovering **P** and making up consistently-related data and parity blocks, we utilize the technique of adding random perturbations to the encoded file matrix and hence hide the secret matrix **P**. We make use of a keyed pseudorandom function $f_{k_j}(\cdot)$ with key k_j and seed s_{ij}^{ver}, both of which has been introduced previously. In order to maintain the systematic layout of data file, we only blind the parity blocks with random perturbations (We can also only blind data blocks and achieve privacy-preserving third party auditing, as shown in Section 3.5). Our purpose is to add "noise" to the set of linear equations and make it computationally infeasible to solve for the correct secret matrix **P**. By blinding each parity block with random perturbation, the malicious servers no longer have all the necessary information to build up the correct linear equation groups and therefore cannot derive the secret matrix **P**.

5.3 Performance Evaluation

We now assess the performance of the proposed storage auditing scheme. We focus on the cost of file distribution preparation as well as the token generation. Our experiment is conducted on a system with an Intel Core 2 processor running at 1.86 GHz, 2048 MB of RAM, and a 7200 RPM Western Digital 250 GB Serial ATA drive. Algorithms are implemented using open-source erasure coding library Jerasure [31] written in C. All results represent the mean of 20 trials.

5.3.1 File Distribution Preparation

As discussed, file distribution preparation includes the generation of parity vectors (the encoding part) as well as the corresponding parity blinding part. We consider two sets of different parameters for the (m, k) Reed-Solomon encoding, both of which work over $GF(2^{16})$. Figure 4 shows the total cost for preparing a 1 GB file before outsourcing. In the figure on the left, we set the number of data vectors m constant at 10, while decreasing the number of parity vectors k from 10 to 2. In the one on the right, we keep the total number of data and parity vectors $m + k$ fixed at 22, and change the number of data vectors m from 18 to 10. From the figure, we can see the number k is the dominant factor for the cost of both parity generation and parity blinding. This can be explained as follows: on the one hand, k determines how many parity vectors are required before data outsourcing, and the parity generation cost increases almost linearly with the growth of k; on the other hand, the growth of k means the larger number of parity blocks required to be blinded, which directly leads to more calls to our non-optimized PRF generation in C. By using more practical PRF constructions, such as HMAC [32], the parity blinding cost is expected to be further improved.

Compared to the existing work [20], it can be shown from Figure 4 that the file distribution preparation of our scheme is more efficient. This is because in [20] an additional layer of error-correcting code has to be conducted on all the data and parity vectors right after the file distribution encoding. For the same reason, the two-layer coding structure makes the solution in [20] more suitable for static data only, as any change to the contents of file **F** must propagate through the two-layer error-correcting code, which entails both high communication and computation complexity. But in our scheme, the file update only affects the specific "rows" of the encoded file matrix, striking a good balance between both error resilience and data dynamics.

5.3.2 Challenge Token Computation

Although in our scheme the number of verification token t is a fixed priori determined before file distribution, we can overcome this issue by choosing sufficient large t in practice. For example, when t is selected to be 7300 and 14600, the data file can be verified every day for the next 20 years and 40 years, respectively, which should be of enough use in practice. Note that instead of directly computing each token, our implementation uses the Horner algorithm suggested in [21] to calculate token $v_i^{(j)}$ from the back, and achieves a slightly faster

Verify daily for next 20 years	$(m,k)=(10,4)$	$(m,k)=(10,6)$	$(m,k)=(10,8)$	$(m,k)=(14,8)$
Storage overhead (KB)	199.61	228.13	256.64	313.67
Computation overhead (Second)	41.40	47.31	53.22	65.05
Verify daily for next 40 years	$(m,k)=(10,4)$	$(m,k)=(10,6)$	$(m,k)=(10,8)$	$(m,k)=(14,8)$
Storage overhead (KB)	399.22	456.25	513.28	627.34
Computation overhead (Second)	82.79	94.62	106.45	130.10

TABLE 1: The storage and computation cost of token pre-computation for 1GB data file under different system settings. The (m,k) denotes the parameters for the underlying Reed-Solomon coding, as illustrated in Fig. 4.

performance. Specifically,

$$v_i^{(j)} = \sum_{q=1}^{r} \alpha_i^{r+1-q} * G^{(j)}[I_q] = (((G^{(j)}[I_1] * \alpha_i + G^{(j)}[I_2]) * \alpha_i + G^{(j)}[I_3] \ldots) * \alpha_i + G^{(j)}[I_r]) * \alpha_i,$$

which only requires r multiplication and $(r-1)$ XOR operations. With Jerasure library [31], the multiplication over $GF(2^{16})$ in our experiment is based on discrete logarithms.

Following the security analysis, we select a practical parameter $r = 460$ for our token pre-computation (see Section 5.2.1), i.e., each token covers 460 different indices. Other parameters are along with the file distribution preparation. Our implementation shows that the average token pre-computation cost is about 0.4 ms. This is significantly faster than the hash function based token pre-computation scheme proposed in [13]. To verify encoded data distributed over a typical number of 14 servers, the total cost for token pre-computation is no more than 1 and 1.5 minutes, for the next 20 years and 40 years, respectively. Note that each token is only an element of field $GF(2^{16})$, the extra storage for those pre-computed tokens is less than 1MB, and thus can be neglected. Table 1 gives a summary of storage and computation cost of token pre-computation for 1GB data file under different system settings.

6 Related Work

Juels *et al.* [9] described a formal "proof of retrievability" (POR) model for ensuring the remote data integrity. Their scheme combines spot-checking and error-correcting code to ensure both possession and retrievability of files on archive service systems. Shacham et al. [16] built on this model and constructed a random linear function based homomorphic authenticator which enables unlimited number of challenges and requires less communication overhead due to its usage of relatively small size of BLS signature. Bowers *et al.* [17] proposed an improved framework for POR protocols that generalizes both Juels and Shacham's work. Later in their subsequent work, Bowers et al. [20] extended POR model to distributed systems. However, all these schemes are focusing on static data. The effectiveness of their schemes rests primarily on the preprocessing steps that the user conducts before outsourcing the data file **F**. Any change to the contents of **F**, even few bits, must propagate through the error-correcting code and the corresponding random shuffling process, thus introducing significant computation and communication complexity. Recently, Dodis *et al.* [19] gave theoretical studies on generalized framework for different variants of existing POR work.

Ateniese *et al.* [10] defined the "provable data possession" (PDP) model for ensuring possession of file on untrusted storages. Their scheme utilized public key based homomorphic tags for auditing the data file. However, the pre-computation of the tags imposes heavy computation overhead that can be expensive for an entire file. In their subsequent work, Ateniese *et al.* [13] described a PDP scheme that uses only symmetric key based cryptography. This method has lower-overhead than their previous scheme and allows for block updates, deletions and appends to the stored file, which has also been supported in our work. However, their scheme focuses on single server scenario and does not provide data availability guarantee against server failures, leaving both the distributed scenario and data error recovery issue unexplored. The explicit support of data dynamics has further been studied in the two recent work [14] and [15]. Wang *et al.* [14] proposed to combine BLS based homomorphic authenticator with Merkle Hash Tree to support fully data dynamics, while Erway *et al.* [15] developed a skip list based scheme to enable provable data possession with fully dynamics support. The incremental cryptography work done by Bellare *et al.* [33] also provides a set of cryptographic building blocks such as hash, MAC, and signature functions that may be employed for storage integrity verification while supporting dynamic operations on data. However, this branch of work falls into the traditional data integrity protection mechanism, where local copy of data has to be maintained for the verification. It is not yet clear how the work can be adapted to cloud storage scenario where users no longer have the data at local sites but still need to ensure the storage correctness efficiently in the cloud.

In other related work, Curtmola *et al.* [18] aimed to ensure data possession of multiple replicas across the distributed storage system. They extended the PDP scheme to cover multiple replicas without encoding each replica separately, providing guarantee that multiple copies of data are actually maintained. Lillibridge *et al.* [22] presented a P2P backup scheme in which blocks of a data file are dispersed across $m+k$ peers using an (m,k)-erasure code. Peers can request random blocks from their backup peers and verify the integrity using separate

keyed cryptographic hashes attached on each block. Their scheme can detect data loss from free-riding peers, but does not ensure all data is unchanged. Filho *et al.* [34] proposed to verify data integrity using RSA-based hash to demonstrate uncheatable data possession in peer-to-peer file sharing networks. However, their proposal requires exponentiation over the entire data file, which is clearly impractical for the server whenever the file is large. Shah *et al.* [11], [12] proposed allowing a TPA to keep online storage honest by first encrypting the data then sending a number of pre-computed symmetric-keyed hashes over the encrypted data to the auditor. However, their scheme only works for encrypted files, and auditors must maintain long-term state. Schwarz *et al.* [21] proposed to ensure static file integrity across multiple distributed servers, using erasure-coding and block-level file integrity checks. We adopted some ideas of their distributed storage verification protocol. However, our scheme further support data dynamics and explicitly study the problem of misbehaving server identification, while theirs did not. Very recently, Wang *et al.* [28] gave a study on many existing solutions on remote data integrity checking, and discussed their pros and cons under different design scenarios of secure cloud storage services.

Portions of the work presented in this paper have previously appeared as an extended abstract in [1]. We have revised the article a lot and add more technical details as compared to [1]. The primary improvements are as follows: Firstly, we provide the protocol extension for privacy-preserving third-party auditing, and discuss the application scenarios for cloud storage service. Secondly, we add correctness analysis of proposed storage verification design. Thirdly, we completely redo all the experiments in our performance evaluation part, which achieves significantly improved result as compared to [1]. We also add detailed discussion on the strength of our bounded usage for protocol verifications and its comparison with state-of-the-art.

7 Conclusion

In this paper, we investigate the problem of data security in cloud data storage, which is essentially a distributed storage system. To achieve the assurances of cloud data integrity and availability and enforce the quality of dependable cloud storage service for users, we propose an effective and flexible distributed scheme with explicit dynamic data support, including block update, delete, and append. We rely on erasure-correcting code in the file distribution preparation to provide redundancy parity vectors and guarantee the data dependability. By utilizing the homomorphic token with distributed verification of erasure-coded data, our scheme achieves the integration of storage correctness insurance and data error localization, i.e., whenever data corruption has been detected during the storage correctness verification across the distributed servers, we can almost guarantee the simultaneous identification of the misbehaving server(s). Considering the time, computation resources, and even the related online burden of users, we also provide the extension of the proposed main scheme to support third-party auditing, where users can safely delegate the integrity checking tasks to third-party auditors and be worry-free to use the cloud storage services. Through detailed security and extensive experiment results, we show that our scheme is highly efficient and resilient to Byzantine failure, malicious data modification attack, and even server colluding attacks.

Acknowledgments

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

References

[1] C. Wang, Q. Wang, K. Ren, and W. Lou, "Ensuring data storage security in cloud computing," in *Proc. of IWQoS'09*, July 2009, pp. 1–9.

[2] Amazon.com, "Amazon web services (aws)," Online at http://aws.amazon.com/, 2009.

[3] Sun Microsystems, Inc., "Building customer trust in cloud computing with transparent security," Online at https://www.sun.com/offers/details/sun_transparency.xml, November 2009.

[4] M. Arrington, "Gmail disaster: Reports of mass email deletions," Online at http://www.techcrunch.com/2006/12/28/gmail-disasterreports-of-mass-email-deletions/, December 2006.

[5] J. Kincaid, "MediaMax/TheLinkup Closes Its Doors," Online at http://www.techcrunch.com/2008/07/10/mediamaxthelinkup-closes-its-doors/, July 2008.

[6] Amazon.com, "Amazon s3 availability event: July 20, 2008," Online at http://status.aws.amazon.com/s3-20080720.html, July 2008.

[7] S. Wilson, "Appengine outage," Online at http://www.cio-weblog.com/50226711/appengine_outage.php, June 2008.

[8] B. Krebs, "Payment Processor Breach May Be Largest Ever," Online at http://voices.washingtonpost.com/securityfix/2009/01/payment_processor_breach_may_b.html, Jan. 2009.

[9] A. Juels and J. Burton S. Kaliski, "Pors: Proofs of retrievability for large files," in *Proc. of CCS'07*, Alexandria, VA, October 2007, pp. 584–597.

[10] 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*, Alexandria, VA, October 2007, pp. 598–609.

[11] M. A. Shah, M. Baker, J. C. Mogul, and R. Swaminathan, "Auditing to keep online storage services honest," in *Proc. of HotOS'07*. Berkeley, CA, USA: USENIX Association, 2007, pp. 1–6.

[12] M. A. Shah, R. Swaminathan, and M. Baker, "Privacy-preserving audit and extraction of digital contents," Cryptology ePrint Archive, Report 2008/186, 2008, http://eprint.iacr.org/.

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

[14] 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, volume 5789 of LNCS*. Springer-Verlag, Sep. 2009, pp. 355–370.

[15] C. Erway, A. Kupcu, C. Papamanthou, and R. Tamassia, "Dynamic provable data possession," in *Proc. of CCS'09*, 2009, pp. 213–222.

[16] H. Shacham and B. Waters, "Compact proofs of retrievability," in *Proc. of Asiacrypt'08, volume 5350 of LNCS*, 2008, pp. 90–107.

[17] K. D. Bowers, A. Juels, and A. Oprea, "Proofs of retrievability: Theory and implementation," in *Proc. of ACM workshop on Cloud Computing security (CCSW'09)*, 2009, pp. 43–54.

[18] R. Curtmola, O. Khan, R. Burns, and G. Ateniese, "Mr-pdp: Multiple-replica provable data possession," in *Proc. of ICDCS'08*. IEEE Computer Society, 2008, pp. 411–420.

[19] Y. Dodis, S. Vadhan, and D. Wichs, "Proofs of retrievability via hardness amplification," in *Proc. of the 6th Theory of Cryptography Conference (TCC'09)*, San Francisco, CA, USA, March 2009.

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

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

[22] M. Lillibridge, S. Elnikety, A. Birrell, M. Burrows, and M. Isard, "A cooperative internet backup scheme," in *Proc. of the 2003 USENIX Annual Technical Conference (General Track)*, 2003, pp. 29–41.

[23] M. Castro and B. Liskov, "Practical byzantine fault tolerance and proactive recovery," *ACM Transaction on Computer Systems*, vol. 20, no. 4, pp. 398–461, 2002.

[24] L. Carter and M. Wegman, "Universal hash functions," *Journal of Computer and System Sciences*, vol. 18, no. 2, pp. 143–154, 1979.

[25] J. Hendricks, G. Ganger, and M. Reiter, "Verifying distributed erasure-coded data," in *Proc. of 26th ACM Symposium on Principles of Distributed Computing*, 2007, pp. 139–146.

[26] J. S. Plank and Y. Ding, "Note: Correction to the 1997 tutorial on reed-solomon coding," University of Tennessee, Tech. Rep. CS-03-504, April 2003.

[27] C. Wang, Q. Wang, K. Ren, and W. Lou, "Privacy-preserving public auditing for storage security in cloud computing," in *Proc. of IEEE INFOCOM'10*, San Diego, CA, USA, March 2010.

[28] C. Wang, K. Ren, W. Lou, and J. Li, "Towards publicly auditable secure cloud data storage services," *IEEE Network Magazine*, vol. 24, no. 4, pp. 19–24, 2010.

[29] R. C. Merkle, "Protocols for public key cryptosystems," in *Proc. of IEEE Symposium on Security and Privacy*, Los Alamitos, CA, USA, 1980.

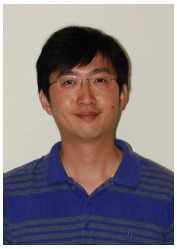
[30] Q. Wang, K. Ren, W. Lou, and Y. Zhang, "Dependable and secure sensor data storage with dynamic integrity assurance," in *Proc. of IEEE INFOCOM'09*, Rio de Janeiro, Brazil, Appril 2009.

[31] J. S. Plank, S. Simmerman, and C. D. Schuman, "Jerasure: A library in C/C++ facilitating erasure coding for storage applications - Version 1.2," University of Tennessee, Tech. Rep. CS-08-627, August 2008.

[32] M. Bellare, R. Canetti, and H. Krawczyk, "Keying hash functions for message authentication," in *Proc. of Crypto'96, volume 1109 of LNCS*. Springer-Verlag, 1996, pp. 1–15.

[33] M. Bellare, O. Goldreich, and S. Goldwasser, "Incremental cryptography: The case of hashing and signing," in *Proc. of CRYPTO'94, volume 839 of LNCS*. Springer-Verlag, 1994, pp. 216–233.

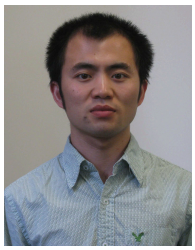
[34] D. L. G. Filho and P. S. L. M. Barreto, "Demonstrating data possession and uncheatable data transfer," Cryptology ePrint Archive, Report 2006/150, 2006, http://eprint.iacr.org/.



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, and secure computation outsourcing.

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.

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 smart grid security and energy efficiency. His research is sponsored by US National Science Foundation and Department of Energy. He is a member of IEEE and ACM.



Ning Cao received his B.E. and M.E. degrees from Xi'an Jiaotong University, China, in 2002 and 2008, respectively. He is currently a PhD student in the Electrical and Computer Engineering Department at Worcester Polytechnic Institute. His research interests are in the areas of storage codes, security and privacy in Cloud Computing, and secure mobile cloud.

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.