Implementation of a Distributed Secure Cloud Storage for Dynamic Data

Nishant Nikam

Implementation of a Distributed Secure Cloud Storage for Dynamic Data

DISSERTATION SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF

Master of Technology
in
Computer Science

by

Nishant Nikam

[Roll No: CS-1518]

under the guidance of

Dr. Sushmita Ruj

Assistant Professor
Cryptology and Security Research Unit

Indian Statistical Institute
Kolkata-700108, India

July 2017

To my family and friends

CERTIFICATE

This is to certify that the dissertation entitled **"Implementation of a Distributed Secure Cloud Storage for Dynamic Data"** submitted by **Nishant Nikam** to Indian Statistical Institute, Kolkata, in partial fulfillment for the award of the degree of **Master of Technology in Computer Science** is a bonafide record of work carried out by him under my supervision and guidance. The dissertation has fulfilled all the requirements as per the regulations of this institute and, in my opinion, has reached the standard needed for submission.

Sushmita Ruj
Assistant Professor,
Cryptology and Security Research Unit,
Indian Statistical Institute,
Kolkata-700108, INDIA.

Acknowledgments

I would like to show my highest gratitude to my advisor, *Sushmita Ruj*, Cryptology and Security Research Unit, Indian Statistical Institute, Kolkata, for her guidance and continuous support and encouragement. She has literally taught me how to do good research, and motivated me with great insights and innovative ideas.

I would also like to thank *Binanda Sengupta*, Senior Research Fellow, Indian Statistical Institute, Kolkata, for his valuable suggestions and discussions.

I would like to thank *Srinivasan Narayanmurthy*, and *Siddhartha Nandi*, *NetApp Inc.* for providing detailed comments and pointers regarding implementation.

My deepest thanks to all the teachers of Indian Statistical Institute, for their valuable suggestions and discussions which added an important dimension to my research work.

Finally, I am very much thankful to my parents and family for their everlasting supports.

Last but not the least, I would like to thank all of my friends for their help and support. I thank all those, whom I have missed out from the above list.

Nishant Nikam
Indian Statistical Institute
Kolkata - 700108 , India.

Abstract

Cloud computing enables clients to outsource large volume of their data to cloud servers. Distributed secure cloud storage schemes ensure that multiple servers store these data in a reliable and untampered fashion. The core problem is to build systems that are efficient and provably secure. In a proof-of-retrievability system, a client is assured by a server that it is storing all of client's data, by running periodic audits. It should be possible for client to extract it's data from the server that passes verification checks. We implement multi-server auditing scheme for static data by encoding data blocks using error-correcting (erasure) codes and then attaching authentication information tags to parity blocks of the codewords. We extend our secure cloud storage scheme for append-only data that handles the challenges efficiently. Compared to existing implementations, our scheme is such that the client need not download any data to update the parity blocks or corresponding tags residing on the servers. This results in low communication costs. It enables the servers to perform the updates themselves and helps the client to detect malicious behavior of the server.

Keywords: *Distributed storage systems, erasure codes, message authentication codes, proof-of-retrievability*

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Introduction

Cloud storage outsourcing has become one of the most popular applications of cloud computing, offering various advantages like flexible accessibility. e.g., Amazon S3, Microsoft Azure, Google Drive etc. Client who doesn't possess capability to store data (large) physically needs a verifiable, secure cloud storage system. When the server is untrusted, an important challenge is to offer provable outsourced storage guarantees. Particularly, a client desires to obtain the following guarantees:

- **Authenticity.** The client desires to verify that data it fetched after storing on server is correct, where correctness is identical to authenticity and freshness;

- **Retrievability.** The client needs a guarantee that the server is truly storing all of the its's data, and there is no data loss.

Cloud service providers offer storage outsourcing facility to their cloud users (clients) who can upload large amount of data to the cloud servers and can access their data later whenever needed. However, as the client stores only some metadata for the data file it uploads, the file may get corrupted at the cloud servers, thus making it impossible to retrieve at some point of time. Secure cloud storage schemes address this problem where the client (or a third party auditor) checks the availability of the uploaded file.

Ateniese et al. [1] introduced the concept of *provable data possession* (PDP) and Juels and Kaliski [15] introduced the concept of *proofs of retrievability* (PoR). Secure cloud storage schemes typically use the concept of PDP or PoR. These schemes employ an auditing mechanism where the client executes a challenge-response protocol to check the integrity of her data. In general, PDP schemes are more efficient than PoR schemes. However, PoR schemes preserve the integrity of *all* of the client's data. Following their works, researchers have come up with many schemes achieving PDP or PoR guarantees [2, 11, 26, 23, 10, 6, 25].

The secure cloud storage schemes mentioned above consider the single-server model where the client uploads it's data to a single cloud server. However, storing the whole data in one server is more prone to adversarial corruptions and hardware outages. A fault in this single point can make the data unrecoverable. To increase data reliability and consistency, the client's data are dispersed among multiple servers in practice. In this model, it is harder for an adversary to corrupt all the servers at a time to make the data unavailable. Moreover, the system is more stable in case of hardware crashes for some of the servers [12]. Curtmola et al. [8] introduce MR-PDP (multiple-replica PDP) where the client uses data replication to make multiple copies of her data and distributes them to multiple servers. Zhu et al. [28] propose CPDP (cooperative PDP) scheme based on homomorphic verifiable response and hash index hierarchy where the client's data are distributed among various cloud service providers (CSPs). Data privacy in this scheme is achieved using the techniques of multiprover zero-knowledge proof system.

Dimakis et al. [9] introduce network coding in the context of distributed storage systems where linear combinations of data segments are distributed to multiple servers. This technique is more efficient than using the conventional error-correcting codes for distributing the segments in terms of bandwidth required to repair a failed server. For such a distributed storage system, there are schemes for remote integrity checking [7, 17, 19] that are designed to achieve fast repair of a failed server. One drawback of network coding is that the code is not systematic (that is, a codeword does not include the input message as it is); thus read operations are not efficient in these schemes. Therefore, these schemes are mostly suitable for archival data (with fast repair) where the client accesses her data less frequently.

Error-correcting codes (mostly for erasures) have been used extensively to design distributed storage systems, and they provide optimal storage overhead to achieve the same reliability compared to other techniques. Moreover, read operations are quite efficient for systematic variants of these codes. Several secure cloud storage schemes employ error-correcting codes to enhance tolerance against faults in storage systems [18, 27, 14].

Schwarz and Miller [21] exploit *algebraic signatures* along with error-correcting codes to construct a distributed secure storage where data blocks in random locations are challenged to check the integrity of the client's data. Algebraic signatures can be aggregated into a single signature that reduces the communication overhead significantly.

1.2 Our Contribution

Our contributions are summarized as follows.

- We implement a distributed secure cloud storage scheme for static data that borrows the basic storage structure from HAIL [5]. The scheme offers PoR guar-

antees. Unlike HAIL, an adversary in the scheme cannot modify a particular dispersal codeword without being detected by the client.

- We implement the extended scheme for append-only (dynamic) data [22].
- In the scheme for append-only (dynamic) data, individual servers can update their parity blocks (for an append) without any intervention of the client.
- For an append, the client in the scheme need not download any parity (or data) block to recompute the authentication tags corresponding to the (updated) parity blocks. She can only send the relevant changes in these tags to the corresponding servers.
- We use systematic Cauchy Reed-Solomon codes in the scheme for static data and implement a technique to extend such codes to accommodate new symbols appended to the existing message symbols. The corresponding updates on the parity symbols do not touch existing message symbols.

1.3 Thesis Outline

The rest of the thesis is organized as follows. In Chapter 2 and 3, we briefly discuss about the preliminaries and background related to our work. Chapter 4, describes the detailed construction of the scheme we used for implementation. In Chapter 5, we describe about implementation details. In Chapter 6, we provide the security analysis and performance analysis our scheme. In the concluding Chapter 7, we summarize the work done and future directions related to our work.

Chapter 2

Preliminaries

2.1 Notation

We take λ to be the security parameter. An algorithm $\mathcal{A}(1^\lambda)$ is a probabilistic polynomial-time algorithm when its running time is polynomial in λ and its output y is a random variable which depends on the internal coin tosses of $\mathcal{A}$. An element a chosen uniformly at random from a set S is denoted as $a \xleftarrow{R} S$. A function $f : \mathbb{N} \to \mathbb{R}$ is called negligible in λ if for all positive integers c and for all sufficiently large λ, we have $f(\lambda) < \frac{1}{\lambda^c}$.

2.2 Error Correcting Codes

An $(n, k, d)_{\sum}$ error-correcting code consists of an encoding algorithm $Enc : \sum^k \to \sum^n$ (encodes a message consisting of k symbols into a longer codeword consisting of n symbols) and a decoding algorithm $Dec : \sum^n \to \sum^k$ (decodes a codeword to a message), where $\sum$ is a finite alphabet and d is the minimum distance (Hamming distance between any two codewords is at least d) of the code. The quantity $\frac{k}{n}$ is called the rate of the code. An $(n, k, d)_{\sum}$ error-correcting code can tolerate up to $\lfloor \frac{d-1}{2} \rfloor$ errors and d - 1 erasures. If $d = n - k + 1$, we call the code a maximum distance separable (MDS) code. We often specify the parameters of an MDS code by denoting it as an (n, k) MDS code, where $\sum$ is implicit and the minimum distance $d = n - k + 1$. Reed-Solomon codes [] and their extensions are examples of non-trivial linear MDS codes.

2.3 Cauchy Reed-Solomon Code

For an (n, k) Cauchy Reed-Solomon code [3], the distribution matrix is an $n \times k$ matrix M CRS over Z_p , where the submatrix consisting of the first k rows is a $k \times k$ identity matrix and the submatrix consisting of the last $s = n - k$ rows is an $s \times k$ Cauchy

matrix. The codeword is obtained by multiplying M_{CRS} with the message vector $\vec{m} = [m_1, m_2, ..., m_k]^T$, where $m_i \in Z_p$ for all $i \in [1, k]$. It is a maximum distance separable (MDS) code with minimum distance $d = n - k + 1$. It can tolerate upto $\lfloor \frac{n-k}{2} \rfloor$ errors and $n - k$ erasures. An $s \times k$ Cauchy matrix ($k + s \leqslant p$) is constructed in the following way. Let $X = \{x_1, x_2, ..., x_s\}$ and $Y = \{y_1, y_2, ..., y_k\}$ be two ordered sets such that the following conditions are satisfied:

1. $x_i, y_j \in Z_p$ for all $i \in [1, s]$ and $j \in [1, k]$,
2. $X \cap Y = \emptyset$ which implies that $x_i - y_j \neq 0$ for all $i \in [1, s]$ and $j \in [1, k]$,
3. $\forall i \in [1, s] \forall l \in [1, s] \backslash \{i\} \qquad x_i \neq x_l$,
4. $\forall j \in [1, k] \forall l \in [1, k] \backslash \{j\} \qquad y_j \neq y_l$.

The $s \times k$ Cauchy matrix defined by X and Y consists of the entries $a_{ij} = \frac{1}{x_i - y_j}$, where $i \in [1, s]$ and $j \in [1, k]$. Any $k \times k$ submatrix of M_{CRS} is invertible.

2.4 Galois Field

Galois Field (named after Évariste Galois), also known as *Finite Field* is a field that contains a finite number of elements.

The elements of Galois Field $GF(p^n)$ is defined as

$$\begin{aligned} GF(p^n) = &(0, 1, 2, ..., p-1) \cup \\ &(p, p+1, p+2, ..., p+p-1) \cup \\ &(p^2, p^2+1, p^2+2, ..., p^2+p-1) \cup ... \cup \\ &(p^{n-1}, p^{n-1}+1, p^{n-1}+2, ..., p^{n-1}+p-1) \end{aligned} \tag{2.1}$$

where $p \in P$ and $n \in Z^+$. The order of the field is given by p^n while p is called the *characteristic* of the field. On the other hand, gf, stands for Galois Field. The degree of polynomial of each element is at most $n - 1$.

Chapter 3

Related Work

3.1 Proofs of Retrievability

A client uploads a file to the cloud server. However, the client needs a guarantee that all it's data are stored in the server untampered. Proofs-of-retrievability (PoR) schemes make the client be assured that it's data are stored intact in the server. Juels and Kaliski introduce proofs of retrievability for static data [15]. Static data mostly include archival data which the client does not modify after it uploads the file to the server. However, some of the PoR schemes deal with dynamic data where the client modifies it's data. We provide a brief idea about the building blocks of PoR schemes. In the setup phase, the client preprocesses her file F_0. The preprocessing step involves encoding the file F_0 with an erasure code to form another file F. Then, an authenticator is attached to each of the blocks of F (for checking the integrity of the blocks later). Finally, the client uploads F along with the authenticators to the server. We consider the file F as a collection of n blocks or segments where each block is an element of Z_p. The client can read data from the file it has outsourced. it performs audits to check the integrity of it's data. An audit comprises of two algorithms for proof-generation and proof-verification. During an audit, the client generates a random challenge and sends it to the server which acts as a prover. Upon receiving the challenge, the server responds to the client with a proof. The client then verifies the integrity of the data by checking the validity of the proof. If the proof is valid, the verification algorithm outputs 1; otherwise, it outputs 0. For dynamic POR schemes, the client can issue write operations along with read operations. POR schemes satisfy two properties: *correctness* and *soundness*.

- **Correctness.** The correctness property demands that the proof generated by an honest server always makes the verification algorithm output 1.

- **Soundness.** The soundness property of POR schemes is formalized by the existence of an extractor algorithm that extracts F after interacting with a malicious

server which passes an audit (that is, the verification algorithm outputs 1) with any probability non-negligible in the security parameter λ.

There are two types of PoR schemes: *privately* verifiable and *publicly* verifiable schemes. In private verification schemes, only the client can perform audits as the verification of a proof requires some secret information. On the other hand, in publicly verifiable schemes, anyone can verify the proof supplied by the server. In privacy preserving auditing, the verifier (any verifier other than the client) cannot gain any knowledge about the data outsourced to the server.

3.2 PoR schemes by Shacham and Waters

Shacham and Waters propose two short and efficient homomorphic authenticators in their PoR schemes for static data [24]. The first one, based on pseudorandom functions, provides a PoR scheme which is privately verifiable (that is, only the client can verify a proof) and secure in the standard $model^1$; the second one, based on BLS signatures, gives a PoR scheme which is publicly verifiable (that is, anyone can verify a proof) and secure in the random oracle $model^2$. As mentioned by Shacham and Waters, Reed-Solomon codes are necessary against adversarial erasures where the server can delete blocks selectively. One drawback of these codes is the complexity of encoding and decoding is $O(n^2)$, where n is the number of blocks of the file uploaded to the server. We can employ codes with linear decoding time instead of Reed-Solomon codes. However, these codes are secure against random erasures only. Shacham and Waters discuss a solution to this problem strictly for the privately verifiable scheme.

3.3 Proofs of Retrievability for Dynamic Data

In the previous section, we have described some PoR schemes for static data which the clients do not modify once they are uploaded in the cloud server. A natural question comes if any PoR schemes are available for dynamic data where the clients modify their outsourced data "efficiently". In this section, we discuss about the difficulties of modification of the uploaded data.

To maintain the retrievability of the whole file, erasure coding has been employed on the file. The blocks of the file are encoded in such a way that the file can be retrieved from a fraction of blocks of the encoded file. The content of each block is now distributed in other $O(n)$ blocks. Therefore, to actually delete a block the server has to delete all the related blocks. This restricts the server from deleting or modifying a block maliciously and still passing the verification with non-negligible probability in λ. However, this advantage comes with some drawbacks. If the client wants to update a single block, she has to update all the related blocks as well. This makes the update process inefficient as n can be very large.

Cash et al. [6] discuss about two failed attempts to provide a solution of the problem mentioned above. In the first case, a possible solution might be to encode the file locally. Now, each codeword consists of a small number of blocks. Therefore, an update of a single block requires an update of a few blocks within that particular codeword. However, a malicious server can gain the knowledge of this small set of blocks (within a codeword) whenever the client updates a single block. Thus, the server can delete this small set of blocks without being noticed during an audit. In the second attempt, after encoding the file locally, all of the n blocks are permuted in a pseudorandom fashion. Apparently, the server cannot get any information about the blocks in a codeword. However, during an update the server can identify the related blocks in a codeword. Therefore, the server can again delete these blocks and pass the verification during an audit. Due to the issues discussed above, only a few PoR schemes for dynamic data are available in the literature.

3.4 HAIL

Schwarz and Miller [21] exploit *algebraic signatures* along with error-correcting codes. Following a similar idea, Bowers et al. [5] propose a distributed secure cloud storage scheme called HAIL (high-availability and integrity layer for cloud storage) that achieves POR guarantees. Encoding of the data blocks of a file is done in two steps: across multiple servers (dispersal code) and within each server (server code). HAIL enjoys several benefits such as: high reliability, low per-server computation and bandwidth (comparable to single-server PoR schemes) and strong adversarial model. Moreover, message authentication codes (MACs) are embedded in the parity blocks (for the dispersal code) without storing them separately; this reduces the storage overhead on the servers. However, HAIL deals with *static* data (that cannot be modified once uploaded to the servers). Extending HAIL for *dynamic* data is left as a future work in [5].

Although generic dynamic data (supporting arbitrary insertions, deletions and modifications) are useful, *append-only* data find numerous applications as well. Various cloud service providers like Amazon Web Services use Hadoop Distributed File System [13] to store huge volume of append-only data. Append-only data are also useful for maintaining log structures (e.g., in certificate transparency schemes [16]). Extending HAIL for append-only data suffers from two issues stated below.

– HAIL uses an adversarial server code [5] that is resistant to a large fraction of adversarial corruptions against a computationally bounded adversary, but is computationally heavy (due to the use of pseudorandom permutations and encryptions). Moreover, for each append, the parity blocks need to be decrypted (using a secret key of the client), recalculated (depending on the new data block appended) and permuted again (using another secret key of the client). This requires the client to download all the parity blocks for a particular server, do computations on them and upload them again.

– In HAIL, MACs (computed using secret keys of the client) are embedded in the parity blocks (for the dispersal code). Therefore, if a classical error-correcting code were used instead of the adversarial code, then also the client would have had to download all the parity blocks of every dispersal codeword for each append.

Chapter 4

Multi-Server Auditing Scheme for Append-only Data

Outline In this chapter, first we briefly describe storage scheme for static data [22] we have used for implementation. Our aim is to implement scheme for append-only (dynamic) data support scheme. The challenges regarding this idea for extension and how these challenges being addressed are described in scheme for append-only data. Also, We briefly describe Extending Cauchy Reed-Solomon codes which ought to remedial for solving some challenges.

4.1 Distributed Secure Cloud Storage for Static Data

Let n (total number of servers) and k (number of primary servers) be the system parameters that are passed as inputs to the procedures of our scheme for static data described below. The audit phase involves the procedures Challenge, Prove and Verify. We note that the client constructs two distribution matrices M_{CRS} and M'_{CRS} to encode blocks of the data file F *row-wise* and *column-wise*, respectively. We denote the state of F after the corruption phase of the t-th epoch by F_t.

• **Setup**(1^λ): The client chooses a random prime p of length $O(\lambda)$ bits. Let $f : \mathcal{K}_{prf} \times \{0,1\}^* \to \mathbb{Z}_p$ be a pseudorandom function (PRF), where $\mathcal{K}_{prf}$ is the key space of the PRF. The client selects an element $\alpha \xleftarrow{R} \mathbb{Z}_p$ and a PRF key $k_{prf} \xleftarrow{R} \mathcal{K}_{prf}$. Her secret key sk is the pair (α, k_{prf}). Let $\mathcal{F}$ be the space of file-identifiers.

• **Outsource**(F, sk): The client chooses a random file-identifier $\texttt{fid} \xleftarrow{R} \mathcal{F}$ for the data file F. She divides the file F as $\{m_{ij}\}_{i\in[1,\tilde{k}],j\in[1,k]}$, where each data block $m_{ij} \in \mathbb{Z}_p$ for all $i \in [1, \tilde{k}]$, $j \in [1, k]$.

For each primary server S_j ($j \in [1,k]$), the client encodes the data blocks $m_{1j}, m_{2j}, \ldots, m_{\tilde{k}j}$ using an $(r,\tilde{k})$ systematic CRS code to form $\tilde{s} = r - \tilde{k}$ parity blocks $m_{(\tilde{k}+1)j}$, $m_{(\tilde{k}+2)j}, \ldots, m_{rj}$ (with the help of an $r \times \tilde{k}$ matrix M'_{CRS} where $r \leqslant p$). After processing the blocks of the primary servers, the client computes the parity blocks for the secondary servers as follows. For each row $i \in [1,r]$, the client uses an (n,k) systematic CRS code to encode the blocks $m_{i1}, m_{i2}, \ldots, m_{ik}$ into $s = n-k$ parity blocks $m_{i(k+1)}, m_{i(k+2)}, \ldots, m_{in}$ using an $n \times k$ matrix M_{CRS}.

The client computes authentication tags (in a similar way as described by Shacham and Waters [23]) for the parity blocks as follows. For each such block m_{ij} ($i \in [1,r], j \in [k+1,n]$), she generates a tag

$$\sigma_{ij} = f_{k_{prf}}(i,j) + \alpha m_{ij} \bmod p. \tag{4.1}$$

Finally, the client sends $\{m_{ij}\}_{i\in[1,r]}$ to the j-th server S_j for each $j \in [1,n]$. In addition, she uploads authentication tags $\{\sigma_{ij}\}_{i\in[1,r]}$ to the j-th secondary server S_j for each $j \in [k+1,n]$.

• **Challenge**($\mathtt{fid}, l, r, t$): During the audit phase in the t-th epoch, the client selects I, a random l-element subset of $[1,r]$ and generates a challenge set $Q = \{(i,\nu_i)\}_{i\in I}$, where each $\nu_i \xleftarrow{R} \mathbb{Z}_p$. Then, the client sends the challenge set Q to all the servers.

• **Prove**($Q, F_t, \mathtt{fid}, t$): Upon receiving the challenge set $Q = \{(i,\nu_i)\}_{i\in I}$, the j-th cloud server S_j computes and sends $\mu_j = \sum_{i\in I} \nu_i m_{ij} \in \mathbb{Z}_p$ to the client, for all $j \in [1,n]$. Additionally, the j-th secondary server sends $\sigma_j = \sum_{i\in I} \nu_i \sigma_{ij} \in \mathbb{Z}_p$ to the client, for all $j \in [k+1,n]$. The responses from the servers constitute the proof Π.

• **Verify**($Q, \Pi, \mathtt{fid}, sk, t$): Using $Q = \{(i,\nu_i)\}_{i\in I}$ and the proof Π sent by the servers, the client constructs a vector $\overrightarrow{\mu} = [\mu_1, \mu_2, \ldots, \mu_k, \mu_{k+1}, \ldots, \mu_n] \in \mathbb{Z}_p^n$. For each $j \in [k+1,n]$, the client checks whether

$$\sigma_j \stackrel{?}{=} \sum_{i\in I} \nu_i f_{k_{prf}}(i,j) + \alpha\mu_j \bmod p. \tag{4.2}$$

If any of the verifications fails, the client outputs 0. Otherwise, the client checks whether $\overrightarrow{\mu}$ forms a valid codeword (this is done by checking whether multiplying $\overrightarrow{\mu}$ with the Cauchy submatrix of M_{CRS} outputs $\mu_{k+1}, \ldots, \mu_n$ as the parity blocks). If it is a valid codeword, the client outputs 1; she outputs 0, otherwise.

• **Redistribute**($\mathtt{fid}, sk, t, F_t, \epsilon_q$): In the audit phase during the t-th epoch, if the client detects that the fraction of corruption exceeds ϵ_q for some server, the client reads all the shares (possibly corrupted) of the file from all the servers,

tries to recover F (by decoding the file shares of F_t) and distributes the newly computed shares to the servers. These shares of the file distributed across the servers constitute the new state of the file F_{t+1}. If the decoding procedure fails (that is, the client cannot recover F), the file is considered to be unavailable.

Decoding during Redistribution We use two CRS codes — row-wise coding for *errors* and column-wise coding for *erasures*. The decoding procedure works in two steps. In the *first* step, the client executes the decoding procedure row-wise to correct the possible errors in each row. Therefore, the row-wise decoding can correct up to $\lfloor \frac{s}{2} \rfloor$ errors. This requirement restricts the the number of servers the adversary can corrupt in each epoch to be bounded by $b \leqslant \lfloor \frac{n-k}{2} \rfloor$. If the decoding fails for a particular row (for more than $\lfloor \frac{s}{2} \rfloor$ errors), then each data block of that row is marked as an erasure. In the *second* step, the client decodes the column-wise codeword for each primary server (erasures are obtained from the first step). The decoding procedure can recover the original data blocks for a primary server if there are up to $\tilde{s} = r - \tilde{k}$ erasures in the corresponding codeword.

4.2 Extending Cauchy Reed-Solomon Codes for Appending Message Symbols

Let an $n \times k$ matrix M_{CRS} defined by $X = \{x_1, x_2, \ldots, x_s\}$ and $Y = \{y_1, y_2, \ldots, y_k\}$ be the distribution matrix for an (n, k) Cauchy Reed-Solomon (CRS) code over $\mathbb{Z}_p$, where the first k rows form a $k \times k$ identity matrix I_k and the last $s = n - k$ rows form an $s \times k$ Cauchy matrix ($n \leqslant p$).

Let m_{k+1} be another symbol to be appended to $\overrightarrow{m}$. *In this work, we keep the value of s fixed.* So the number of codeword-symbols n increases by 1 for each append, i.e., we set $k_{new} = k + 1$ and $n_{new} = n + 1$ with the restriction $n_{new} \leqslant p$. We choose an element $y_{k_{new}} \in \mathbb{Z}_p$ such that $y_{k_{new}} \notin X$ and $y_{k_{new}} \notin Y$. Then, we add $y_{k_{new}}$ to Y as the last element and construct an $s \times k_{new}$ Cauchy matrix with entries $a_{ij} = \frac{1}{x_i - y_j}$ for $i \in [1, s]$ and $j \in [1, k_{new}]$. To achieve this incrementally, we simply append to the previous $s \times k$ Cauchy matrix a column with entries $a_{ik_{new}} = \frac{1}{x_i - y_{k_{new}}}$ for $i \in [1, s]$. The matrix thus formed is indeed a Cauchy matrix as it satisfies all the conditions mentioned in Section 2.3. Finally, this matrix is appended to the identity matrix I_{k+1} to obtain the updated M_{CRS}.

4.3 Distributed Secure Cloud Storage for Append-only Data

We start with an idea for possible extension of the scheme for static data in order to support append operations and describe some of its challenges. Then, we describe

the distributed secure cloud storage scheme for append-only data that addresses these challenges efficiently.

4.3.1 Idea for Extension

We assume that the client appends a *row* of data blocks at a time, that is, each primary (secondary) server gets a block (a block-tag pair) during an append. The client encodes the new k blocks into n blocks using M_{CRS}, generates tags for the last s blocks and distributes them among the servers. Then, column-wise parity blocks are updated using M'_{CRS} for each server. The client uses two pairs of ordered sets (X_{row}, Y_{row}) and (X_{col}, Y_{col}) each satisfying the conditions mentioned in Section 2.3. We note that X_{row} and Y_{row} are fixed unless we change the number of primary servers or the number of secondary servers. However, for column-wise CRS coding using M'_{CRS}, the set Y_{col} needs to be changed as discussed in Section 4.2.

Challenges We discuss about some challenges regarding the idea as follows.

1. For row-wise (or column-wise) CRS coding, the client needs to store the matrices M_{CRS} (or M'_{CRS}) to encode data blocks. Alternatively, the client can store only the respective Cauchy matrices at her end to reduce the storage overhead. This overhead can be further alleviated if the client stores only the pairs (X_{row}, Y_{row}), (X_{col}, Y_{col}) and computes the required entries of the matrices from them on-the-fly. However, storing these pairs also requires $O(p \log p)$ space that is exponential in λ as $p = 2^{O(\lambda)}$.

2. Let $\sigma_{ij} = f_{k_{prf}}(i, j) + \alpha m_{ij} \bmod p$ and $\sigma_{(i+1)j} = f_{k_{prf}}(i+1, j) + \alpha m_{(i+1)j} \bmod p$ be the tags on the i-th and $(i+1)$-th parity blocks of the j-th secondary server S_j *after the t-th append*, respectively ($i \in [\tilde{k}+1, r-1], j \in [k+1, n]$). Now, *after the $(t+1)$-th append*, the row-wise index of the previous i-th block is $i+1$; and let $\sigma'_{(i+1)j} = f_{k_{prf}}(i+1, j) + \alpha m'_{(i+1)j} \bmod p$ be its updated tag. Therefore, a (possibly) malicious secondary server S_j can compute $f_{k_{prf}}(i+1, j)$ and α using $m_{(i+1)j}$, $m'_{(i+1)j}$, $\sigma_{(i+1)j}$ and $\sigma'_{(i+1)j}$; and thus it can later generate a valid tag on a block indexed by $(i+1, j)$.

3. In column-wise CRS coding, for each server, the client needs to download all the parity blocks (and corresponding tags for each secondary server), update them using the updated M'_{CRS} and upload them to the corresponding server. This requires a huge client-server communication bandwidth.

Addressing the Challenges We describe some remedial measures in order to address the challenges discussed above.

1. We note that the elements in X_{row} and Y_{row} belong to $\mathbb{Z}_p$, where $|X_{row}| = s$ and $|Y_{row}| = k$. We include the first s elements of $\mathbb{Z}_p$ in X_{row} and the last k elements of $\mathbb{Z}_p$ in Y_{row}; that is, $X_{row} = \{0, 1, \ldots, s-1\}$ and $Y_{row} = \{p-k, p-k+1, \ldots, p-1\}$. We can easily verify that X_{row} and Y_{row} thus formed indeed satisfy the conditions mentioned in Section 2.3 as long as $n = k + s \leqslant p$. So the knowledge of $i \in [1, s]$ and $j \in [1, k]$ is sufficient to get the entries of X_{row}, Y_{row} and to compute the entries a_{ij} of M_{CRS} on-the-fly. Therefore, the client need not store these sets. The sets X_{col} and Y_{col} can be formed using the same technique, except that $|Y_{col}|$ varies with the value of $\tilde{k}$.

2. The client uses a counter `ctr`. For the *initial upload*, the client sets the counter `ctr` to 0 and computes tags $\sigma_{ij} = f_{k_{prf}}(i, j, \texttt{ctr}) + \alpha m_{ij} \bmod p$ for all $i \in [1, r]$ and for all $j \in [k+1, n]$. For each append, the client increments `ctr` by 1 and updates the tags σ_{ij} (only for $i \in [\tilde{k}+1, r], j \in [k+1, n]$) accordingly. We note that, for the first $\tilde{k}$ rows (systematic part), the tags corresponding to blocks in the secondary servers never get updated (as the only operation allowed is append in the $\tilde{k}+1$-th row). Therefore, at any point of time, the value of `ctr` is 0 for the first $\tilde{k}$ rows and the value of `ctr` for the rest of the rows is the number of appends that have taken place so far. So due to the properties of a pseudorandom function, the j-th ($j \in [k+1, n]$) server cannot exploit the knowledge of $\sigma_{ij} = f_{k_{prf}}(i, j, \texttt{ctr}) + \alpha m_{ij} \bmod p$ and $\sigma'_{ij} = f_{k_{prf}}(i, j, \texttt{ctr}') + \alpha m'_{ij} \bmod p$ to compute the value(s) of $f_{k_{prf}}(i, j, \texttt{ctr})$, $f_{k_{prf}}(i, j, \texttt{ctr}')$ or α, for any value of $i \in [\tilde{k}+1, r]$ and for any $\texttt{ctr}' > \texttt{ctr}$.

3. For column-wise CRS coding, each server can maintain the updated M'_{CRS} (or compute its entries on-the-fly) and update its parity blocks using M'_{CRS}. *This requires no client-server communication.* A server S_j ($j \in [1, n]$) updates each of its column-wise parity blocks as follows. Let $\tilde{k}_a$ be the number of data blocks (systematic part) present in the column-wise codeword after the t-th append and $m_{\tilde{k}_a j}$ be the newly appended data block for S_j. Let m_{ij} and m'_{ij} be the contents of the i-th parity block ($i \in [\tilde{k}_a + 1, r]$) of S_j after and before the t-th append, respectively. The server S_j multiplies $m_{\tilde{k}_a j}$ with the i-th entry of the newly added $\tilde{k}_a$-th column of the Cauchy submatrix of M'_{CRS}. Then, it adds this product to m'_{ij} in order to get m_{ij}, the updated content of the parity block. Here, we note that *S_j need not touch any existing data blocks m_{ij} ($i \in [1, \tilde{k}_a - 1]$) to update its parity blocks.*

   On the other hand, for each secondary server, the tags on the last $\tilde{s} = n - \tilde{k}$ blocks need to be updated with the latest value of i and `ctr` (both incremented by 1) as discussed above. We describe the procedure for updation of the tag of such a block for S_j ($j \in [k+1, n]$) as follows. Let $(m_{\tilde{k}_a j}, \sigma_{\tilde{k}_a j})$ be the new block-tag pair for S_j when the client encodes the $\tilde{k}_a$-th row using row-wise encoding. Let (m_{ij}, σ_{ij}) and (m'_{ij}, σ'_{ij}) be the block-tag pairs for the i-th block

($i \in [\tilde{k}_a + 1, r]$) of S_j after and before the t-th append, respectively. So, we have $\sigma_{ij} = f_{k_{prf}}(i, j, t) + \alpha m_{ij} \bmod p$ and $\sigma'_{ij} = f_{k_{prf}}(i, j, t-1) + \alpha m'_{ij} \bmod p$. We define $\Delta_\sigma = \sigma_{ij} - \sigma'_{ij} \bmod p$ and $\Delta_m = m_{ij} - m'_{ij} \bmod p$.

As both the row-wise and the column-wise codes used are linear codes, it is easy to see that the content of the i-th block of S_j ($i \in [\tilde{k}_a + 1, r], j \in [k+1, n]$) is the same irrespective of whether we get it by column-wise-then-row-wise encoding or the other way. This crucial observation leads us to the fact that Δ_m can be computed solely from $m_{\tilde{k}_a j}$ and M'_{CRS} as $\Delta_m = m_{\tilde{k}_a j} M'_{CRS}[i, \tilde{k}_a] \bmod p$. Thus, we have

$$\Delta_\sigma = f_{k_{prf}}(i, j, t) - f_{k_{prf}}(i, j, t-1) + \alpha \Delta_m \bmod p. \tag{4.3}$$

The client sends *only* these Δ_σ's for all relevant parity blocks to the secondary servers, and the servers update the respective tags accordingly. Hence, *the client need not download the updated blocks, recompute the tags and then upload them to the secondary servers.*

4.3.2 Scheme for Append-only Data

We assume that the client stores M_{CRS} (and M'_{CRS}) in order to encode/decode the blocks of F row-wise (and to compute changes in tags using Eqn. 4.3). We also assume that each server stores M'_{CRS} to encode its own blocks column-wise. However, we have argued in Section 4.3.1 that *the entries of M_{CRS} and M'_{CRS} can be computed on-the-fly with only $O(1)$ amount of storage.* We also note that M_{CRS} is static whereas M'_{CRS} is extended for each append. The procedures **Setup**, **Challenge**, **Prove** and **Redistribute** in our scheme for append-only data are the same as those described for *static data* in Section 4.1. We describe the rest of the procedures as follows.

• **Outsource**(F, sk): The procedure is the same as the procedure **Outsource** described in Section 4.1 except the following. Instead of using Eqn. 4.1, the client computes an authentication tag for each parity block m_{ij} $i \in [1, r]$,$j \in [k+1, n]$ as

$$\sigma_{ij} = f_{k_{prf}}(i, j, 0) + \alpha m_{ij} \bmod p. \tag{4.4}$$

• **Append**$(\texttt{fid}, sk, \tilde{k}, r, \texttt{ctr}, t)$: The client increments each of $\tilde{k}$, r and `ctr` by 1 and updates M'_{CRS}. She encodes k data blocks (the row to be appended to F) into n blocks using M_{CRS}, generates authentication tags

$$\sigma_{\tilde{k}j} = f_{k_{prf}}(\tilde{k}, j, 0) + \alpha m_{\tilde{k}j} \bmod p \tag{4.5}$$

for all $j \in [k+1, n]$, and sends the blocks and the tags to the corresponding n servers. The servers append the respective blocks (and tags), increment each

of $\tilde{k}$, r and $\texttt{ctr}$ they maintain by 1 and update M'_{CRS} at their end. Each of the servers updates its column-wise parity blocks using M'_{CRS} as described in Section 4.3.1.

For each secondary server, the client also computes the changes in the existing tags by taking $t = \texttt{ctr}$ and $\tilde{k}_a = \tilde{k}$ in Eqn. 4.3 and sends them to the secondary servers. The secondary servers update the tags on the existing column-wise parity blocks accordingly.

• **Verify**$(Q, \Pi, \texttt{fid}, sk, \texttt{ctr}, \tilde{k}, t)$: The procedure is the same as the procedure **Verify** described in Section 4.1 except the following. Instead of using Eqn. 4.2, the client checks whether

$$\sigma_j \stackrel{?}{=} \sum_{i \in I, i \leqslant \tilde{k}} \nu_i f_{k_{prf}}(i, j, 0) + \sum_{i \in I, i > \tilde{k}} \nu_i f_{k_{prf}}(i, j, \texttt{ctr}) + \alpha\mu_j \bmod p. \tag{4.6}$$

Number of Parity Blocks in a Column-wise Codeword In our scheme, we have kept $\tilde{s}$ (the number of parity blocks in a column-wise codeword) fixed throughout a series of append operations in order to avoid inserting rows in the Cauchy submatrix of M'_{CRS} (otherwise, each server has to touch all the existing data blocks to compute the newly inserted parity blocks). On the other hand, if the value of $\tilde{s}$ becomes very small compared to the increasing value of $\tilde{k}$, then the decoding probability of the data blocks in the codeword decreases significantly. To address this trade-off, we take a parameter ϵ_p for our system. If the fraction of parity blocks in a codeword drops below ϵ_p, the client adds some parity blocks to each column-wise codeword to restore the fraction well above the threshold (during the procedure Redistribute).

Chapter 5

Implementation

Let n (total number of servers) and k (number of primary servers) be the system parameters. We assume that a client wants to distribute it's data file F among n servers. It chooses k primary servers to store the data blocks of F and $s = n - k$ secondary servers to store the parity blocks. For distribution of data to cloud servers, we use modules of the library Jerasure-1.2 [20]. We will describe about implementation details through following sections.

5.1 Jerasure-1.2 Library

To distribute file into servers, we use error-correcting codes (erasure codes). We use Jerasure-1.2, a library in C/C++ that supports erasure coding applications. Jerasure supports a *horizontal mode* of erasure codes. We assume that we have k servers that hold data. To that, we will add s servers whose contents will be calculated from the original k servers. If the erasure code is a Maximum Distance Separable (MDS) code, then the entire system will be able to tolerate the loss of any s servers. In our distributed systems, Jerasure is very effective tool for fault tolerance.

As shown in Figure 5.1, the encoding process takes the original k data servers, and from them calculates s coding servers. The decoding process takes the collection of

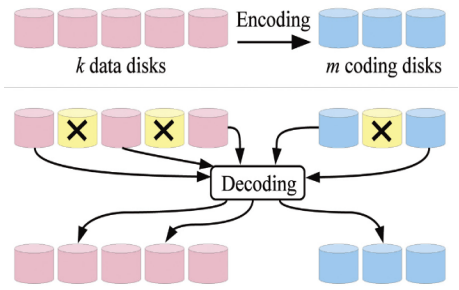

Figure 5.1: Encoding and Decoding process

$(k+s=n)$ total servers with erasures, and from the surviving servers recalculates the contents of the original k data servers. Most codes have a third parameter w, which is the *word size*. The description of a code views each device as having w bits worth of data. The data servers are denoted D_0 through D_{k-1} and the coding servers are denoted C_0 through C_{s-1} . Each server D_i or C_j holds w bits, denoted $d_{i,0}, ..., d_{i,w-1}$ and $c_{j,0}, ..., c_{j,w-1}$. In reality, servers hold megabytes of data.

When $w \in 8, 16, 32$, we can consider each collection of w bits to be a byte, short word or word respectively. Consider the case when $w = 8$. Let us say each server to hold B bytes (example). The first byte of each coding server will be encoded with the first byte of each data server. The second byte of each coding server will be encoded with the second byte of each data server. And so on upto B many times.

We have given brief information of routines we used in implementation here.

5.1.1 Galois Field Arithmetic

As we using $w \in 8, 16, 32$, we would be working in $GF(2^8)$, $GF(2^{16})$ or $GF(2^{32})$. The Galois Field Arithmetic for these fields are integrated with Jerasure library already. The files **galois.h** and **galois.c** contain procedures for Galois Field arithmetic in $GF(2^w)$ for $1 \leq w \leq 32$. Following are some procedures we used from *galois.h* and *galois.c*.

Arithmetic Routines

- **galois_single_multiply(int a, int b, int w)** and **galois_single_divide(int a, int b, int w)**. These perform multiplication and division on single elements a and b of $GF(2^w)$.
- **galois_w08_region_multiply(char *region, int multby, int nbytes, char *r2, int add)**. This multiplies an entire region of bytes by the constant multby in $GF(2^8)$.If r2 is NULL then region is overwritten. Otherwise, if add is zero, the products are placed in r2. If add is non-zero, then the products are XOR'd with the bytes in r2.
- **galois_w16_region_multiply()** and **galois_w32_region_multiply()** are identical to **galois_w08_region_multiply()**, except they are in $GF(2^{16})$ and $GF(2^{32})$ respectively.

There are some other procedures, but we have not used them. Galois field arithmetic is used in all other parts of library e.g. encoding, decoding procedures.

5.1.2 Basic Routines

The files **jerasure.h** and **jerasure.c** implement procedures that are common to many aspects of coding. We use some of the procedures from these routines.

Parameters Some of the important parameters used in library are described below.

- **int k**: The number of data servers.
- **int m**: The number of coding servers.
- **int w**: The word size of the code.
- **int size**: The total number of bytes per server to encode/decode. This must be a multiple of **sizeof(long)**.
- **int *matrix**: This is an array with $k \times m$ elements that represents the coding matrix i.e. the last m rows of the distribution matrix. Its elements must be between 0 and $2^w - 1$.
- **char **data ptrs**: This is an array of k pointers to size bytes worth of data. Each of these must be long word aligned.
- **char **coding ptrs**: This is an array of m pointers to size bytes worth of coding data. Each of these must be long word aligned.
- **int *erasures**: This is an array of id's of erased devices. Id's are numbers between 0 and $k + m - 1$.

Encoding Routines Encoding routines used by us are described below.

- **void jerasure_matrix_encode(k, m, w, matrix, data_ptrs, coding_ptrs, size)**. This encodes with a matrix in $GF(2^w)$. w must be $\in 8, 16, 32$.

Decoding Routines Encoding routines used by us are described below.

- **jerasure_matrix_decode(k, m, w matrix, erasures, data_ptrs, coding_ptrs, size)**: This decodes using a matrix in $GF(2^w)$, $w \in 8, 16, 32$. This works by creating a decoding matrix and performing the matrix/vector product, then re-encoding any erased coding devices.

Matrix Routines Some basic matrix routines are used by us.

- **int jerasure_invert_matrix(int *mat, int *inv, int rows, int w)**: This inverts a $(rows \times rows)$ matrix in $GF(2^w)$. It puts the result in *inv*.

5.1.3 Cauchy Reed-Solomon Coding Routines

The files **cauchy.h** and **cauchy.c** implement procedures for Cauchy Reed-Solomon coding.

- **int *cauchy_original_coding_matrix(k, m, w)**: This allocates and returns the originally defined Cauchy matrix.
- **int *cauchy_xy_coding_matrix(k, m, w, int *X, int *Y)**: This allows the user to specify sets X and Y to define the matrix. Set X has m elements of $GF(2^w)$ and set Y has k elements. Neither set may have duplicate elements and $X \cap Y = \emptyset$.

5.2 Schema of Implementation

We have described routines we used from library in earlier section. Now, we will describe how we implemented our proposed scheme using these routines. Our scheme for append-only contains procedures **Setup**, **Outsource**, **Challenge**, **Prove**, **Verify**, **Append**. We have described these procedures thoroughly already in chapter 4.

5.2.1 Preprocessing

Preprocessing the data contains procedures **Setup**, and **Outsource**.

The *setup* procedure contain preliminary secret key selection, pseudorandom function (PRF) key selection. The *Outsource* procedure contain encoding of the file and tag creation.

Key Generation For generation of random keys required in *setup* phase, we have used *Pseudorandom number generator* ($PRNG$). A PRNG can be started from an arbitrary initial state using a *seed state*. The client chooses a random prime p. The client selects an element $\alpha \xleftarrow{R} \mathbb{Z}_p$ and a PRF key $k_{prf} \xleftarrow{R} \mathcal{K}_{prf}$. Her secret key sk is the pair (α, k_{prf}).

Encoding Jerasure-1.2 provides different encoding and decoding routines. For encoding the data, Jerasure creates **Generating Distribution Matrices** using different methods e.g. *Vandermonde Distribution Matrices*, *Cauchy Reed-Solomon Coding*. Jerasure gives very fast results for arithmetic of elements in $GF(2^8), GF(2^{16}), GF(2^{32})$. We generally take elements from $GF(2^8)$.

The client chooses the data file F to store on n servers. She chooses k *primary* servers $S_1, S_2, \ldots, S_k$ to store the data blocks of F and $s = n - k$ *secondary* (or redundant) servers $S_{(k+1)}, S_{(k+2)}, \ldots, S_n$ to store the parity (or redundant) blocks.

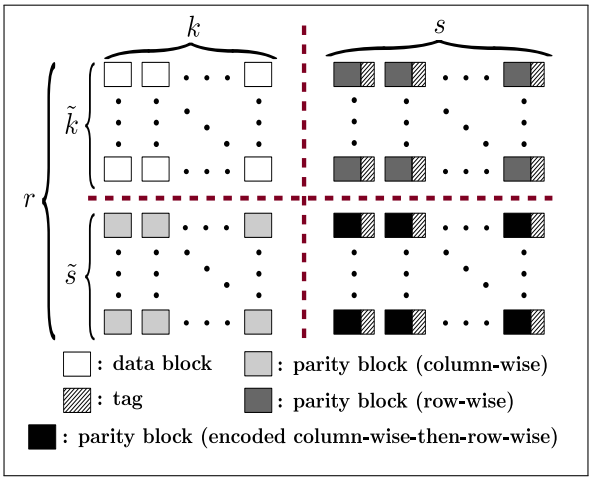

Figure 5.2: Distributed storage structure.

She divides the file F as $\{m_{ij}\}_{i\in[1,\tilde{k}],j\in[1,k]}$, where each data block $m_{ij} \in \mathbb{Z}_p$ for all $i \in [1,\tilde{k}]$, $j \in [1,k]$. Figure 5.2 gives an overview of the storage structure for F distributed among the primary and the secondary servers.

The processed file is split into blocks, and each block is split into sectors. Each element of sector is in either of above specified fields. We have used block as consists of 8 sectors.

In our setting, Encoding of the data blocks are done row-wise (inter-server) and column-wise (intra-server).

For row-wise encoding, we are using **Cauchy Reed-Solomon Encoding** to distribute data over servers. For purpose, we use procedure *cauchy_original_coding_matrix* described in above section.

For column-wise encoding, we are using **Extending Cauchy Reed-Solomon code** described in our proposed scheme to create Generating Distribution Matrix. We are using *cauchy_xy_coding_matrix* procedure described in above section.

These procedures will give us Generating Distribution Matrix. Client side will use matrix to distribute the main file (data). So, using encoding, client file gets distributed at k primary servers and s secondary servers, where total servers are $n = k + s$.

Tag creation Tag creation will be done as described in our proposed scheme for append-only data in previous chapter. The client computes authentication tags for parity blocks as per construction m_{ij} $(i \in [1,r], j \in [k+1,n])$

$$\sigma_{ij} = f_{k_{prf}}(i,j,0) + \alpha m_{ij} \bmod p. \tag{5.1}$$

Tag calculation uses Galois Field Arithmetic described in previous section. We use *galois_single_multiply* for multiplication in chosen $GF(2^w)$. Tags are calculated for respective sectors of each block and stored with respective block.

After this preprocessing procedures, distributed files and corresponding tags are outsourced to respective servers.

5.2.2 Auditing

Auditing part contains procedures **Challenge**, **Prove**, and **Verify** which we already described in previous chapter.

Challenge This part implements **Challenge** procedure. In audit part, the client selects I, a random l-element subset of $[1, r]$ and client side generates randomly challenge set $Q = \{(i, \nu_i)\}_{i \in I}$, where each $\nu_i \xleftarrow{R} \mathbb{Z}_p$ using PRNG. These challenges (queries) are sent to server side by client side.

Prove This part implements **prove** procedure. Server side receives challenge from client side. Then, all n servers calculate aggregated codeword block. Lets say, the j-th cloud server S_j computes and sends $\mu_j = \sum_{i \in I} \nu_i m_{ij} \in \mathbb{Z}_p$ to the client side, for all $j \in [1, n]$.

All secondary s servers calculate aggregated tag. The j-th secondary server sends $\sigma_j = \sum_{i \in I} \nu_i \sigma_{ij} \in \mathbb{Z}_p$ to the client, for all $j \in [k+1, n]$.

This all combined codeword and tag constitutes proof π by server side to client side. The proof calculation uses Galois Field Arithmetic described in previous section. We use *galois_single_multiply* for multiplication in chosen $GF(2^w)$.

Verify This part implements **verify** procedure. Client side receives proof π from server side and already have challenge set Q.

It then checks whether received aggregated codeword by server side is valid or not. We use procedure *cauchy_original_coding_matrix* for verification of codeword.

Then, for each $j \in [k+1, n]$, the client checks whether

$$\sigma_j \stackrel{?}{=} \sum_{i \in I} \nu_i f_{k_{prf}}(i, j, 0) + \alpha \mu_j \bmod p. \tag{5.2}$$

If this verification process fails, then the proof received is wrong and we output 0. Tag verification uses Galois Field Arithmetic described in previous section. We use *galois_single_multiply* for multiplication in chosen $GF(2^w)$.

5.2.3 Corruption and Reconstruction

Data corruption is among the most common data errors. Data corruption happens when data is intentionally or unintentionally changed from its original, correct form. Corruption can be systematic or random, and even a small change can fundamentally render a file useless. In these cases, data server cannot access the specific sector or specific block or specific whole server.

We have k primary and s secondary servers totaling $n = k + s$ servers. So, as we are using *MDS* codes, we can reconstruct any s blocks out of k blocks in single row. So, even if whole s servers are corrupted, we can reconstruct them back. If servers find out some data blocks are inaccessible (corrupted) , then servers start the process of reconstruction.

For reconstruction process, server use the *row-wise* reconstruction process. Lets say $c \leq s$ blocks in one row got corrupted, then selecting any k blocks out of remaining $k \leq (n - c)$, we can reconstruct all blocks in corresponding row back. We choose any k blocks out of remaining blocks and take corresponding rows of Generating Distribution Matrix M_{CRS} *(Cauchy Reed-solomon)*. After getting $k \times k$ matrix, we take inverse of that matrix. As all rows of Generating Distribution Matrix are independent, then any $k \times k$ matrix made out of $n \times k$ matrix has inverse. We multiply chosen k blocks with the inverse of above described $k \times k$ matrix and we get back all data blocks return.

Mathematically, let G $(k \times n)$ be the generator matrix and vector $\vec{D} = \{d_0, d_1, ..., d_{k-1}\}$ be the data stripe and vector $\vec{C} = \{c_0, c_1, ..., c_{s-1}\}$ be the coding stripe.

$$G^T \times D^T = \mu^T \tag{5.3}$$

where, a vector $\vec{\mu} = \{d_0, d_1, ..., d_{k-1}, c_0, c_1, ..., c_{s-1}\}$ is the final codeword stripe.

Suppose, at most s characters (symbols) of codeword stripe got corrupted. Let, $\vec{\omega}$ be surviving vector. Let, B $(k \times k)$ be the G^T with deleted rows of corresponding corrupted characters. Then, it implies,

$$B \times D^T = \omega^T \tag{5.4}$$

$$B^{-1} \times B \times D^t = B^{-1} \times \omega^T \tag{5.5}$$

$$D^t = B^{-1} \times \omega^T \tag{5.6}$$

Finally, we get data stripe back.

Sometimes, it is possible that more than s i.e. $c > s$ blocks in one row get corrupted. In that case, reconstruction using row-wise parity is not possible. We use column-wise parity in this case for corrupted block numbers. In this case, we construct Generating Distribution Matrix M'_{CRS} *(Extending Cauchy Reed-solomon)*. Similar process as row reconstruction is followed. We reconstruct the whole server back. Following process consist reconstructing row earlier using row-wise parity.

This reconstruction process requires Matrix, Galois Field Arithmetic as well as Decoding routines. We will use *jerasure_invert_matrix()* as well as *jerasure_matrix_decode()* procedures for reconstruction. Also, We use *galois_single_multiply* for multiplication in chosen $GF(2^w)$. We might require *jerasure_matrix_encode()* to re-encode data back if some parity block is present in c corrupted blocks.

Example in Figure 5.3 shows corruption of encoded random data (in Hexadecimal format) and decoding (reconstruction) of data.

```
ndn@ndn-HP-PC:~/thesis/example_7$ ./cauchy_01 6 4 8
Last m rows of the Distribution Matrix:

  1   1   1   1   1   1
  1 225 151 172  82 200
  1 166 196 238  83 146
  1 123 245 143 244 142
Encoding Complete:

Data                            Coding
D0 : 53 61 6d 70 6c 65 20 23    C0 : 6f 28 0c 04 24 ff fd e3
D1 : 31 48 69 20 54 68 65 72    C1 : 4d a9 1f c7 a6 c7 eb e0
D2 : 65 77 68 61 74 20 64 6f    C2 : 21 0e 44 9a 05 af 17 66
D3 : 20 79 61 20 77 61 6e 74    C3 : b9 3c ef bd 35 cf f5 8e
D4 : 20 66 6f 72 20 6e 6f 74
D5 : 68 69 6e 67 3f dd dd dd

Erased 4 random devices:

Data                            Coding
D0 : 00 00 00 00 00 00 00 00    C0 : 6f 28 0c 04 24 ff fd e3
D1 : 00 00 00 00 00 00 00 00    C1 : 4d a9 1f c7 a6 c7 eb e0
D2 : 65 77 68 61 74 20 64 6f    C2 : 00 00 00 00 00 00 00 00
D3 : 20 79 61 20 77 61 6e 74    C3 : b9 3c ef bd 35 cf f5 8e
D4 : 00 00 00 00 00 00 00 00
D5 : 68 69 6e 67 3f dd dd dd

State of the system after decoding:

Data                            Coding
D0 : 53 61 6d 70 6c 65 20 23    C0 : 6f 28 0c 04 24 ff fd e3
D1 : 31 48 69 20 54 68 65 72    C1 : 4d a9 1f c7 a6 c7 eb e0
D2 : 65 77 68 61 74 20 64 6f    C2 : 21 0e 44 9a 05 af 17 66
D3 : 20 79 61 20 77 61 6e 74    C3 : b9 3c ef bd 35 cf f5 8e
D4 : 20 66 6f 72 20 6e 6f 74
D5 : 68 69 6e 67 3f dd dd dd

ndn@ndn-HP-PC:~/thesis/example_7$
```

Figure 5.3: Example of Encoding and Decoding, k=6, m=4, w=8

5.2.4 Append

This part implements **append** procedure.

Let us say, we had total r blocks earlier in every server. Out of r blocks, r_1 blocks are of data, and r_2 blocks are of *column-wise parity* s.t. $r = r_1 + r_2$.

We append new row of data. Using M_{CRS}, we calculate parity blocks for this newly added blocks. So, each n servers, gets added one new block and contains now $r_1 + 1$ data rows. We construct respective tags for newly added parity blocks. These tags are placed respective to new parity blocks in parity servers.

This append will also affect our column-wise parity of every sever and their respective tags. Let us say, (m_{ij}, σ_{ij}) and (m'_{ij}, σ'_{ij}) be the block-tag pairs for the i-th block ($i \in [r_1 + 1, r]$) of S_j, $j \in [1, n]$ after and before the t-th append, respectively. So, we have $\sigma_{ij} = f_{k_{prf}}(i, j, t) + \alpha m_{ij} \bmod p$ and $\sigma'_{ij} = f_{k_{prf}}(i, j, t-1) + \alpha m'_{ij} \bmod p$. We define $\Delta_\sigma = \sigma_{ij} - \sigma'_{ij} \bmod p$ and $\Delta_m = m_{ij} - m'_{ij} \bmod p$.

Servers calculate Δ_m of relevant parity blocks on server side and add it to previous column-wise parity blocks in respective servers. Client sends Δ_σ of related parity blocks to secondary servers. Servers add Δ_σ to respective tags. The append process requires *galois_single_multiply* for multiplication in chosen $GF(2^w)$. We might require *jerasure_matrix_encode()* to calculate parity blocks for newly added row.

5.2.5 File Retrieval

Client distributes its data to multiple cloud servers. But, afterwards client might require data physically. So, client need to download the outsourced file back. At such time, client might want to retrieve its file (data) back.

For static data, we store the hash of file before preprocessing. After downloading the file from servers, we calculate the hash value of downloaded file. If hash of newly downloaded file matches previous hash value, then file we retrieved is correct.

For append data, above method is not useful. In this case, we check if every row is correct codeword or not. In other words, we audit every row of servers. If we get correct audit proof, then we say file retrieved is correct.

Chapter 6

Security and Performance Analysis

6.1 Security Model

We assume that a client (cloud user) wants to distribute it's data file F among n servers. It chooses k *primary* servers $S_1, S_2, \ldots, S_k$ to store the data blocks of F and $s = n-k$ *secondary* (or redundant) servers $S_{(k+1)}, S_{(k+2)}, \ldots, S_n$ to store the parity (or redundant) blocks. Data blocks are encoded *row-wise* (inter-server or dispersal code) and *column-wise* (intra-server or server code) using Cauchy Reed-Solomon (CRS) codes.

We follow a security model similar to that discussed in HAIL [4] but for append-only data. After the client initially uploads F to the servers, the lifetime of the system is split into some time intervals called *epochs*. For a parameter b, a PPT adversary $\mathcal{A}$ is modeled as *mobile* (i.e., in each epoch it can corrupt up to b servers chosen arbitrarily) and fully *Byzantine* (i.e., it can arbitrarily modify or delete any part of the storage in any server it corrupts). An epoch consists of four phases: an *append* phase (the client appends data blocks to the existing file residing on the servers), a *corruption* phase ($\mathcal{A}$ chooses a fresh set of b servers to corrupt), an *audit* phase (the client challenges the servers via spot checking) and a *remediation* phase (the client checks if some corrupted servers provide incorrect responses above a certain threshold fraction ϵ_q). A bound on b is discussed in Section 4.1. In the remediation phase, if the fraction of corruptions exceeds ϵ_q for some server, the client reads all the file shares from each server and tries to decode F. We define the distributed cloud storage scheme to be secure if, for any $\mathcal{A}$, the client correctly decodes F with high probability.

6.2 Security of scheme for static data

We consider the same security model discussed in Section 6.1, except that the *append* phase is not present in an epoch. Security of the scheme for static data is derived

from [23] in the same way as in HAIL [4]. We note that an adversary can corrupt a particular dispersal codeword in HAIL. For example, the adversary can replace the codeword in the second row with that in the first row in a span of $f = \lceil \frac{n}{b} \rceil$ epochs. The probability that the second row is not challenged in any of these f epochs is $(1 - \frac{l}{r})^f$ that is quite high. During this span only, the codeword in the second row can be detected as invalid; it is a valid codeword, otherwise. Our scheme prevents this attack by embedding row-indices in the tags using Eqn. 4.1 and by verifying them using Eqn. 4.2.

6.3 Security of scheme for append-only data

Security of the scheme for append-only data is the same as that in the scheme for static data described in Section 4.1, except that the parity blocks in each server are updated for each append. We observe that the first $\tilde{k}$ rows are never updated in the scheme; only the parity rows (i.e., column-wise parity blocks for each server) are updated for an append. If the servers retain the i-th ($i \in [\tilde{k} + 1, r]$) row of parity blocks with older contents (and tags for an older `ctr` value), the client can easily detect this anomaly while verifying the proof (using Eqn. 4.6) as the latest counter value would not match with `ctr`.

6.4 Performance Analysis

Symbol	Definition
n	Number of total servers
k	Number of Primary servers
s	Number of Secondary servers
$\tilde{s}$	Number of column-wise parity rows

Table 6.1: Notations used for analysis.

We have implemented the scheme for static data as given in section 4.1. Also, we extended this scheme to append-only (dynamic) data. Currently, We have implemented the scheme with encoding of files in C language using *Jerasure* routines and ran our experiments on an Intel Core 3 processor running at 2.40 GHz. We have currently written our code on Ubuntu 16.04.2 LTS linux machine.

In our experiments, we have used fixed Cauchy Reed-Solomon code ($n = 8, k = 6, s = 2$) for horizontal parity over $GF(2^8)$. For column-wise parity, we use Extended Cauchy Reed-Solomon code with $\tilde{s} = 10$. Currently, we are in primitive stages of experiments which are giving successful results.

We ran some timing experiments for computation of tags (MAC) against increasing file size. Results for these experiments are shown in the Figure 6.1.

We ran some experiments for reconstruction of corrupted data. Figure 6.2 shows the results for reconstruction of whole m corrupted (erasure) servers against increasing file size.

As we increase the file size, we check the time require for auditing which consists challenge, prove and verify phases. Figure 6.3 shows the time for audit verification for a fixed number of queries against increasing file size.

We varied number of audit queries and check the time for audit verification. Figure 6.4 shows the number of queries against time for audit verification.

Observations

– We notice time for computation of MAC (tags) is gradually increasing as we start to increase the size of file.

– Time for reconstruction of corrupted server is monotonically increasing while increase in file size.

– Increment in file size causing the increment in time for audit verification.

– As we are increasing the number of queries, the time for computation of proof is increasing which in resulting in time increase in audit verification.

– We deduce that file size and challenge size are very important factors for timing evaluations. As sample size we have taken for variables is small scale, we can see small fluctuations in graphs.

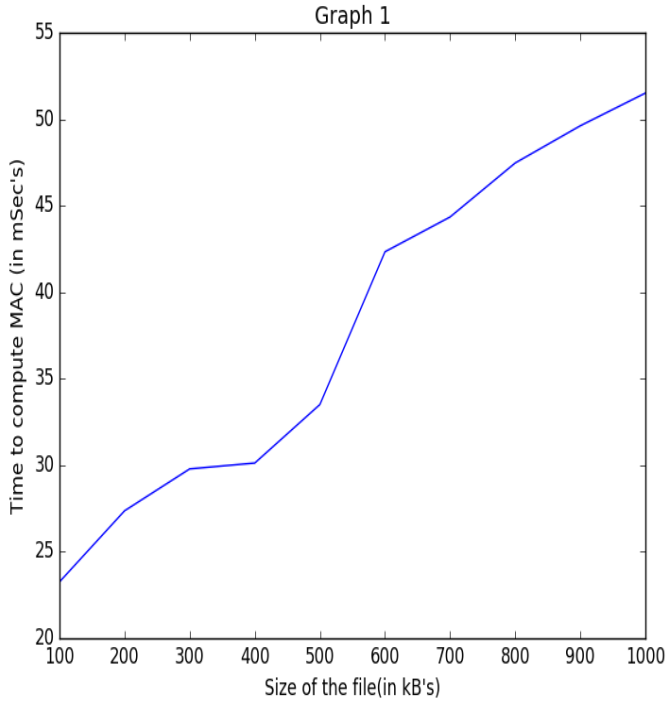

Figure 6.1: Size of File vs. Time to compute MAC (tag)

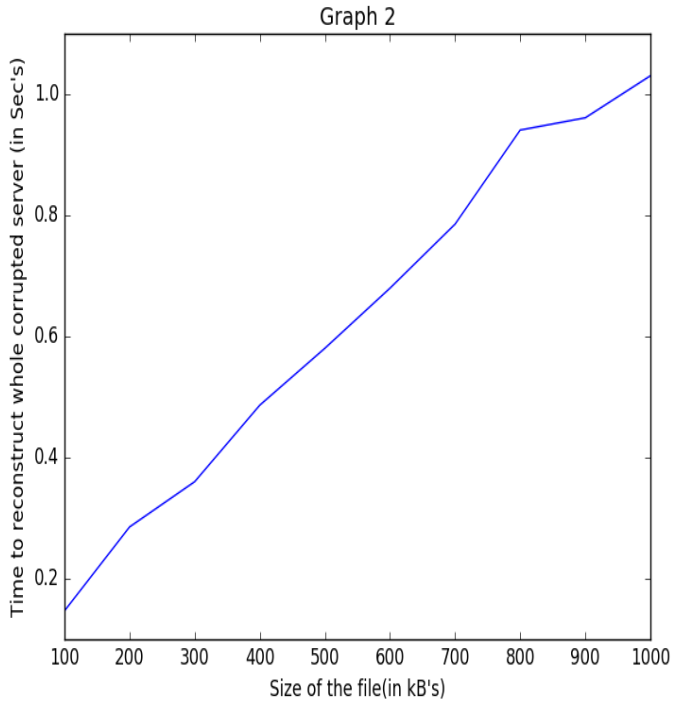

Figure 6.2: Size of File vs. Time to reconstruct whole corrupted server

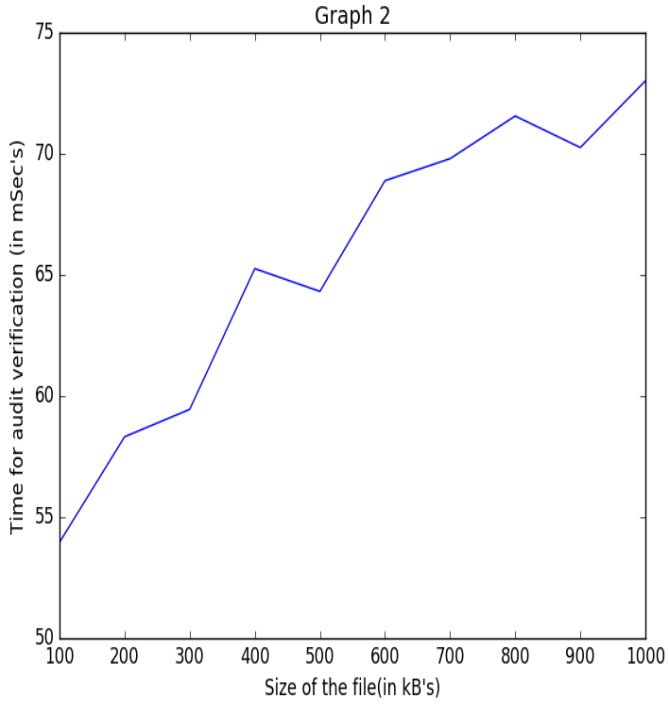

Figure 6.3: Size of File vs. Time for audit verification

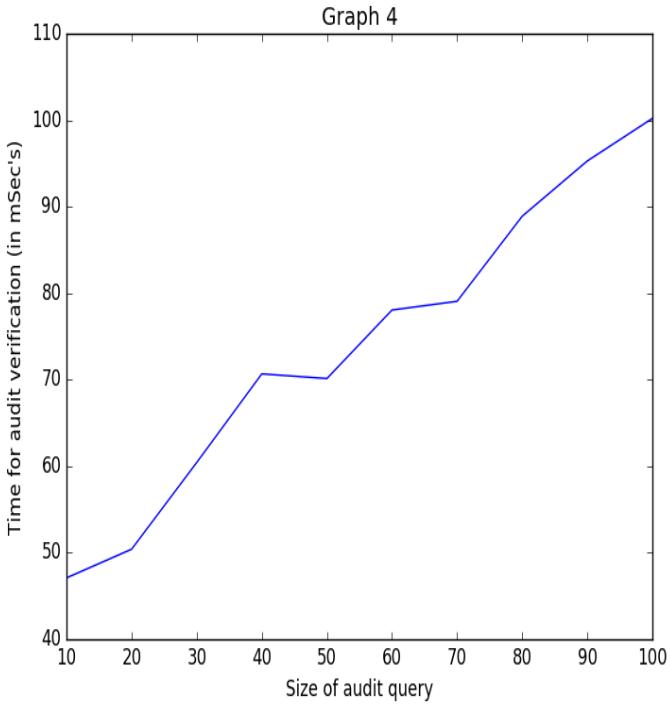

Figure 6.4: Size of Audit query vs. Time for audit verification

Chapter 7

Future Work and Conclusion

In this work, we have implemented a distributed secure cloud storage scheme for static data. Then, we have extended this scheme to accommodate append-only data such that the client can efficiently append data after the initial data outsourcing. The scheme provides a partial solution (for append-only data) to the problem of designing a distributed secure cloud storage *achieving PoR guarantees for dynamic data*. To the best of our knowledge, there is no work found in the literature that addresses this problem for generic dynamic data or append-only data.

We have implemented scheme for append-only data. We are investigating performance analysis of our system thoroughly and thrive for improvements in our work. As the work we have done is completely new, the experiments done are still on smaller scale. We are trying to reflect the scheme on cloud platforms (large scale). We believe that our scheme help cloud community to reaching remarkable approaches.

Bibliography

[1] G. Ateniese, R. C. Burns, R. Curtmola, J. Herring, L. Kissner, Z. N. J. Peterson, and D. X. Song. Provable data possession at untrusted stores. In *ACM Conference on Computer and Communications Security, CCS 2007*, pages 598–609, 2007.

[2] G. Ateniese, R. D. Pietro, L. V. Mancini, and G. Tsudik. Scalable and efficient provable data possession. In *International Conference on Security and Privacy in Communication Networks, SECURECOMM 2008*, page 9, 2008.

[3] J. Blömer, M. Kalfane, R. Karp, M. Karpinski, M. Luby, and D. Zuckerman. An XOR-based erasure-resilient coding scheme, August 1995. Technical Report TR-95-048, International Computer Science Institute.

[4] K. D. Bowers, A. Juels, and A. Oprea. HAIL: A high-availability and integrity layer for cloud storage. In *ACM Conference on Computer and Communications Security, CCS 2009*, pages 187–198, 2009.

[5] K. D. Bowers, A. Juels, and A. Oprea. Proofs of retrievability: Theory and implementation. In *ACM Cloud Computing Security Workshop, CCSW 2009*, pages 43–54, 2009.

[6] D. Cash, A. Küpçü, and D. Wichs. Dynamic proofs of retrievability via oblivious RAM. In *Advances in Cryptology - EUROCRYPT 2013*, pages 279–295. Springer Berlin Heidelberg, 2013.

[7] B. Chen, R. Curtmola, G. Ateniese, and R. C. Burns. Remote data checking for network coding-based distributed storage systems. In *ACM Cloud Computing Security Workshop, CCSW 2010*, pages 31–42, 2010.

[8] R. Curtmola, O. Khan, R. C. Burns, and G. Ateniese. MR-PDP: Multiple-replica provable data possession. In *IEEE International Conference on Distributed Computing Systems, ICDCS 2008*, pages 411–420, 2008.

[9] A. G. Dimakis, B. Godfrey, M. J. Wainwright, and K. Ramchandran. Network coding for distributed storage systems. In *IEEE International Conference on Computer Communications, INFOCOM 2007*, pages 2000–2008, 2007.

[10] Y. Dodis, S. P. Vadhan, and D. Wichs. Proofs of retrievability via hardness amplification. In *Theory of Cryptography Conference, TCC 2009*, pages 109–127, 2009.

[11] C. C. Erway, A. Küpçü, C. Papamanthou, and R. Tamassia. Dynamic provable data possession. In *ACM Conference on Computer and Communications Security, CCS 2009*, pages 213–222, 2009.

[12] J. Fingas. Amazon outage breaks large parts of the internet, February 2017. `https://www.engadget.com/2017/02/28/amazon-aws-outage/`.

[13] T. A. S. Foundation. Apache Hadoop, June 2016.

[14] C. Huang, H. Simitci, Y. Xu, A. Ogus, B. Calder, P. Gopalan, J. Li, and S. Yekhanin. Erasure coding in windows azure storage. In *USENIX Annual Technical Conference, ATC 2012*, pages 15–26, 2012.

[15] A. Juels and B. S. Kaliski, Jr. PORs: Proofs of retrievability for large files. In *ACM Conference on Computer and Communications Security, CCS 2007*, pages 584–597, 2007.

[16] B. Laurie, A. Langley, and E. Kasper. Certificate transparency, June 2013. `https://tools.ietf.org/html/rfc6962`.

[17] A. Le and A. Markopoulou. NC-audit: Auditing for network coding storage. In *International Symposium on Network Coding, NetCod 2012*, pages 155–160, 2012.

[18] M. Lillibridge, S. Elnikety, A. Birrell, M. Burrows, and M. Isard. A cooperative internet backup scheme. In *USENIX Annual Technical Conference, ATC 2003*, pages 29–41, 2003.

[19] K. Omote and T. T. Phuong. DD-POR: Dynamic operations and direct repair in network coding-based proof of retrievability. In *Computing and Combinatorics - 21st International Conference, COCOON 2015*, pages 713–730, 2015.

[20] J. S. Plank, J. Luo, C. D. Schuman, L. Xu, and Z. Wilcox-O'Hearn. A performance evaluation and examination of open-source erasure coding libraries for storage. In *USENIX Conference on File and Storage Technologies, FAST 2009*, pages 253–265, 2009.

[21] T. J. E. Schwarz and E. L. Miller. Store, forget, and check: Using algebraic signatures to check remotely administered storage. In *IEEE International Conference on Distributed Computing Systems, ICDCS 2006*, page 12, 2006.

[22] B. Sengupta, N. Nikam, S. Ruj, S. Narayanamurthy, and S. Nandi. A distributed secure cloud storage for append-only data, February 2017. manuscript.

[23] H. Shacham and B. Waters. Compact proofs of retrievability. In *Advances in Cryptology - ASIACRYPT 2008*, pages 90–107, 2008.

[24] H. Shacham and B. Waters. Compact proofs of retrievability. *Journal of Cryptology*, 26(3):442–483, 2013.

[25] E. Shi, E. Stefanov, and C. Papamanthou. Practical dynamic proofs of retrievability. In *ACM Conference on Computer and Communications Security, CCS 2013*, pages 325–336, 2013.

[26] Q. Wang, C. Wang, J. Li, K. Ren, and W. Lou. Enabling public verifiability and data dynamics for storage security in cloud computing. In *European Symposium on Research in Computer Security - ESORICS 2009*, pages 355–370, 2009.

[27] Wikipedia. RAID. `http://en.wikipedia.org/wiki/RAID`.

[28] Y. Zhu, H. Hu, G. Ahn, and M. Yu. Cooperative provable data possession for integrity verification in multicloud storage. *IEEE Transaction on Parallel and Distributed Systems*, 23(12):2231–2244, 2012.