

Title	クラウド・コンピューティングのセキュリティに関する研究
Author(s)	Tran, Thao Phuong
Citation	
Issue Date	2015-09
Type	Thesis or Dissertation
Text version	ETD
URL	http://hdl.handle.net/10119/12965
Rights	
Description	Supervisor:面 和成, 情報科学研究科, 博士

A STUDY ON SECURITY FOR CLOUD COMPUTING

By TRAN, THAO PHUONG

submitted to
Japan Advanced Institute of Science and Technology,
in partial fulfillment of the requirements
for the degree of
Doctor of Philosophy

Written under the direction of
Associate Professor Kazumasa Omote

September, 2015

A STUDY ON SECURITY FOR CLOUD COMPUTING

By TRAN, THAO PHUONG (1220210)

A thesis submitted to
School of Information Science,
Japan Advanced Institute of Science and Technology,
in partial fulfillment of the requirements
for the degree of
Doctor of Information Science
Graduate Program in Information Science

Written under the direction of
Associate Professor Kazumasa Omote

and approved by
Associate Professor Kazumasa Omote
Professor Atsuko Miyaji
Professor Kazuhiro Ogata
Professor Ryuhei Uehara
Doctor Kiyomoto Shinsaku

July, 2015 (Submitted)

ABSTRACT

Keywords: cloud computing, proofs of retrievability, secret sharing scheme, network coding, Slepian-Wolf coding.

Since amount of data is increasing exponentially, data storage and management become burdensome tasks of the data owner. To reduce the burdens for the data owners, the concept of remote storage known as cloud has been proposed. A cloud is considered as a service through which the clients can use to publish, access, manage and share their data remotely and easily from anywhere via the Internet. Although data outsourcing reduces storage burden for the client, this method still has a problem that the service provider is typically not fully trusted. Thus, this model introduces numerous interesting research challenges: (i) data privacy, (ii) data availability and (iii) data integrity. Data confidentiality consists of the two research approaches: the cryptographic approach and the information-theoretic approach. In this study, we focus on integrity, availability and information-theoretic confidentiality. We choose the information-theoretic approach because our security analysis derives purely from information theory. Our goal is to construct a practical and secure cloud system. Based on this goal, we are interested in two research directions: Proof Of Retrievability (POR) and Secret Sharing Scheme (SSS).

The POR has been proposed to allow the client to check whether his/her data stored in the servers is available, intact and is always retrievable. Based on the POR protocol, four common techniques are used: replication, erasure coding, ORAM and network coding. In this study, we focus on the network coding because: it achieve better storage cost compared with replication, and better computation and communication costs compared with erasure coding and ORAM. Although many network coding-based PORs have been proposed, the efficiency and practicality have not been addressed simultaneously.

The SSS is a method for protecting distributed file systems against data leakage and data loss. In this scheme, the secret is encoded into a number of shares. The shares are then distributed among a group of participants

where each participant holds a share of the secret. The secret can be only reconstructed when a sufficient number of shares are reconstituted. Although many SSSs are introduced, they have not achieved an optimal share size and have not supported the share repair feature.

In this dissertation, we propose three schemes, named the MD-POR (Multi-client and Direct repair for POR), DD-POR (Dynamic operation and Direct repair for POR) and SW-SSS (Slepian-Wolf coding-based SSS).

The MD-POR is our main proposed POR which has the following contributions: (i) The scheme can support direct repair feature. This means that if a corrupted server is detected, the healthy servers are required to provide their coded blocks directly to the new server. The new server can verify the provided coded blocks and can compute the new coded blocks for itself without disturbing the client. This mechanism can reduce the communication cost and the burden for the client; (ii) Multiple clients who own different secret keys can participate in the system. Their data are mixed together without losing the data confidentiality of individual clients; (iii) The scheme is constructed using symmetric key setting for the efficiency; and (iv) The scheme supports public authentication. This means that not only the client but also any entity who has a given information can check the cloud servers while learning nothing about the secret key of each client. We employ a Third Party Auditor (TPA) on behalf of the clients to check the servers periodically. By delegating the responsibility of checking the servers to the TPA, the clients are free of the burden of checking the servers.

The DD-POR scheme is an improvement of the MD-POR scheme. Concretely, this scheme can support dynamic operations unlike the MD-POR scheme. The client not only can read the data but also can modify, insert, and delete the data. However, the DD-POR scheme is a partial improvement of the MD-POR scheme because in this DD-POR scheme, we can only deal with a single client instead of multiple clients as the MD-POR scheme. Furthermore, the DD-POR does not deal with the public authentication as the MD-POR scheme. The DD-POR scheme has the following contributions: (i) This scheme can support direct repair feature like the MD-POR scheme. When a server is corrupted, the healthy servers will provide their coded blocks and tags directly to the new server without

sending them back to the client. Then, the new server can check them, and can compute the new coded blocks and the tags for itself; (ii) Unlike the MD-POR the client not only can check and retrieve the data, but also can perform dynamic operations such as modification, insertion and deletion on the data stored in the servers; and (iii) The scheme is constructed using symmetric key setting for the efficiency.

The SW-SSS scheme, we show that the Slepian-Wolf Coding, which is used to compress a data stream in a network, can be applied to the SSS to achieve the following advantages: (i) The shares are constructed using the XOR for fast computation; (ii) The parameter can be chosen arbitrarily; (iii) The direct share repair is supported; and (iv) The size of a share is optimized compared with previous schemes.

Acknowledgements

There are not many chances in our lives when we have the opportunity to acknowledge the people who really help us to achieve the success and who always encourage us in good and bad situations. I find myself very lucky to have the chance to express my thanks and appreciation to all those kind people in this PhD thesis.

First of all, I would like to thank my advisor Associate Professor Kazumasa Omote of Japan Advanced Institute of Science and Technology (JAIST). He is the person who has made this work possible by leading it in a feasible direction. Standing behind his valuable advice, I always receive precious comments and consistent encouragement which guided me since the early stages of my study and through my most difficult time in research.

I would like to thank Professor Atsuko Miyaji of JAIST for broadening my horizons on this work. Her dedication in teaching and research has always been a rich source of inspiration. I am really grateful to Associate Professor Yuto Lim of JAIST for his supervision of my minor research. Being confronted with his challenging questions has furthered my maturity in scientific life. I express my gratefulness to Associate Professor Ogata and Professor Ryuhei Uehara of JAIST, and Doctor Kiyomoto Shinsaku of KDDI R & D Laboratories Inc. for their supportive discussions and suggestions.

Last but not least, I am much grateful to my beloved family for keeping their faith in me throughout my time at doctoral course. I devote my sincere thanks and appreciation to all of them.

Contents

Acknowledgements	1
Table of Contents	4
List of Figures	5
List of Tables	6
List of Abbreviations	7
1 Introduction	8
1.1 Challenge of Cloud Computing	8
1.2 Research Goal	9
1.2.1 Proof Of Retrievability (POR)	9
1.2.2 Secret Sharing Scheme (SSS)	10
1.3 Contributions	11
1.3.1 MD-POR: Multi-client and Direct Repair for POR	11
1.3.2 DD-POR: Dynamic Operations and Direct Repair for POR	12
1.3.3 ND-POR: Network Coding and Dispersal Coding for POR	13
1.3.4 SW-SSS: Slepian-Wolf coding-based SSS	15
1.4 Thesis Outline	16
2 Related Work	17
2.1 POR	17
2.1.1 State Of The Art	17
2.1.2 Problem Statement	30
2.2 SSS	32
2.2.1 State Of The Art	32
2.2.2 Problem Statement	36
3 Preliminary	38
3.1 POR	38
3.2 Network Coding	38
3.2.1 Fundamental Concept	39
3.2.2 Application in Distributed Storage System	39

3.3	Homomorphic MAC	40
3.3.1	Inner-product MAC	41
3.3.2	Inter MAC	41
3.3.3	Inter MAC in Network Coding	42
3.4	Dispersal Coding	44
3.4.1	Building Block	45
3.5	Shamir SSS	47
3.6	Ramp SSS	49
3.7	SWC	51
4	MD-POR: Multi-client and Direct Repair for POR	54
4.1	System Model	54
4.2	Adversarial Model	55
4.3	Proposed MD-POR Scheme	57
4.3.1	Keygen	60
4.3.2	Encode	62
4.3.3	Check	63
4.3.4	Repair	64
4.4	Correctness	65
4.5	Security Analysis	66
4.5.1	Mobile Attack	66
4.5.2	Curious Attack	67
4.5.3	Response Forgery	68
4.5.4	Pollution Attack	69
4.6	Efficiency Analysis	71
4.6.1	Storage Cost	71
4.6.2	Encode cost	73
4.6.3	Check Cost	75
4.6.4	Repair Cost	76
4.6.5	Total cost	77
4.7	Performance Evaluation	79
4.7.1	Computation Performance	79
	CASE 1: fix number of blocks, change block size	79
	CASE 2: fix block size, change number of blocks	83
	CASE 1 vs CASE 2	87
4.7.2	Communication Performance	89
4.8	Numeric Example of Keygen Phase	90
4.8.1	The key of the client \mathcal{C}_1	91
4.8.2	The key of the client \mathcal{C}_2	92
4.8.3	The key of the TPA	93
4.8.4	The key of the new server	93
4.9	Summary	95

5	DD-POR: Dynamic Operations and Direct Repair for POR	96
5.1	System Model	96
5.2	Adversarial Model	96
5.3	Proposed DD-POR Scheme	97
5.3.1	Keygen	98
5.3.2	Encode	98
5.3.3	Check	99
5.3.4	Repair	100
5.4	Correctness	101
5.5	Dynamic Operations	102
5.5.1	Modification	102
5.5.2	Insertion	105
5.5.3	Deletion	111
5.6	Security Analysis	115
5.6.1	Pollution Attack	115
5.6.2	Curious Attack	117
5.6.3	File reconstruction condition	117
5.7	Efficiency Analysis	118
5.7.1	Encode Computation	118
5.7.2	Check Computation	118
5.7.3	Repair Computation	118
5.7.4	Modification Computation	120
5.7.5	Insertion Computation	120
5.7.6	Deletion Computation	120
5.8	Numeric Example	120
5.8.1	Generating Keys	121
5.8.2	Dynamic Operations	122
	Modification	122
	Insertion	124
	Deletion	125
5.9	Summary	127
6	ND-POR: Network Coding and Dispersal Coding for POR	128
6.1	System Model	128
6.2	Adversarial Model	128
6.3	Proposed ND-POR Scheme	130
6.3.1	Keygen	131
6.3.2	Encode	132
6.3.3	Check	133
6.3.4	Repair	134
6.4	Security Analysis	135
6.4.1	Adversarial Check and Repair	135
6.4.2	Small Corruption Attack	137
6.4.3	Large Corruption Attack, Replay Attack and Pollution Attack . . .	138

6.5	Efficiency Analysis	139
6.5.1	Encode Phase	139
6.5.2	Check Phase	139
6.5.3	Repair Phase	141
6.5.4	Storage Cost	141
6.5.5	Numerical Examples of The Parameters	142
6.6	Summary	143
7	SW-SSS: Slepian-Wolf Coding-based SSS	144
7.1	System Model	144
7.2	Revisited XOR Network Coding-based SSS	145
7.3	Proposed SW-SSS	146
7.3.1	Share Generation	147
7.3.2	Secret Reconstruction	148
7.3.3	Share Repair	150
7.4	Secrecy and Share Size	152
7.4.1	Secrecy	152
7.4.2	Share Size	153
7.5	Efficiency Analysis	154
7.5.1	Storage Cost	154
7.5.2	Computation Cost	154
7.5.3	Communication Cost	157
7.6	Implementation	158
7.6.1	Speeding up the FindShare algorithm	158
7.6.2	Speeding up the FindXOR algorithm	158
7.6.3	Performance Evaluation	160
7.7	Summary	161
8	Conclusion and Future works	162
8.1	Conclusion	162
8.2	Future work	164
	Bibliography	165
	Publications	178
A	Appendix	180
A.1	The Algorithms of the SW-SSS Scheme	180
A.1.1	Share Generation	180
A.1.2	Secret Reconstruction	181
A.1.3	Share Repair	182
A.1.4	Speeded Up Algorithms	183

List of Figures

1.1	My research map	11
1.2	Thesis Outline	16
3.1	An example of data repair of network coding	40
3.2	SWC	52
4.1	System model of the MD-POR scheme	55
4.2	Technical roadmap	58
4.3	Computation time performance of init and keygen phases	81
4.4	Computation time performance of encode phase	81
4.5	Computation time performance of check phase	82
4.6	Computation time performance of repair phase	82
4.7	Computation time performance of init and keygen phases	85
4.8	Computation time performance of encode phase	85
4.9	Computation time performance of check phase	86
4.10	Computation time performance of repair phase	86
4.11	Computation time performance of init and keygen phases	88
4.12	Computation time performance of encode phase	88
4.13	Computation time performance of check phase	89
4.14	Computation time performance of repair phase	89
4.15	Communication time performance	90
6.1	The structure of the ND-POR scheme	131
7.1	System Model of the SW-SSS	144
7.2	Computation performance of the SW-SSS scheme	160

List of Tables

2.1	Notations used in the RDC-NC scheme	20
2.2	Notations used in the NC-Audit scheme	26
2.3	Previous PORs vs our POR	32
2.4	Previous SSSs vs our SSS	36
4.1	List of notations in the MD-POR scheme	59
4.2	Efficiency comparison between the MD-POR and previous schemes	72
4.3	Summary of computation results in case 1 (time unit: second)	80
4.4	Summary of computation results in case 2 (time unit: second)	84
5.1	List of notations in the DD-POR scheme.	97
5.2	Efficiency comparison between the DD-POR and previous schemes	119
6.1	List of notations in the ND-POR scheme	130
6.2	The comparison between the RDC-NC and ND-POR schemes	140
7.1	List of notations in the revisited SSS and the SW-SSS	145
7.2	Efficiency comparison between the SW-SSS and previous schemes	155

List of Abbreviations

CPA	Chosen Plaintext Attack
DD-POR	Dynamic operation and Direct repair for POR
ECC	Error-Correcting Code
FMSR	Functional Minimum-Storage Regenerating
LDPC	Low Density Parity Check
LT	Luby Transform
MAC	Message Authentication Code
MD-POR	Multi-client and Direct repair for POR
NC-Audit	Audit for Network Coding storage
ND-POR	Network Coding and Dispersal Coding for POR
ORAM	Obvious RAM
PDP	Provable Data Possession
POR	Proof Of Retrievability
PRF	Pseudo-random Function
RDC-NC	Remote Data Checking for Network Coding-based distributed storage system
RDC-EC	Remote Data Checking for Erasure Coding-based distributed storage with server-side repair
RDC-SR	Remote Data Checking for replication-based distributed storage with Server-side Repair
RS	Reed Solomon code
RS-UHF	Universal Hash Function which is constructed using the Reed-Solomon code
SSS	Secret Sharing Scheme
SW-SSS	Slepian-Wolf coding-based SSS
SWC	Slepian-Wolf Coding
TPA	Third Party Auditor
UHF	Universal Hash Function
UMAC	MAC based on Universal Hash Function
XOR	Exclusive-OR

Chapter 1

Introduction

1.1 Challenge of Cloud Computing

Since amount of data is increasing exponentially, data storage and data management become troublesome tasks of the data owners. To reduce the burdens for the data owners, the concept of remote storage known as *cloud* has been proposed. A cloud is considered as a service through which the clients (the data owners) can use to publish, access, manage and share their data remotely and easily from anywhere via the Internet. Several examples of clouds which are commonly used are Amazon S3 [1], Storage Request Broker [2], Google's BigTable [3], HP Public Cloud [4]; and the mostly recent clouds are Dropbox [5], Google Drive [6] and iCloud [7].

Although data outsourcing to clouds can reduce the storage and management burdens for the client, this method still encounters a problem that the service provider may be typically not fully trusted. Therefore, this method introduces numerous interesting research challenges in data security: (i) data availability, (ii) data integrity and (iii) data confidentiality.

- *Data availability*: For any information system to serve its purpose, the data must be always ready when it is needed. This means that the computing systems used to store and process the information, the security controls used to protect it, and the communication channels used to access it must be functioning correctly. High availability systems aim to remain available at all times, preventing service disruptions due to power outages, hardware failures, system upgrades and denial-of-service (DOS) attacks such as a flood of incoming messages to the target system essentially forcing it to shut down.
- *Data integrity*: Integrity involves maintaining the consistency, accuracy, and trustworthiness of data over its entire life cycle. Data must not be changed in transit, and steps must be taken to ensure that data cannot be altered by unauthorized or undetected manner. Integrity is violated when the data is actively modified.
- *Data confidentiality*: The data needs to be prevented from the disclosure to unauthorized individuals or systems. The system attempts to enforce confidentiality

by encrypting the data or by restricting access. Data confidentiality consists of the two research approaches: the cryptographic approach and the information-theoretic approach. Compared with the cryptographic confidentiality approach, the information-theoretic confidentiality approach achieves a security level determined by thresholds.

In this study, we focus on data availability, data integrity and data information-theoretic confidentiality. We choose the information-theoretic confidentiality approach because our security analysis derives purely from information theory.

1.2 Research Goal

Our general goal is to construct a cloud system which is practical, efficient and secure. To obtain the goal, our research consists of two directions: Proof Of Retrievability (POR) and Secret Sharing Scheme (SSS).

1.2.1 Proof Of Retrievability (POR)

The POR is important because it has been proposed to help the client check whether his/her data stored in the cloud servers ('the servers' for short) is always available, intact and retrievable. Based on the POR protocol, the following four techniques can be used: (i) replication, (ii) erasure coding, (iii) obvious RAM (ORAM) and (iv) network coding. In this study, we focus on the network coding because it can achieve better storage cost compared with the replication, and better computation and communication costs compared with the erasure coding and the ORAM. Although many network coding-based PORs have been proposed, the efficiency and practicality have not been addressed simultaneously (We will describe more detail about previous work in Chapter 2). Therefore, we would like to construct a new network coding-based POR which satisfies the following two aims:

- The first aim is that our proposed network coding-based POR should be practical. Concretely:
 - The system model should consist of multiple clients, not just a single client like previous network coding-based PORs. Each client should keep a different secret key. This is because in many distributed storage systems today such as Dropbox, each client has a personal data and should compute the authentication information for that data using his/her own secret key in order to ensure that data integrity and data confidentiality are satisfied.
 - The clients should be able to check and retrieve the data, and also be able to perform dynamic operations on their data such as modification, deletion and insertion. This is because in a real cloud system, the dynamic operations happen very often during the system lifetime.

- The second aim is that our proposed network coding-based POR should be lightweight. Concretely:
 - The clients should be free of two heaviest tasks: (i) periodically checking the servers to ensure that the data stored in the servers is available, intact and retrievability and (ii) repairing the data stored in corrupted servers.
 - The system should be constructed using a symmetric key setting which is a well-known lightweight cryptography rather than an asymmetric key setting.

1.2.2 Secret Sharing Scheme (SSS)

SSS is important because it is a method for protecting distributed file systems against data leakage and data loss. In this scheme, the secret is encoded into a number of shares. The shares are then distributed to a group of participants where each participant holds a share of the secret. The secret can be only reconstructed if and only if a sufficient number of valid shares are reconstituted. A general SSS consists of two algorithms: (i) share generation and (ii) secret reconstruction. Although many SSSs have been introduced, they have not achieved an optimal share size and cannot support the share repair feature (We will describe more detail about previous work in Chapter 2). Therefore, we would like to construct a new SSS which satisfies the following two aims:

- The first aim is that our proposed SSS should be lightweight. Concretely:
 - The computation costs of the share generation and secret reconstruction algorithms should be reduced as much as possible because in a real system, the size of the secret is very large; thus, it will affect the computation costs of the share generation and the secret reconstruction algorithms.
 - The bit-size of the shares should be optimized. If the bit-size of the shares is optimized, the required storage cost for the system will be also optimized.
- The second aim is that our proposed SSS should be practical. Concretely:
 - The parameters can be chosen arbitrarily, not strictly constrained as previous schemes.
 - The direct share repair feature should be supported because in a real scenario, a share which is held by a participant could be corrupted or lost. This share corruption or share loss will reduce the entropy of the system and will make the secret reconstruction impossible.

1.3 Contributions

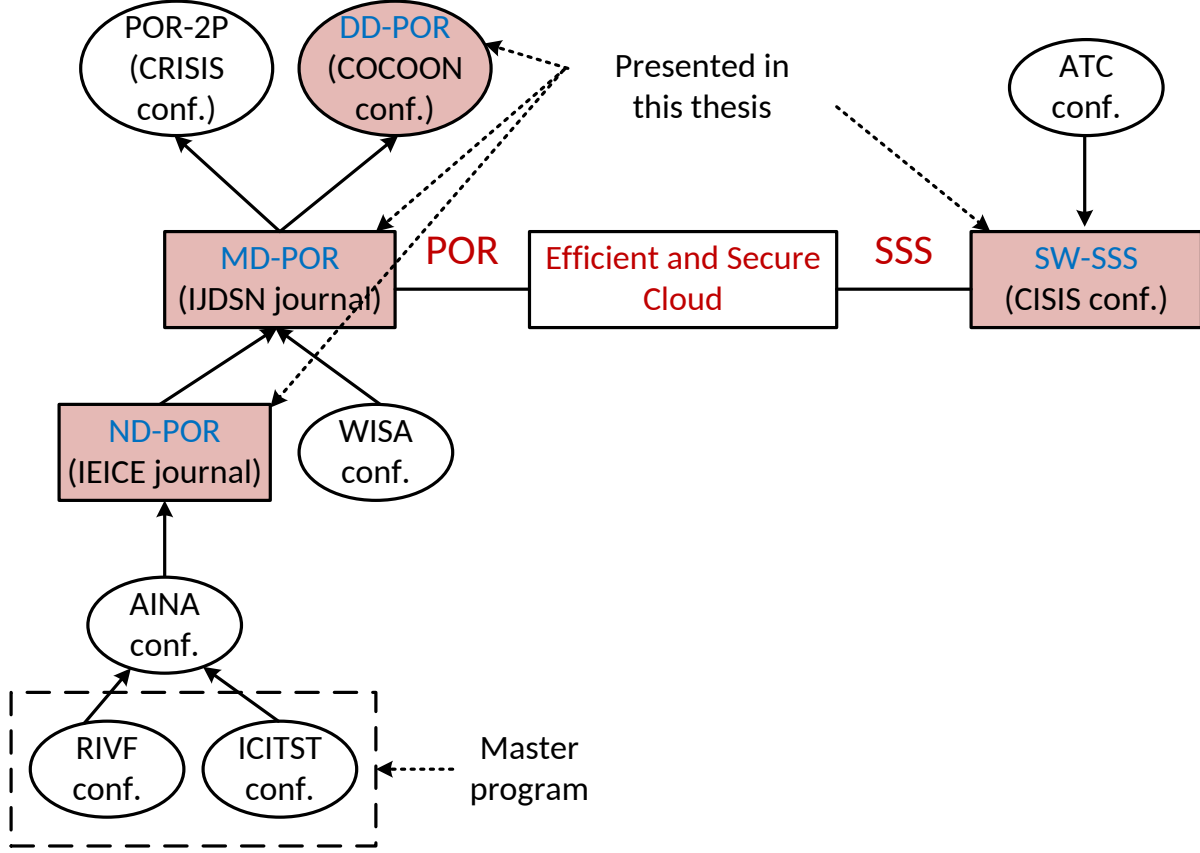


Figure 1.1: My research map

As depicted in Figure 1.1, we firstly present an overview about my research map. Up to now, we have some papers and journals. An arrows in Figure 1.1 is used to connect a prior work to a later work. Our main contributions are the following schemes:

- The MD-POR scheme, which is the main proposed POR.
- The SW-SSS scheme, which is the main proposed SSS.
- The DD-POR scheme, which is the partial improvement of the MD-POR scheme.
- The ND-POR scheme, which is our first proposed POR.

In this thesis, we will introduce these four schemes.

1.3.1 MD-POR: Multi-client and Direct Repair for POR

This MD-POR scheme is our main proposed network-coding POR. To the best of our knowledge, we are the first to propose a symmetric key setting-based direct repair for

the POR. Furthermore, our proposed scheme can also support multi-client and public authentication. Namely, the MD-POR has the following contributions:

- *Direct repair*: When a corrupted server is detected during the check phase, a number of healthy servers are required to provide their coded blocks along with the tags directly to the new server, instead of sending them back to the client. Afterwards, the new server can verify the coded blocks and the tags it received, and computes new coded blocks and new tags for itself without disturbing the client. This mechanism can reduce a lot of the communication cost and the burden for the client.
- *Multi-client*: To enable multiple clients, our method does not simply duplicate the process of a single client to multiple parallel processes for multiple clients. Instead, in our proposed scheme, the data of multiple clients are mixed together without losing the data confidentiality of individual client. To enable such a multi-client setting, we employ the inter MAC technique [79] which was proposed for network scenario. The inter MAC technique allows multiple sources to send their packages to the network using different secret keys and allows the recipients to verify the packages they received.
- *Symmetric key setting*: The MD-POR scheme is constructed based on the symmetric key setting. We use only secret keys without any public key, unlike an asymmetric key setting.
- *Public authentication*: Not only the client but also any entity who is given our additional information can check the servers while learning nothing about the secret keys of the clients. However, there should be a consistent entity who has responsibility to check the servers periodically. Therefore, we employ a Third Party Auditor (TPA) to check the servers periodically on behalf of the clients. By delegating the responsibility of checking the servers to the TPA, the clients are free of the burden of checking the servers. Otherwise, for the non-existence of TPA, the clients have to periodically check the servers, and the public authentication feature cannot be supported. The interesting point here is that although the proposed MD-POR scheme supports the public authentication feature, our method does not use an asymmetric key setting.

1.3.2 DD-POR: Dynamic Operations and Direct Repair for POR

The DD-POR scheme is a partial improvement of the MD-POR scheme. In this scheme, we point out that the previous schemes do not consider the dynamic operations. That is, the client can only perform the data check and data retrieval, but cannot perform the modification, insertion and deletion. Several PORs have been proposed to deal with the dynamic operations, e.g., [34, 36, 67, 69–73]. However, all these schemes are based on the erasure coding, not the network coding. Therefore, our aim on this DD-POR scheme is that we want to construct a new network coding-based POR which can support both the direct repair feature and dynamic operations.

There are two most notable schemes which are mostly related to our aim. The first one is our proposed MD-POR scheme, which can support the direct repair, but cannot support the dynamic operations. The second one is the NC-Audit scheme [62], which also considered the direct repair and dynamic operations. However, when the direct repair is supported, this scheme cannot prevent the pollution attack which is a common attack of the network coding. This is because the new server cannot check the provided coded blocks it receives during the repair phase. In addition, the authors only discuss about the dynamic operations without clear details. For example, for the modification, the authors discuss how to update the tag without mentioning how to update the coded blocks which are related to the modified file block. For the deletion, there is no concrete explanation. Moreover, the dynamic operations in the scheme have not been completed because for the insertion, the authors mentioned that the insertion does not work in their scheme.

For this motivation, we propose the DD-POR scheme with the following contributions:

- Direct repair: when a server is corrupted, the healthy servers will provide their coded blocks and tags directly to the new server without sending them back to the client. Then, the new server can check them to prevent the pollution attack, and can compute the new coded blocks and the tags for itself. The client is thus free from the repair process.
- Dynamic operations: the client not only can check and retrieve the data, but also can modify, insert and delete the data.
- Symmetric key setting: our scheme does not use any public key as in an asymmetric key setting. The direct repair feature introduces a challenge that how to let the new server which is untrusted check and compute the new coded blocks and the tags without using a public key. Our scheme can address this problem by employing the inter MAC technique [79].

Note that this proposal is a partial improvement of our first proposed MD-POR scheme. This is because in this DD-POR scheme, we can only deal with a single client instead of multiple clients as in the MD-POR scheme. Furthermore, the DD-POR does not deal with the public authentication as in the MD-POR scheme.

1.3.3 ND-POR: Network Coding and Dispersal Coding for POR

The ND-POR is one of our very first proposed POR scheme in which we started studying about network coding. The purpose when we propose this scheme is to construct a POR which can beat the RDC-NC scheme [61] in both security (i.e., small corruption attack) and efficiency.

In these network coding-based POR schemes, the most notable scheme is the RDC-NC scheme [61]. It, unlike the other previous schemes, not only focuses on the efficiency, but also considers how to prevent the three common attacks of the POR: *replay attack*, *pollution attack* and *large corruption attack*. However, the RDC-NC scheme has some shortcomings: (i) the corruption check is still inefficient because only one server can be

checked per challenge and (ii) it cannot prevent another common attack of the POR: *small corruption attack*. The small corruption attack is defined in [26, 67, 105, 106]. In this attack, the adversary tries to corrupt the data with a small data unit to hide data loss incidents. Protecting against the small corruption attack protects the data itself, not just the storage resource. Modifying a single bit may destroy an encrypted file or invalidate authentication information. The difference between the large and small corruption attacks is that the small corruption attack corrupts at most t -fraction of the file while the large corruption attack corrupts more than t -fraction of the file, where t is a parameter. These are described more details in the adversarial model of the ND-POR scheme.

To address the small corruption attack, the common solution is to use the Error-Correcting Code (ECC) [94], which allows the data to be checked for errors and corrected even one bit on the fly. The ECC has several types, i.e., Hamming code, Golay code, Reed-Muller code, Reed-Solomon code, etc. However, our scheme uses the Reed-Solomon code because the Universal Hash Function can be constructed using the Reed-Solomon code. Bowers et al. [26] then proposed the *dispersal coding* using the Reed-Solomon code in order to prevent the small corruption attack and to ensure the file integrity with high probability. However, [26] uses the erasure coding instead of the network coding.

The ND-POR scheme has been proposed using the network coding and the dispersal coding. To the best of our knowledge, the ND-POR scheme is the first POR to apply both the dispersal coding and the network coding. The ND-POR scheme has the following contributions:

- *Security*: The ND-POR scheme, unlike the RDC-NC scheme, can prevent the small corruption attack.
- *Efficiency*:
 - The RDC-NC scheme allows the client to check one server for each challenge. Meanwhile, the ND-POR scheme allows the client to check all servers simultaneously for each challenge.
 - In the RDC-NC scheme, the number of MACs is $n\alpha s$ where n denotes the number of servers, α denotes the number of coded blocks stored on a server and s denotes the number of segments in a coded block. In the ND-POR scheme, the number of MACs is only $l\alpha$ where l denotes some servers out of n servers ($l < n$) and is far less than the dominant parameter s .
 - In data repair, the RDC-NC scheme uses the network coding to repair the corruptions. Meanwhile, the ND-POR scheme performs two phases: if the number of corruptions is smaller than the ECC boundary, the ECC is used to repair the corruptions; otherwise the network coding is used to repair the corruptions. Thus, the corruptions are repaired with an overwhelming probability. Furthermore, the ECC uses the parity information on the server itself to repair without the other healthy servers as the network coding.

The dispersal coding is constructed based on UMAC (MAC obtained from Universal Hash Function) which is closely related to the network coding-based schemes as indicated

in [52, 104]. Hence, the network coding and the dispersal coding can be suitably combined together in the ND-POR scheme.

1.3.4 SW-SSS: Slepian-Wolf coding-based SSS

Before presenting the main proposed SW-SSS scheme, we firstly revisit the network coding based on the XOR [135–139] and show that it can be applied to for SSS to address the drawbacks of the previous schemes. Concretely, the revisited XOR network coding-based SSS has the following four advantages:

- The shares are constructed using the XOR for fast computation.
- The parameters (m, n) can be chosen arbitrarily.
- The direct share repair is supported.
- The size of a share is smaller than the size of the secret.

We then show that another coding named the Slepian-Wolf Coding (SWC) [140, 142–145], which is commonly used to compress a data stream in a network, can be also applied for SSS to reduce the share size of the revisited XOR network coding-based SSS. We name our proposed scheme as the SW-SSS. The SW-SSS has the following advantages:

- The share size in the SW-SSS is surprisingly less than the share size in the revisited XOR network coding-based SSS. In other words, the SW-SSS scheme improves the fourth advantage of the revisited XOR network coding-based SSS.
- The SW-SSS still satisfies the first three advantages of the revisited XOR network coding-based SSS.

1.4 Thesis Outline

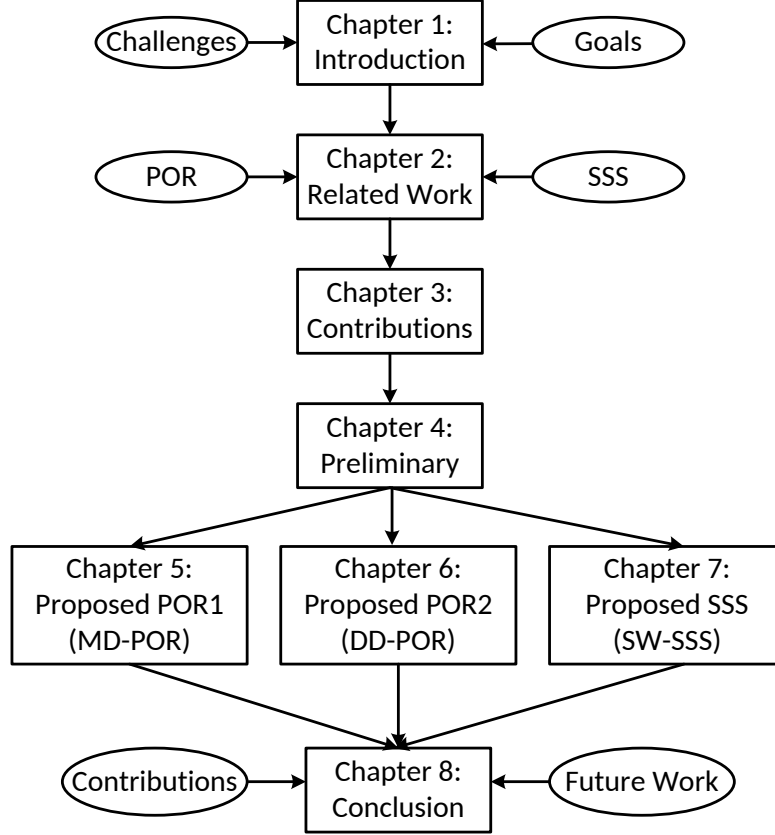


Figure 1.2: Thesis Outline

This thesis consists of 7 chapters as depicted in Figure 1.2. In Chapter 2, we discuss several previous works which are related to our two research directions: POR and SSS. In Chapter 3, we introduce several preliminaries which are used in our proposed schemes: POR, network coding, homomorphic MAC, Shamir SSS, Ramp SSS and SWC. In Chapter 4, we describe our proposed MD-POR scheme (Multi-client and Direct repair for POR) along with its security, efficiency and performance evaluation analyses. In Chapter 5, we describe our propose DD-POR scheme (Dynamic operation and Direct repair for POR) along with its security, efficiency and performance evaluation analyses. In Chapter 6, we describe our proposed ND-POR scheme (Network Coding and Dispersal Coding for POR) along with its security, efficiency performance evaluation analyses, and numeric example. In Chapter 7, we describe our proposed SW-SSS scheme (Slepian-Wolf coding-based SSS) along with its secrecy, efficiency and performance evaluation analyses. Finally, Chapter 8 will summarize this thesis, point out the contributions and suggest for the future research directions.

Chapter 2

Related Work

2.1 POR

2.1.1 State Of The Art

POR. To assist the client in checking whether the data stored in the servers is always available, intact and retrievable, researchers proposed Provable Data Possession (PDP) [102, 108, 109] and Proof of Retrievability (POR) [8–13] which are challenge-response protocols between a verifier (client) and a prover (cloud server). Both protocols support data check. However, only the POR can ensure that the data are always retrievable and can support data repair. Thus, the POR is considered to be a stronger tool. A POR consists of four phases: (i) keygen, (ii) encode, (iii) check and (iv) repair. Below we generally review the four phases (we will describe them formally in Section 3.1 in Chapter 3).

- **Keygen:** The client performs this algorithm to generate a pair of secret key and public key. In case of symmetric key setting, the public key is set to be null.
- **Encode:** The client uses his/her secret key to transform an original file to an encoded file, then stores the encoded file in the server.
- **Check:** This is the challenge-response protocol which happens as follows:
 - **Challenge:** The client generates a challenge and sends it to the server.
 - **Respond:** The server computes a corresponding response and sends it back to the client.
 - **Verify:** The client verifies whether the response is valid or not in order to conclude that the server is corrupted or not.
- **Repair:** If the server is detected as corrupted during the check phase, the client will perform this algorithm to repair the data stored in the corrupted server.

Approaches in POR. Based on the POR protocol, there are two research approaches:

- The first approach is that the data is stored in only a single server. The client can periodically check data possession at the server and can thus detect data corruption. However, the drawback of this approach is that when a corruption is detected, the data repair will not be supported.
- The second approach is that the data is stored redundantly in multiple servers. When a server is corrupted, the client will use the remaining healthy servers to repair the data stored in the corrupted server. This approach consists of the following four techniques: replication, erasure coding, ORAM, and network coding.
 - *Replication.* Replication is a technique which allows the client to store file replicas (file copies) in the servers. The replication was firstly proposed in [14–16] and has been applied to distributed storage systems in [17, 18]. The client can perform periodic server checks. When a corrupted server is detected, the client will use the replica stored in one of the healthy servers to repair the data stored in the corrupted server. The drawback of this technique, however, is that it incurs high storage cost because the client must store a whole file copy in each server.
 - *Erasure Coding.* Erasure coding was used traditionally in communication systems [19] and then has been applied in distributed storage systems [20–27] for optimal data redundancy. Instead of storing file replicas in the server as the replication, in this technique, the client stores file blocks (parts of the file) in each server. Thus, the erasure coding can reduce the storage cost of the replication. However, the drawback of this technique is that to repair a corrupted server, the client must reconstruct the original file before repairing the corruption. Therefore, the computation cost is increased during data repair.
 - *ORAM.* ORAM was initially introduced for protecting software [28–33]. Recently, the ORAM has been applied to distributed storage systems [34–37]. Basically, this technique is proposed for privacy-preserving the data access pattern. By using the ORAM structure, the servers cannot obtain the data access patterns when the client performs the data checks. For the data repair, the ORAM-based POR embeds the erasure coding to repair corruptions. However, the drawback of this technique is that the ORAM structure leads to high storage cost because of its hierarchical storage layout. Moreover, the ORAM structure leads to high computation cost because of its shuffling procedure every number of read/write operations.
 - *Network Coding.* Network coding was firstly proposed in the network scenario [38–50]. To address the drawback of the erasure coding, the network coding has been applied [51, 60–65] to distributed storage systems to improve the efficiency in data repair. The client does not need to reconstruct the entire file before generating new coded blocks as the erasure coding. Instead, the coded blocks which are collected from the healthy servers can be used to generate new

coded blocks. Compare with the ORAM, the structure of the network coding is much simpler with no hierarchical storage, no shuffling procedure and no the drawback of the erasure coding. Therefore, in this thesis, we focus on the network coding technique.

Message Authentication Code (MAC) vs. Digital Signature (signature). The data stored in the servers cannot be checked without additional authentication information. The authentication information can be (i) MAC or (ii) digital signature (or just signature for short).

- A MAC is also called a *tag*. A MAC protects against message forgery by anyone who does not know the secret key (which is shared by sender and receiver). A MAC is used only in a symmetric key setting. The traditional MAC and digital signature might not be suitable for network coding; thus, new technique called *homomorphic MAC* [52–54] has been proposed.
- A (digital) signature is created with a private key, and verified with the corresponding public key of an asymmetric key-pair. Only the holder of the private key can create this signature, and normally anyone knowing the public key can verify it. Therefore, the digital signature is used only in an asymmetric key setting. Similar to the MAC approach, the homomorphic signature [55–59] have been proposed to combine with the network coding.

In this thesis, we focus on a symmetric key setting for efficiency. We thus use homomorphic MAC approach in our proposed schemes.

Network Coding. Because the network coding technique is focused on in this thesis as we mentioned before, in this part, we introduce several previous works about the network coding. The network coding was originally proposed in the networks, and then has been applied to distributed storage systems.

- *Network coding in networks:* Ahlswede et al. [42] were the first to consider the problem multicast of an error-free network. In their work, which had its precursor in earlier work relating to specific network topologies [38–41], the authors showed that coding at intermediate nodes is in general necessary to achieve the capacity of a multicast connection in an error-free network and characterized that capacity. This result generated renewed interest in error-free networks, and it was quickly strengthened by Li et al. [43] and Koetter et al. [44], who independently showed that linear codes (i.e., codes where nodes are restricted to performing operations that are linear over some base finite field) suffice to achieve the capacity of a multicast connection in an error-free network. Ho et al. [45] then introduced the random linear network coding as a method for multicast in lossless packet networks and analysed its properties. The random linear network coding for multicast in lossless packet networks was further studied in [46–48]. Li et al. [49] proposed a tree structure data regeneration with the linear network coding to achieve an efficient regeneration

traffic and bandwidth capacity by using an undirected-weighted maximum spanning tree and the Prim algorithm. In their paper, the authors analysed the bottleneck bandwidth that the tree-structured regeneration can achieve, but did not analysed the constraint of the threshold which is the number of providers. Therefore, the authors then improved their paper in [50] to present an in-depth analysis of the general case that the number of providers.

- *Network coding in distributed storage systems:* Dimakis et al. [51] was the first to apply the network coding to distributed storage systems and achieve a remarkable reduction in the communication overhead of the repair component. Acedanski et al. [60] demonstrated that when the random linear coding is applied to distributed storage system, it performs as well without suffering additional storage space required at the centralized server before distribution among multiple locations. Further, with a probability close to one, the minimum number of storage location a downloader needs to connect to (for reconstructing the entire file), can be very close to the case where there is complete coordination between the storage locations and the downloader. Chen et al. [61] presented the RDC-NC scheme (Remote Data Checking for Network Coding-based distributed storage systems) which provides a decent solution for efficient data repair by recoding encoded blocks on the healthy servers during the repair procedure. Le et al. [62] introduced the NC-Audit scheme (Audit for Network Coding storage) for efficient data check and data repair. Furthermore, the NC-Audit scheme can also prevent data leakage to a Third Party Auditor (TPA) using a combination of a homomorphic MAC called SpaceMac and a Chosen Plaintext Attack (CPA)-secure encryption called NCrypt. Cao et al. [63] applied the Luby Transform (LT) code for reducing the computation cost because the LT code is a special network code which works in the finite field of order two and only uses the XOR operations. Chen et al. [64] proposed the NC-Cloud scheme to improve the cost-effectiveness of repair using the Functional Minimum-Storage Regenerating (FMSR) code, which lightens the encoding requirement of storage nodes during repair. The authors then extended their prior work to [65] with more in-depth analysis and evaluations on their implementable design of FMSR codes. Chen et al. [66] investigated the intrinsic relationship between secure cloud storage and secure network coding and proposed a publicly verifiable secure cloud storage protocol in the standard model.

Overview of RDC-NC scheme. In this part, we briefly describe the RDC-NC scheme [61] which is a notable previous network coding-based POR proposed by Chen et al. We will use this scheme to compare with our schemes in later chapters. The notations used throughout this scheme are given in Table 2.1.

Table 2.1: Notations used in the RDC-NC scheme

Notation	Description
\mathcal{C}	client

F	original file of \mathcal{C}
m	number of file blocks
n	number of servers
α	number of coded blocks stored in a server
s	number of segments in a coded block
u	number of symbols in a coded block
b_k	file block ($k \in \{1, \dots, m\}$)
\mathcal{S}_i	server ($i \in \{1, \dots, n\}$)
c_{ij}	coded block ($i \in \{1, \dots, n\}, j \in \{1, \dots, \alpha\}$)
f	pseudo-random function $f : \{0, 1\}^* \times \{0, 1\}^\kappa \rightarrow \mathbb{F}_q$
t_{ijk}	challenge tag of c_{ij} ($i \in \{1, \dots, n\}, j \in \{1, \dots, \alpha\}, k \in \{1, \dots, s\}$)
T_{ij}	repair tag of c_{ij} ($i \in \{1, \dots, n\}, j \in \{1, \dots, \alpha\}$)
\mathbb{F}_q	finite field of a prime order q
z_{ij1}, \dots, z_{ijm}	coding coefficients
$\epsilon_{ij1}, \dots, \epsilon_{ijm}$	encrypted coefficients
r	number of spot checks in the check phase
\mathcal{S}_y	corrupted server
\mathcal{S}'	new server which is used to replace \mathcal{S}_y

We now describe the RDC-NC scheme via each phase of the POR as follows:

Keygen:

1. \mathcal{C} divides F into m blocks: $F = b_1 || \dots b_m$.
2. \mathcal{C} generates the secret key $sk = (K_1, K_2, K_3, K_4, K_{enc})$, where each of these five keys is chosen at random from $\{0, 1\}^\kappa$.

Encode: For each server $\forall i \in \{1, \dots, n\}$:

1. \mathcal{C} computes values for generating challenge tags and repair tags

- \mathcal{C} generates a value δ which will be used for generating the challenge tags:

$$\delta = f_{K_1}(i) \quad (2.1)$$

- \mathcal{C} generates u values $\lambda_1, \dots, \lambda_u$ which will be used for generating the repair tag:
 $\forall k \in \{1, \dots, u\}$:

$$\lambda_k = f_{K_2}(i || k) \quad (2.2)$$

2. \mathcal{C} generates coded blocks and metadata to be stored at server \mathcal{S}_i :

$$\forall j \in \{1, \dots, \alpha\}:$$

- \mathcal{C} randomly generates coefficients $z_k \xleftarrow{rand} \mathbb{F}_q$ for $\forall k \in \{1, \dots, m\}$.
- \mathcal{C} computes coded block:

$$c_{ij} = \sum_{k=1}^m z_k b_k \quad (2.3)$$

Note that the symbols in the vector c_{ij} are elements in \mathbb{F}_q .

- \mathcal{C} views coded block c_{ij} as an ordered collection of s segments $c_{ij} = (c_{ij1}, \dots, c_{ijs})$ where each segment contains one symbol from \mathbb{F}_q , and computes a *challenge tag* for each segment:
 $\forall k \in \{1, \dots, s\}$:

$$t_{ijk} = f_{K_3}(i||j||k||z_1||\dots||z_m) + \delta c_{ijk} \mod q. \quad (2.4)$$

- \mathcal{C} views coded block c_{ij} as a column vector of u symbols $c_{ij} = (c_{ij1}, \dots, c_{iju})$ where each symbol $c_{ijk} \in \mathbb{F}_q$, and computes a *repair tag* for block c_{ij} :

$$T_{ij} = f_{K_4}(i||j||z_1||\dots||z_m) + \sum_{k=1}^u \lambda_k c_{ijk} \mod q. \quad (2.5)$$

- \mathcal{C} then encrypts the coefficients:
 $\forall k \in \{1, \dots, m\}$:

$$\epsilon_{ijk} = \text{Enc}_{K_{enc}}(z_{ijk}) \quad (2.6)$$

3. \mathcal{C} sends the following data to the server S_i for storage:

$\forall j \in \{1, \dots, \alpha\}$:

- c_{ij} : coded block.
- $\epsilon_{ij1}, \dots, \epsilon_{ijm}$: encrypted coefficients.
- t_{ij1}, \dots, t_{ijs} : challenge tags.
- T_{ij} : repair tag.

\mathcal{C} can now delete the file F and stores only the secret key sk .

Check: For each of n servers, \mathcal{C} checks possession of each of α coded blocks at each server by using spot-checking of segments for each coded block. In this process, each server uses its stored blocks and the corresponding challenge tags to prove data possession.

For each server \mathcal{S}_i ($\forall i \in \{1, \dots, n\}$):

1. \mathcal{C} generates a set of queries to send to each server:

- \mathcal{C} generates r pairs (k, v_k) (correspond to the segments that are being checked) where:
 – $k \xleftarrow{rand} \{1, \dots, s\}$ (k is the index of the segment).

- $v_k \xleftarrow{rand} \mathbb{F}_q$ (v_k is the corresponding query coefficient).

Let the query Q be the r -element set $\{(k, v_k)\}$. \mathcal{C} sends Q to each server.

2. \mathcal{S}_i computes a proof of possession for coded block:

$\forall j \in \{1, \dots, \alpha\}$:

- \mathcal{S}_i computes the proof:

$$t_{ij} = \sum_{(k, v_k) \in Q} v_k t_{ijk} \mod q \quad (2.7)$$

$$\rho_{ij} = \sum_{(k, v_k) \in Q} v_k c_{ijk} \mod q \quad (2.8)$$

- \mathcal{S}_i sends to \mathcal{C} :
 - The proof of possession (t_{ij}, ρ_{ij}) .
 - The encrypted coefficients $(\epsilon_{i11}, \dots, \epsilon_{i1m}, \epsilon_{i21}, \dots, \epsilon_{i2m}, \dots, \epsilon_{i\alpha 1}, \dots, \epsilon_{i\alpha m})$.

3. \mathcal{C} checks the validity of the proof of possession (t_{ij}, ρ_{ij}) :

For $\forall j \in \{1, \dots, \alpha\}$:

- \mathcal{C} decrypts the encrypted coefficients:

$\forall k \in \{1, \dots, m\}$:

$$z_{ijk} = Dec_{K_{enc}}(\epsilon_{ijk}) \quad (2.9)$$

- \mathcal{C} regenerates $\delta = f_{K_1}(i)$
- \mathcal{C} checks if:

$$t_{ij} = \sum_{(k, v_k) \in Q} v_k f_{K_3}(i || j || k || z_{ij1} || \dots || z_{ijm}) + \delta \rho_{ij} \mod p \quad (2.10)$$

If the equality does not hold, \mathcal{C} declares \mathcal{S}_i faulty.

The correctness of Equation 2.10 is proved as follows:

Proof.

$$\begin{aligned} t_{ij} &= \sum_{(k, v_k) \in Q} v_k t_{ijk} \mod q \text{ //because of Equation 2.7} \\ &= \sum_{(k, v_k) \in Q} v_k f_{K_3}(i || j || k || z_{ij1} || \dots || z_{ijm}) + \delta \rho_{ij} \mod p \\ &= \text{//because of Equation 2.4} \end{aligned}$$

Therefore, Equation 2.10 holds. \square

Repair: Assume that in the check phase, \mathcal{C} has identified S_y as a corrupted server whose coded blocks are $S_{y1}, \dots, S_{y\alpha}$. \mathcal{C} will contact l healthy servers S_{i_1}, \dots, S_{i_l} and asks each of them to generate a new coded block. \mathcal{C} further combines these l coded blocks to generate α new coded blocks and metadata, and then stores them on a new server S' .

1. \mathcal{C} contacts l healthy servers S_{i_1}, \dots, S_{i_l} to ask them to generate new coded blocks.

For $\forall i \in \{i_1, \dots, i_l\}$:

- \mathcal{C} generates a set of coefficients (x_1, \dots, x_α) where $x_k \xleftarrow{\text{rand}} \mathbb{F}_q$ with $k \in \{1, \dots, \alpha\}$.
- \mathcal{C} asks server S_i to provide a new coded block and the proof of correct encoding using the coefficients (x_1, \dots, x_α) .
- Server S_i executes as follows:
 - S_i computes $\bar{a}_i = \sum_{j=1}^{\alpha} x_j c_{ij}$ (here the symbols a_{ir} of block \bar{a}_i are computed as $a_{ir} = \sum_{j=1}^{\alpha} x_j c_{ijr} \mod q$ for $r \in \{1, \dots, u\}$).
 - S_i computes a proof of correct encoding:

$$\tau_i = \sum_{j=1}^{\alpha} x_j T_{ij} \mod q \quad (2.11)$$

- S_i sends to \mathcal{C} :

$$\begin{aligned} & * \bar{a}_i \\ & * \tau_i \\ & * \{\epsilon_{i11}, \dots, \epsilon_{i1m}, \epsilon_{i21}, \dots, \epsilon_{i2m}, \dots, \epsilon_{i\alpha 1}, \dots, \epsilon_{i\alpha m}\} \end{aligned}$$

- \mathcal{C} decrypts the encrypted coefficients ϵ to recover coefficients $z_{i11}, \dots, z_{i1m}, z_{i21}, \dots, z_{i2m}, \dots, z_{i\alpha 1}, \dots, z_{i\alpha m}$.
- \mathcal{C} regenerates u values $\lambda_1, \dots, \lambda_u \in \mathbb{F}_q$:
 $\forall k \in \{1, \dots, u\}$:

$$\lambda_k = f_{K_2}(i||k) \quad (2.12)$$

- \mathcal{C} checks if:

$$\tau_i = \sum_{j=1}^{\alpha} x_j f_{K_4}(i||j||z_{ij1}||\dots||z_{ijm}) + \sum_{k=1}^u \lambda_k a_{ik} \mod q \quad (2.13)$$

where a_{i1}, \dots, a_{iu} are symbols of block \bar{a}_i . If the equality does not hold, then \mathcal{C} declares S_i faulty.

The correctness of Equation 2.13 is proved as follows:

Proof.

$$\begin{aligned} \tau_i &= \sum_{j=1}^{\alpha} x_j T_{ij} \mod q \text{ //because of Equation 2.11} \\ &= \sum_{j=1}^{\alpha} x_j f_{K_4}(i||j||z_{ij1}||\dots||z_{ijm}) + \sum_{k=1}^u \lambda_k a_{ik} \mod q \\ &= \text{//because of Equation 2.5 and replacing } c_{ijk} \text{ by } a_{ik} \end{aligned}$$

Therefore, Equation 2.13 is corrected. \square

2. \mathcal{C} combines these l coded blocks to generate α new coded blocks and metadata.

- \mathcal{C} generates a value δ which will be used for generating the challenge tags:

$$\delta = f_{K_1}(y) \quad (2.14)$$

- Generate u values $\lambda_1, \dots, \lambda_u$ which will be used for generating the repair tag:
 $\forall k \in \{1, \dots, u\}$:

$$\lambda_k = f_{K_2}(y||k) \quad (2.15)$$

- For $\forall j \in 1, \dots, \alpha$:

- \mathcal{C} randomly generates coefficients $z_k \xleftarrow{rand} \mathbb{F}_q$ where $\forall k \in \{1, \dots, l\}$.
- \mathcal{C} computes coded block:

$$c_{yj} = \sum_{k=1}^l z_k \overline{a_k} \quad (2.16)$$

The symbols in the vector c_{yj} are elements in \mathbb{F}_q .

- \mathcal{C} views c_{yj} as an ordered collection of s segments $c_{yj} = (c_{yj1}, \dots, c_{yjs})$ where each segment contains one symbol from \mathbb{F}_q , and computes a *challenge tag* for each segment:
 $\forall k \in \{1, \dots, s\}$:

$$t_{yjk} = f_{K_3}(y||j||k||z_{i_1}||\dots||z_{i_l}) + \delta c_{yjk} \mod q \quad (2.17)$$

- \mathcal{C} views c_{yj} as a column vector of u symbols $c_{yj} = (c_{yj1}, \dots, c_{yju})$ with $c_{yjk} \in \mathbb{F}_q$, and computes a *repair tag* for the block c_{yj} :

$$T_{yj} = f_{K_4}(y||j||z_{i_1}||\dots||z_{i_l}) + \sum_{k=1}^u \lambda_k c_{yjk} \mod q \quad (2.18)$$

- \mathcal{C} encrypts coefficients:
 $\forall k \in \{1, \dots, m\}$:

$$\epsilon_{yjk} = Enc_{K_{enc}}(z_{yjk}) \quad (2.19)$$

3. \mathcal{C} sends the new coded blocks to the new server S' :

For $\forall j = \{1, \dots, \alpha\}$: \mathcal{C} sends to S' :

- c_{yj} : new coded block
- $\epsilon_{yj1}, \dots, \epsilon_{yjm}$: encrypted coefficients
- t_{yj1}, \dots, t_{yjs} : challenge tags
- T_{yj} : repair tag

Overview of NC-Audit scheme. In this part, we briefly describe the NC-Audit scheme [62] which is another notable previous network coding-based POR proposed by Le et al. We will use this scheme to compare with our schemes in later chapters. The notations used throughout this scheme are given in Table 2.2.

Table 2.2: Notations used in the NC-Audit scheme

Notation	Description
\mathcal{C}	client
F	original file of \mathcal{C}
m	number of file blocks
\mathbb{F}_q	finite field of a prime order q
$n - 1$	number of elements in \mathbb{F}_q of a file block
M	number of coded blocks stored in a server
$b_i \in \mathbb{F}_q^{n-1}$	file block ($i \in \{1, \dots, m\}$)
$\bar{b}_i \in \mathbb{F}_q^n$	padded block of b_i
$b_i \in \mathbb{F}_q^{n+m}$	augmented block of \hat{b}_i
t_{b_i}	tag of b_i
k_1	MAC key
k_2	encryption key
F_1	pseudo-random function $F_1 : \mathcal{K}_1 \times [1, n + m] \rightarrow \mathbb{F}_q$
F_2	pseudo-random function $F_2 : \mathcal{K}_2 \times ([1, n - 1] \times [1, n - 1]) \rightarrow \mathbb{F}_q$
F_3	pseudo-random function $F_3 : \mathcal{K}_2 \times (\{0, 1\}^\lambda \times [1, n - 1]) \rightarrow \mathbb{F}_q$
e_j	coded block ($j \in \{1, \dots, M\}$)
t_{e_j}	tag of e_j
p_1, \dots, p_{n-1}	tagging elements
$aug(e_j)$	coding coefficients of coded block e_j ($j \in \{1, \dots, M\}$)

We now describe the NC-Audit scheme as follows:

Keygen:

- \mathcal{C} generates MAC key: $k_1 \xleftarrow{rand} \{0, 1\}^\lambda$.
- \mathcal{C} generates encryption key: $k_2 \xleftarrow{rand} \{0, 1\}^\lambda$.

Encode:

- \mathcal{C} divides the file into m blocks of size $(n - 1)$ instead of n :

$$F = \bar{b}_1 || \dots || \bar{b}_m \quad (2.20)$$

Each $\bar{b}_i \in \mathbb{F}_q^{n-1}$ for all $i \in \{1, \dots, m\}$.

- \mathcal{C} pads to each file block $\bar{b}_i \in \mathbb{F}_q^{n-1}$ a random element $rand$ in \mathbb{F}_q . A padded block is denoted by \hat{b}_i .

$$\hat{b}_i = (\bar{b}_i, rand) \in \mathbb{F}_q^n \quad (2.21)$$

- \mathcal{C} creates augmented block for each \hat{b}_i , denoted by b_i :

$$b_i = (\hat{b}_i, \underbrace{0, \dots, 0, 1, 0, \dots, 0}_i^m) \in \mathbb{F}_q^{n+m} \quad (2.22)$$

- \mathcal{C} then setups the encryption scheme by computing the tagging elements, p_1, \dots, p_{n-1} :

- Compute a value \bar{r} :

$$\bar{r} = (F_1(k_1, 1), \dots, F_1(k_1, n-1)) \quad (2.23)$$

- Compute $(n-1)$ values $\bar{p}_1, \dots, \bar{p}_{n-1}$:
 $\forall i \in \{1, \dots, n-1\}$:

$$\bar{p}_i = (F_2(k_2, i, 1), \dots, F_2(k_2, i, n-1)) \in \mathbb{F}_q^{n-1} \quad (2.24)$$

- Compute $(n-1)$ values p_1, \dots, p_{n-1} :
 $\forall i \in \{1, \dots, n-1\}$:

$$p_i = \bar{r} \cdot \bar{p}_i \in \mathbb{F}_q \quad (2.25)$$

- \mathcal{C} computes a tag for each augmented block b_i :

- Compute a value r :

$$r = (F_1(k_1, 1), \dots, F_1(k_1, n+m)) \quad (2.26)$$

- Compute tag for b_i :

$$t_{b_i} = b_i \cdot r \in \mathbb{F}_q \quad (2.27)$$

- \mathcal{C} computes M coded blocks:

$\forall j \in \{1, \dots, M\}$:

$$e_j = \sum_{i=1}^m \alpha_{ij} b_i \in \mathbb{F}_q^{n+m} \quad (2.28)$$

Note that a coded block has the following form:

$$e_j = \underbrace{\left(\sum_{i=1}^m \alpha_{ij} \hat{b}_i \right)}_{\hat{e}_j} \underbrace{(\alpha_{1j}, \dots, \alpha_{mj})}_{\text{aug}(e_j)} \in \mathbb{F}_q^{n+m} \quad (2.29)$$

$\underbrace{\hat{e}_j}_{\bar{e}_j, e_j^{(n)}}$

where $\bar{e}_j \in \mathbb{F}_q^{n-1}$, $e_j^{(n)} \in \mathbb{F}_q$ and $\text{aug}(e_j) \in \mathbb{F}_q^m$.

- \mathcal{C} computes M tags corresponding to M coded blocks:
 $\forall j \in \{1, \dots, M\}$:

$$t_{e_j} = \sum_{i=1}^m \alpha_{ij} t_{b_i} \in \mathbb{F}_q \quad (2.30)$$

- \mathcal{C} sends to the server the following information:
 - Coded blocks: e_1, \dots, e_M
 - Tags: t_{e_1}, \dots, t_{e_M}
 - Tagging elements: p_1, \dots, p_{n-1}
 - Encryption key: k_2
- \mathcal{C} sends to the TPA the following information:
 - Coding coefficients of coded blocks: $aug(e_1), \dots, aug(e_M)$ which are the last m elements of each coded block.
 - MAC key: k_1

The authors assume that \mathcal{C} uses private and authentic channels to send k_1 and k_2 while using an authentic channel for sending the other data. The user then keeps the coding coefficients $(aug(e_1), \dots, aug(e_M))$ for repair and the keys (k_1, k_2) but delete all other data.

Check:

- The TPA chooses a set of indexes of coded blocks to be checked $\mathcal{I} \subseteq \{1, \dots, M\}$, and chooses the coefficients for these blocks uniformly at random: $\gamma_j \xleftarrow{rand} \mathbb{F}_q$ for $j \in \mathcal{I}$. The challenge includes the indexes of the blocks and their corresponding coefficients:

$$chal = \{(j, \gamma_j) | j \in \mathcal{I}\} \quad (2.31)$$

- The server generates the proof of storage, V , is implemented as follows:
 - Compute the aggregated block:

$$\hat{e} = \sum_{j \in \mathcal{I}} \gamma_j \hat{e}_j \quad (2.32)$$

- Parse $\hat{e} = (\bar{e}, e^{(n)})$ where $\bar{e} \in \mathbb{F}_q^{n-1}$ and $e^{(n)} \in \mathbb{F}_q$.
- Compute the aggregated tag:

$$t = \sum_{j \in \mathcal{I}} \gamma_j t_{e_j} \quad (2.33)$$

– Encrypt the response block:

- * Compute $(n - 1)$ values $\bar{p}_1, \dots, \bar{p}_{n-1}$ using k_2 :
 $\forall i \in \{1, \dots, n - 1\}$:

$$\bar{p}_i = (F_2(k_2, i, 1), \dots, F_2(k_2, i, n - 1)) \in \mathbb{F}_q^{n-1} \quad (2.34)$$

- * Choose r uniformly at random: $r \xleftarrow{\text{rand}} \{0, 1\}^\lambda$.

- * Compute the masking coefficients:

$$\forall i \in \{1, \dots, n - 1\}:$$

$$\beta_i = F_3(k_2, r, i) \in \mathbb{F}_q \quad (2.35)$$

- * Compute masking vector:

$$\bar{m} = \sum_{i=1}^{n-1} \beta_i \bar{p}_i \in \mathbb{F}_q^{n-1} \quad (2.36)$$

- * Compute a value \bar{c} :

$$\bar{c} = \bar{e} + \bar{m} \in \mathbb{F}_q^{n-1} \quad (2.37)$$

- * Compute a value p :

$$p = \sum_{i=1}^{n-1} \beta_i p_i \in \mathbb{F}_q \quad (2.38)$$

In essence, the data is masked with a randomly chosen vector $\bar{m} \in \text{span}(\bar{p}_1, \dots, \bar{p}_{n-1})$.

– The server sends to the TPA:

$$\text{resp} = (\bar{c}, \bar{r}, p, e^{(n)}, t) \quad (2.39)$$

- The TPA verifies the proof V as follows:

– Compute coefficients of \hat{e} :

$$\text{aug}(e) = \sum_{j \in \mathcal{I}} \gamma_j \cdot \text{aug}(e_j) \quad (2.40)$$

– Let c as:

$$c = (\bar{c}, e^{(n)}, \text{aug}(e)) \in \mathbb{F}_q^{n+m} \quad (2.41)$$

where $\bar{c} \in \mathbb{F}_q^{n-1}$, $e^{(n)} \in \mathbb{F}_q$ and $\text{aug}(e) \in \mathbb{F}_q^m$.

– Compute a value r :

$$r = (F_1(k_1, 1), \dots, F_1(k_1, n + m)) \in \mathbb{F}_q^{n+m} \quad (2.42)$$

– Compute a value t' :

$$t' = c \cdot r \in \mathbb{F}_q \quad (2.43)$$

– Check if:

$$t' = t + p \quad (2.44)$$

If the equality holds, the TPA will output 1 (the server is healthy). Otherwise, the TPA will output 0 (the server is corrupted).

The correctness of Equation 2.44 is proved as follows:

Proof.

$$\begin{aligned} c &= (\bar{c}, e^{(n)}, \text{aug}(e)) \\ &= ((\bar{e} + \bar{m}), e^{(n)}, \text{aug}(e)) \\ &= e + (\bar{m}, 0, \dots, 0) \end{aligned}$$

In the verification:

$$\begin{aligned} t' &= c \cdot r \\ &= e \cdot r + \bar{m} \cdot \bar{r} \\ &= t + \sum_{i=1}^{n-1} \beta_i \bar{p}_i \cdot \bar{r} \\ &= t + \sum_{i=1}^{n-1} \beta_i p_i \\ &= t + p \end{aligned}$$

Therefore, $t' = t + p$. □

Repair (discuss): When there is a corrupted server, \mathcal{C} creates a new server to replace this corrupted server. Based on the coding coefficients of the coded blocks at the remaining healthy servers, \mathcal{C} instructs the healthy servers to send appropriate coded blocks to the new server. The new server then linearly combines them, according to the user instruction, to construct its own coded blocks.

The verification tags of the newly constructed blocks at the new server do not need to be computed by \mathcal{C} . In particular, the healthy servers can send along the verification tags of the coded blocks that they send to the new server. The new server can generate tags corresponding to the coded blocks that it needs to construct. Finally, \mathcal{C} sends the coding coefficients of the coded blocks at the newly constructed server to the TPA so that it can audit this new server. Finally, the TPA audits the new server based on the new set of coefficients.

2.1.2 Problem Statement

Although many network coding-based PORs have been proposed, none of them satisfies our goals (we mentioned our goals in Section 1.2) because of the following reasons:

- The system models in these previous PORs only have a single client. In other words, multiple clients cannot participant in the system as one of our goals.
- The check phase and repair phase in these previous PORs bring a lot of burden to the client. Concretely:

- The client must periodically check the servers. The task of checking the servers must be executed very often during the system lifetime. Thus, the client incurs very high computation and communication costs during the check phase
- The previous PORs can only support the indirect repair. That is, to repair a corrupted server, the client must require a number of healthy servers to compute the aggregated coded blocks and the aggregated tags. Each of these healthy servers then sends its aggregated coded block along with the corresponding aggregated tag back to the client. After that, the client checks the provided coded blocks using the provided tags, and computes the new coded blocks and new tags to replace the corruption. Finally, the client sends these new coded blocks and new tags to the new server. It is clear that this repair mechanism is a troublesome task for the client. The data repair is performed very often during the system lifetime; thus, the client incurs high computation and communication costs during the repair phase.

Le et al. after that proposed the NC-Audit scheme [62] in which a Third Party Auditor (TPA) is employed and is delegated the responsibility of checking the servers periodically. The client does not need to check the servers any more. The authors also discussed a new repair mechanism in which the new server is able to check the coded blocks provided from the healthy servers, and is able to compute the new coded blocks along with the tags for itself without the need of the client. We call that mechanism as direct repair. Unfortunately, the NC-Audit has the following weak points:

- The direct repair in the scheme is not completed because the authors mainly focused on how to prevent the data leakage from the TPA instead of data repair.
- The scheme is constructed in an asymmetric key setting.
- The scheme does not deal with multiple clients.

Chen et al. [80] also proposed RDC-SR scheme (a Remote Data Checking scheme for replication-based distributed storage which enables Server-side Repair) in which direct repair is supported. However, this scheme is based on replication, not network coding as our objective. Chen et al. [81] afterwards improved their RDC-SR scheme to the RDC-EC scheme (a Remote Data Checking scheme for erasure coding-based distributed storage which enables Server-side Repair). Again, this scheme is based on erasure coding, not network coding as our objective.

To support multiple sources for the network coding, several papers have been discussed [82–85]. However, these schemes also have the following problems:

- In these schemes, the network coding with multiple sources is applied in the network scenario instead of the distributed storage system or the POR as our scenario.
- These schemes are based on an asymmetric key setting instead of a symmetric key setting as one of our goal. These schemes use the digital signatures as the authentication information instead of the MACs.

Opposite with the previous schemes, we propose our scheme to simultaneously address all the drawbacks mentioned above. We now briefly make an overview to compare our contribution with the previous schemes in Table 2.3.

Table 2.3: Previous PORs vs our POR

	Previous PORs	Our PORs
Practicality	(impractical): Only a single client can participate in system.	(practical): Multiple clients can participate in system.
	(impractical): Client <u>cannot</u> perform dynamic operations (modification, insertion, deletion)	(practical): Client <u>can</u> perform dynamic operations (modification, insertion, deletion).
Efficiency	(inefficient): Direct repair <u>is not</u> supported. ⇒ Client is burdened.	(efficient): Direct repair <u>is</u> supported. ⇒ Client is free.
	(inefficient): Public authentication <u>is not</u> supported. ⇒ Client is burdened.	(efficient): Public authentication <u>is</u> supported. ⇒ Client is free.
	(inefficient): Asymmetric key setting is used.	(efficient): Symmetric key setting is used.

2.2 SSS

2.2.1 State Of The Art

Shamir-SSS. SSSs are ideal for storing information that is sensitive. The basic ideas of SSS were independently invented by Shamir [110] and Blakley [111]. A secret S is encoded into n shares. These n shares are distributed to n participants. Each participant

receives one share. This is known as the (m, n) -threshold SSS in which any m or more shares can be used to reconstruct the secret and in which the bit-size of a share is the same as the bit-size of the secret. In a SSS, there is a dealer and n participants (sometimes is called as players). The dealer has responsibility to perform share generation and secret reconstruction. The participants have responsibility to hold their own shares and provide the shares to dealer when the dealer requires. The efficiency of the scheme is evaluated by the entropy of each share, and it must hold that $H(C_i) \geq H(S)$ where $H(S)$ and $H(C_i)$ are the entropies of a secret S and shares C_i where $i \in \{1, \dots, n\}$ [112, 113].

Ramp-SSS. In order to improve the efficiency of the Shamir-SSS, the Ramp-SSS was proposed [114–118] with a trade-off between security and coding efficiency. The Ramp-SSS has three parameters (m, L, n) instead of two parameters (m, n) as in the Shamir-SSS. The (m, L, n) -Ramp-SSS is constructed in a way that the secret S can be reconstructed from any m or more shares; no information about S can be obtained from less than L shares; but a partial information of S can be leaked from any arbitrary set of $(m - t)$ shares with equivocation $(t/m)H(S)$ for $t \in \{1, \dots, m - L\}$. It can attain that $H(C_i) = H(S)/(m - L)$, and that is the reason the Ramp-SSS is more efficient than the Shamir-SSS [114, 115].

However, the drawback of these previous SSSs is the heavy computational cost because the shares are constructed using multiplicative polynomials. An example of a polynomial is the Reed-Solomon code, which takes $O(n \log n)$ field operations for the share distribution and $O(m^2)$ field operations for the secret reconstruction, as showed by Wang et al. in [119].

XOR-based SSS. For the purpose to realize high performance, researchers proposed SSSs which use just XOR operations to make shares and reconstruct the secret, instead of the polynomials. We call these SSSs as XOR-SSSs. Examples include:

- Ishizu et al. [120] proposed a fast $(2, 3)$ -SSS as a solution for the high computational cost due to polynomials.
- Fujii et al. [121], Hosaka et al. [122] and Suga et al. [123] then proposed $(2, n)$ -SSSs to generalize Ishizu’s scheme for the number of participants.
- Kurihara et al. [124], after that, proposed a fast $(3, n)$ -SSS using XOR operations as an extension of Fujii’s scheme by constructing shares with the XOR between the secret and two random numbers.
- Wang et al. [119] proposed the (m, n) -SSS where $m = \{2, 3, 4, n - 3, n - 2, n - 1\}$.

The shortcoming of these XOR-SSSs is that the parameters (m, n) are constrained. For example, m must be 2 and n must be 3 in [120]; m must be 2 in [121–123]; m must be 3 in [124]; and m must be $\{2, 3, 4, n - 3, n - 2, n - 1\}$ in [119]. To extend the previous XOR-SSSs, Shiina et al. [125] firstly proposed another fast (m, n) -SSS using XOR operations. However, the share size in this scheme is larger $O(n^{m-1})$ times of the secret size. Kunii et al. then improve Shiina’s scheme in [126]. However, the share size is larger than or

equal to $\log_2 n$ time the secret size, as indicated in [127]. Afterwards, Kurihara et al. [127], Chunli et al. [128] and Wang et al. [129] proposed the (m, n) -SSSs based on XOR operations in which the share size is equal to the secret size. Kurihara et al. then improved their scheme [127] to achieve the Ramp-SSS in [130].

Most of these previous schemes can support the share generation and the secret reconstruction. Nonetheless, these previous schemes cannot support the direct share repair. This means that when a share is corrupted, without the direct share repair, the dealer must reconstruct the secret S at first and then generate the new share to replace the corruption later. If the direct share repair is supported, the corrupted share can be repaired directly from the remaining healthy shares without the need to reconstruct the secret S .

Network coding-based SSS. Breaking with the flow of previous schemes, recently, there are several SSSs in which the linear network coding is applied to deal with the direct share repair. We call these SSSs as NC-SSSs. Some notable NC-SSSs are proposed by Rashmi et al. [131], Kurihara et al. [132], Tang et al. [133], Shah et al. [134]. Unfortunately, the NC-SSSs have been constructed using a linear combination instead of the XOR.

Overview of a Previous Network Coding-based SSS. In this part, we briefly describe a network coding-based SSS proposed by Rashmi et al. in [131] which is a notable network coding-based SSS.

Let (m, n) denote the parameters of the scheme. Let d denote another parameter such that $2m - 2 \leq d \leq n - 1$. Let $B = m(2d - m + 1)/2$. Let S_1 be a $(m \times m)$ symmetric matrix constructed so that the $m(m + 1)/2$ components in the upper-triangular half of the matrix are filled up by $m(m + 1)/2$ distinct secret symbols drawn from the set of the B secret symbols of the secret. The remaining $m(d - m)$ secret symbols are used to fill up a second $m(d - m)$ matrix S_2 . Let O denote the $(d - m) \times (d - m)$ zero matrix with all zero components. A secret matrix S is then defined as the $(d \times d)$ symmetric matrix given by:

$$S = \begin{pmatrix} S_1 & S_2 \\ S_2^t & O \end{pmatrix} = \begin{pmatrix} u_{1,1} & \cdots & u_{1,m} & u_{1,m+1} & \cdots & u_{1,d} \\ \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ u_{m,1} & \cdots & u_{m,m} & u_{m,m+1} & \cdots & u_{m,d} \\ u_{m+1,1} & \cdots & u_{m+1,m} & 0 & \cdots & 0 \\ \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ u_{d,1} & \cdots & u_{d,m} & 0 & \cdots & 0 \end{pmatrix} \quad (2.45)$$

where S_2^t is the transpose of the matrix S_2 . From the definition of the secret matrix S with components $u_{i,j}$, note that $u_{i,j} = u_{j,i}$ for all $i, j \in \{1, \dots, d\}$ and $u_{i,j} = 0$ for all $i, j \in \{m + 1, \dots, d\}$.

Share Generation: For each $i \in \{1, \dots, n\}$, assign a unique and public symbol x_i in \mathbb{F}_q to participant i in such a way that the following two conditions are satisfied.

Condition for x_i :

- For any $i \in \{1, \dots, n\}$, $x_i \neq 0$.
- For any $i, j \in \{1, \dots, n\}$ if $i \neq j$ then $x_i \neq x_j$.

For the secret matrix S , the share c_i stored in participant i is then defined as:

$$c_i = [c_{i,1}, c_{i,2}, \dots, c_{i,d}]^t = S \cdot \phi_i \in \mathbb{F}_q^d \quad (2.46)$$

where $\phi_i = [1, x_i, x_i^2, \dots, x_i^{d-1}]^t \in \mathbb{F}_q^d$ is a coefficient vector associated with participant i and all operations are done over \mathbb{F}_q . Note that the size of a share is d from $B = m(2d - m + 1)/2$ and from Equation 2.46. Thus, the B secret symbols are encoded to n shares c_1, \dots, c_n .

Secret Reconstruction: From Equation 2.45, we can observe that to reconstruct S , only S_1 and S_2 need to be reconstructed. Let S_{SR} be a matrix which is defined as follows:

$$S = (S_1 \ S_2) = \begin{pmatrix} u_{1,1} & \cdots & u_{1,m} & u_{1,m+1} & \cdots & u_{1,d} \\ \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ u_{m,1} & \cdots & u_{m,m} & u_{m,m+1} & \cdots & u_{m,d} \end{pmatrix} \quad (2.47)$$

S_{SR} consists of $(m \times d)$ unknowns that need to be solved. In addition, from Equation 2.46, each share c_i consists of d elements ($c_i = [c_{i,1}, c_{i,2}, \dots, c_{i,d}]^t$). Therefore, to solve S_{SR} , only m shares are required to reconstitute together. Let C_{SR} denote the matrix which consists of these m shares. Let ϕ_{SR} denote the matrix which consists of the coefficient vectors of these m shares. From Equation 2.46, $c_i = S \cdot \phi_i$. Thus,

$$C_{SR} = S_{SR} \cdot \phi_{SR} \quad (2.48)$$

Therefore, $S_{SR} = C_{SR} \cdot \phi_{SR}^{-1}$. Finally, S can be reconstructed using Equation 2.45.

Share Repair: Suppose a participant \mathcal{P}_f is corrupted and the healthy participants $\mathcal{P}_1, \dots, \mathcal{P}_d$ are used to repair \mathcal{P}_f . For each index $p \in \{1, \dots, d\}$, the healthy participant \mathcal{P}_p computes a piece d_{f,\mathcal{P}_p} for the corrupted participant from the share $c_{\mathcal{P}_p}$ of itself and the coding vector ϕ_f of the corrupted participant as follows:

$$d_{f,\mathcal{P}_p} = c_{\mathcal{P}_p}^t \phi_f \in \mathbb{F}_q, p = 1, \dots, d \quad (2.49)$$

and sends it to the corrupted participant. Note that $id_{f,\mathcal{P}_p} = d_{f,\mathcal{P}_p}^t$ because a piece d_{f,\mathcal{P}_p} is a scalar in \mathbb{F}_q . As a result, the corrupted participant \mathcal{P}_f obtains the piece-vector d_f consisting of d pieces as follows:

$$d_f = [d_{f,\mathcal{P}_1}, d_{f,\mathcal{P}_2}, \dots, d_{f,\mathcal{P}_d}]^t \in \mathbb{F}_q^d \quad (2.50)$$

The sizes of a piece and a piece-vector are one and d , respectively. Using the piece-vector d_f and the d coding vectors $i\phi_{\mathcal{P}_1}, \dots, \phi_{\mathcal{P}_d}$ associated with the healthy participants $\mathcal{P}_1, \dots, \mathcal{P}_d$, the corrupted participant can compute the same share c_f as follows:

$$c_f = ([\phi_{\mathcal{P}_1}, \dots, \phi_{\mathcal{P}_d}]^t)^{-1} d_f \quad (2.51)$$

Note that the $(d \times d)$ matrix $[\phi_{\mathcal{P}_1}, \dots, \phi_{\mathcal{P}_d}]^t$ is non-singular because the determinant of the matrix is the Vandermonde determinant from the conditions for x_i .

2.2.2 Problem Statement

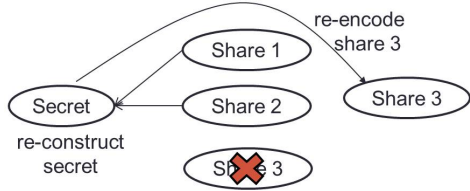
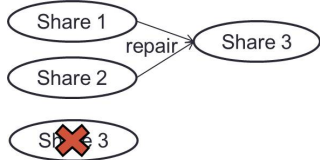
We now summarize all the shortcomings of the previous SSSs that have been mentioned above.

- Most of the previous SSSs only support two functions which are share generation and secret reconstruction, but have not focused on share repair function. In a real system, because a share stored in a participant is probably corrupted, share repair should be considered.
- The parameters (m, n) are constrained. For instance, m must be $\{2, 3, 4, n-3, n-2, n-1\}$. It would be more practical if the parameters can be chosen arbitrarily.
- From a secret, shares are computed by a multiplicative polynomial over a finite field. In a real system, because the parameters (m, n) are very large, the the computation cost to compute the shares and the communication cost to distributed the shares to the participants are required.
- Indirect share repair: when a share stored in a participant is corrupted, the only way to repair the corrupted share is that the dealer must reconstructing the secret by requiring the other shares to reconstitute together. After that, the dealer re-encodes the new share which is used to replace the corrupted share. This mechanism yields a high computation cost for reconstructing the secret.
- Although in most of existing SSSs, the size of a share is less than or equal to the size of the secret, they might not achieve an optimal share size.

Opposite with the previous SSSs, we propose our SSS to simultaneously address all the drawbacks mentioned above. We now briefly make an overview to compare our contribution with the previous SSSs in Table 2.4.

Table 2.4: Previous SSSs vs our SSS

	Previous SSSs	Our SSS
Practicality	(impractical): Two functions are supported: - Share generation - Secret reconstruction	(practical): Three functions are supported: - Share generation - Secret reconstruction

Efficiency		- Share repair
	(impractical): Parameters (m, n) <u>are</u> constrained.	(practical): Parameters (m, n) <u>are not</u> constrained.
	(inefficient): Shares are computed using multiplicative polynomials.	(efficient): Shares are computed using XORs.
	(inefficient): Direct share repair <u>is not</u> supported. \Rightarrow Secret must be reconstructed. 	(efficient): Direct share repair <u>is</u> supported. \Rightarrow Secret does not need to be reconstructed. 
	(inefficient): Share size is not optimal.	(efficient): Share size is optimal.

Chapter 3

Preliminary

3.1 POR

The POR [8–13] is a challenge-response protocol between a verifier (client) and a prover (server) that supports the client to check whether his/her data stored in the servers is always available, intact and retrievable. The POR consists of the functions defined below.

1. **Keygen** (1^λ) \rightarrow $\{\mathbf{sk}, \mathbf{pk}\}$: This function is performed by the client. This function takes a security parameter (λ) as an input, and outputs a pair of $\{\mathbf{sk}, \mathbf{pk}\}$ where \mathbf{sk} denotes the secret key and \mathbf{pk} denotes the public key. For a symmetric key setting, \mathbf{pk} is set to be null.
2. **Encode**(F, \mathbf{sk}) $\rightarrow F^*$: This function allows the client to encode his/her original file (F) to an encoded file (F^*) using the secret key \mathbf{sk} , then stores F^* into the server.
3. **Check**() $\rightarrow \{\text{accept/deny}\}$: This function conducts the challenge-response protocol between the client and the server during which the client uses \mathbf{sk} to generate a challenge (c) and sends the c to the server. The server computes a corresponding response (r) and sends the r back to the client. The client then verifies the server based on c and r , and outputs **accept** (the server is healthy) or **deny** (the server is corrupted).
4. **Repair**(): When a corrupted data from a server is detected in the **check** function, this function is executed by the client to repair the corrupted data. The repair function is depended on the used techniques, e.g., replication, erasure coding, ORAM or network coding.

3.2 Network Coding

Network coding has been proposed to improve the network throughput and the efficiency of data transmission and data repair. Network coding was originally proposed for the network scenario [38–50]. It then is applied to the distributed storage system scenario [51, 60–65].

3.2.1 Fundamental Concept

In the network scenario, suppose that a source node wants to send a message to a receiver node. Before transmitting the message, the source node breaks the message into m blocks $v_1 || \dots || v_m$. Each message block v_i belongs to \mathbb{F}_q^ξ where $i \in \{1, \dots, m\}$ and \mathbb{F}_q^ξ denotes a vector consisting of ξ elements in a finite field \mathbb{F} of a prime order q . The source node augments each message block v_i where $i \in \{1, \dots, m\}$ with a vector of length m in which a single '1' is placed in the i -th position and '0's are placed elsewhere. Let w_1, \dots, w_m denote the augmented blocks. Each augmented block has the following form:

$$w_i = (v_i, \underbrace{0, \dots, 0, 1, 0, \dots, 0}_i) \in \mathbb{F}_q^{\xi+m} \quad (3.1)$$

These augmented blocks are then sent as packets to the network. When an intermediate node in the network receives t packets, the intermediate node will generate t coefficients, linearly combines the t packets using the generated coefficients and transmits the result to its adjacent nodes. Consequently, the receiver node can receive combinations of all augmented blocks. The receiver node can reconstruct the m augmented blocks using any set of m combinations. Suppose that the receiver node receives m packages denoted by $\{y_1, \dots, y_m\}$. Each packet $y_i \in \mathbb{F}_q^{\xi+m}$ where $i \in \{1, \dots, m\}$. The receiver node can solve all m augmented blocks $\{w_1, \dots, w_m\}$ using the accumulated coefficients which are contained in the last m coordinates of each package y_i . Afterwards, each of the m file blocks $\{v_1, \dots, v_m\}$ can be obtained from the first coordinate of each augmented block. Finally, the original message can be reconstructed by concatenating all message blocks.

3.2.2 Application in Distributed Storage System

In the network scenario, there are multiple types of entities: source node, intermediate nodes, and receiver node. However, when the network coding is applied to the distributed storage system scenario, there are two types of entities: a client and servers. Suppose that a client owns an original file F . The client firstly divides F into m file blocks: $F = v_1 || \dots || v_m$. Each file block $v_i \in \mathbb{F}_q^\xi$ where $i \in \{1, \dots, m\}$. The client wants to store redundantly encoded blocks in the servers in a way that the client can reconstruct the original file F and can repair the encoded blocks in a corrupted server. From the m file blocks, the client firstly creates m augmented blocks $\{w_1, \dots, w_m\}$ in which $w_i \in \mathbb{F}_q^{\xi+m}$ where $i \in \{1, \dots, m\}$ has the form as Equation 3.1. The client then randomly chooses m coding coefficients $\alpha_1, \dots, \alpha_m \xleftarrow{\text{rand}} \mathbb{F}_q$ and computes coded blocks using the linear combination as follows:

$$c = \sum_{i=1}^m \alpha_i \cdot w_i \in \mathbb{F}_q^{\xi+m} \quad (3.2)$$

The coded blocks are then stored in the servers. To reconstruct the original file F , any m coded blocks are required to solve m augmented blocks w_1, \dots, w_m using the accumulated coefficients contained in the last m coordinates of each coded block. After

these m augmented blocks are solved, m file blocks v_1, \dots, v_m can be obtained from the first coordinate of each augmented block. Finally, the original file F is reconstructed by concatenating all file blocks.

Note that the matrix consisting of the coefficients used to construct any m coded blocks should have full rank. Koetter et al. [44] proved that if the prime q is chosen large enough and the coefficients are chosen randomly, the probability for the matrix having full rank is high.

When a corrupted server is detected, the client repairs it as follows: the client firstly retrieves coded blocks from the healthy servers and linearly combines them to regenerate new coded blocks. An example about the data repair of network coding is given in Figure 3.1. From three augmented blocks $\{w_1, w_2, w_3\}$, the client computes six coded blocks and stores two coded blocks in each of servers $\mathcal{S}_1, \mathcal{S}_2, \mathcal{S}_3$. Suppose that \mathcal{S}_3 is corrupted, the client requires \mathcal{S}_1 and \mathcal{S}_2 to create new blocks using linear combinations. The client then mixes them using linear combinations to obtain two new coded blocks. The client finally stores the new coded blocks in the new server which is used to replace the corrupted server.

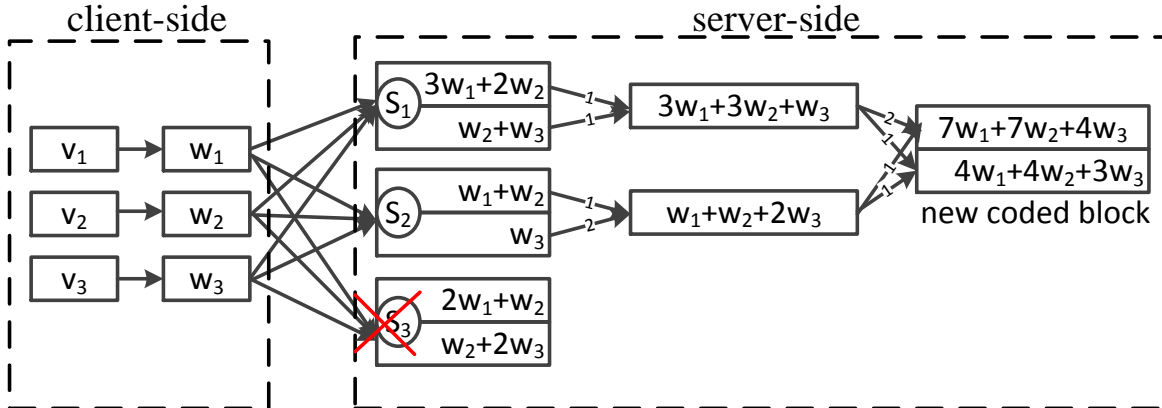


Figure 3.1: An example of data repair of network coding

3.3 Homomorphic MAC

A MAC is proposed to provide integrity and authenticity assurances on the message by allowing verifiers (who also possess the secret key) to detect any changes to the message content. A MAC consists of a tuple of algorithms (KeyGen, Tag, Verify) as follows:

- **KeyGen**(1^λ) $\rightarrow k$: This algorithm takes a security parameter λ as the input, and outputs a secret key k .
- **Tag**(M, k) $\rightarrow t$: This algorithm takes k and a message M as the inputs, and outputs a tag t .

- $\text{Verify}(M, t, k) \rightarrow \{0, 1\}$: This algorithm takes M , t and k as the input, and outputs 1 if t is a valid tag and 0 otherwise.

A MAC is an *additive homomorphic MAC* if it has the following property:

$$\text{Tag}(M + M', k) = \text{Tag}(M, k) + \text{Tag}(M', k) \quad (3.3)$$

A MAC is a *multiplicative homomorphic MAC* if it has the following property:

$$\text{Tag}(M \cdot M', k) = \text{Tag}(M, k) \cdot \text{Tag}(M', k) \quad (3.4)$$

3.3.1 Inner-product MAC

The inner-product MAC consists of the following algorithms:

- $\text{KeyGen}(1^\lambda) \rightarrow k$: This algorithm takes a security parameter λ as the input, and outputs a secret key k .
- $\text{Tag}(M, k) \rightarrow t$: This algorithm takes k and a message M as the inputs, and outputs a tag t such that:

$$t = M \cdot k \quad (3.5)$$

- $\text{Verify}(M, t, k) \rightarrow \{0, 1\}$: This algorithm takes M , t and k as the input, and outputs 1 if t is a valid tag and 0 otherwise.

Theorem 1. *The inner-product MAC is an additive homomorphic MAC.*

Proof. Because $t = M \cdot k$ and $t' = M' \cdot k$, it is easy to see that $t' = (M + M') \cdot k$. Therefore, the inner-product MAC is an additive homomorphic MAC. \square

3.3.2 Inter MAC

The inter MAC consists of the following algorithms:

- $\text{KeyGen}(1^\lambda, M) \rightarrow \{k, k'\}$: This algorithm takes a security parameter λ and a message M as the input, and outputs secret keys $\{k, k'\}$ where $M \cdot k' = 0$.
- $\text{Tag}(M, k, k') \rightarrow t$: This algorithm takes the message M and the secret keys $\{k, k'\}$ as the inputs, and outputs a tag t such that:

$$t = M \cdot (k + k') \quad (3.6)$$

- $\text{Verify}(M, t, k + k') \rightarrow \{0, 1\}$: This algorithm takes M , t and $(k + k')$ as the inputs, and outputs 1 if t is a valid tag and 0 otherwise. Note that this algorithm takes the summation $(k + k')$ as an input (not k and k' separately).

Theorem 2. *The inter MAC is an additive homomorphic MAC.*

Proof. $t = M \cdot (k + k') = M \cdot k$ because $M \cdot k' = 0$. Similarly, $t' = M' \cdot (k + k') = M' \cdot k$ because $M' \cdot k' = 0$. Thus, $t + t' = (M + M') \cdot k$. Therefore, the inter MAC is an additive homomorphic MAC. \square

3.3.3 Inter MAC in Network Coding

The inter MAC is firstly combined with the network coding in the network scenario [79]. We now describe it via two cases: a network with a single source node and a network with multiple source nodes.

a) Single Source Node. Suppose that the single source node owns an original file F . The source node divides F into m blocks: $F = v_1 || \dots || v_m$. $v_i \in \mathbb{F}_q^\xi$ where $i \in \{1, \dots, m\}$. From m file blocks $\{v_1, \dots, v_m\}$, m augmented blocks $\{w_1, \dots, w_m\}$ are created as Equation 3.1. $w_i \in \mathbb{F}_q^{\xi+m}$ where $i \in \{1, \dots, m\}$. The tuple of algorithms (KeyGen-SS, Tag-SS and Verify-SS) is given as follows:

- **KeyGen-SS**($1^\lambda, \{w_1, \dots, w_m\}$) $\rightarrow \{k, k'\}$: This algorithm takes a security parameter λ and a set of m augmented blocks $\{w_1, \dots, w_m\}$ as the inputs, and outputs secret keys $\{k, k'\}$ where $w_i \cdot k' = 0$ for all $i \in \{1, \dots, m\}$.
- **Tag-SS**($\{w_1, \dots, w_m\}, k$) $\rightarrow \{t_1, \dots, t_m\}$: This algorithm takes $\{w_1, \dots, w_m\}$ and k as the inputs, and outputs a set of m tags such that $t_i = w_i \cdot k$ for all $i \in \{1, \dots, m\}$.
- **Verify-SS**($c, t, k + k'$) $\rightarrow \{0, 1\}$: This algorithm takes c, t and $(k + k')$ as the inputs where c and t are the linear combinations of $\{w_1, \dots, w_m\}$ and $\{t_1, \dots, t_m\}$, respectively. In other words, $c = \sum_{i=1}^m \alpha_i w_i$ and $t = \sum_{i=1}^m \alpha_i t_i$ where α_i denotes a coding coefficient. This algorithm outputs 1 if t is a valid tag and 0 otherwise.

The KeyGen-SS introduces a challenge that how to generate k' such that it is orthogonal to all m augmented blocks. Formally, $k' \cdot w_i = 0$ for all $i \in \{1, \dots, m\}$. The algorithm to generate k' is given as follows.

- **OrthogonalGen-SS** (w_1, \dots, w_m) $\rightarrow k'$:
 - Find the span π of $w_1, \dots, w_m \in \mathbb{F}_q^{\xi+m}$.
 - Construct the matrix M in which $\{w_1, \dots, w_m\}$ are the rows of M .
 - Find the null-space of M , denoted by π_M^\perp , which is the set of all vectors $u \in \mathbb{F}_q^{\xi+m}$ such that $M \cdot u^T = 0$.
 - Find the basis vectors of π_M^\perp , denoted by $B_1, \dots, B_\xi \in \mathbb{F}_q^{\xi+m}$ // Theorem 3 will explain why the number of the basis vectors is ξ .
 - Compute $k' \leftarrow \text{Kg-SS}(B_1, \dots, B_\xi)$.
- **Kg-SS**(B_1, \dots, B_ξ) $\rightarrow k'$: this is the sub-algorithm used in the OrthogonalGen-SS algorithm:
 - Let f be a pseudo-random function such that $\mathcal{K} \times [1, \xi] \rightarrow \mathbb{F}_q$.
 - Generate $r_x \leftarrow f(k_{PRF}, x) \in \mathbb{F}_q, \forall x \in \{1, \dots, \xi\}$ where $k_{PRF} \in \mathcal{K}$.
 - Compute $k' \leftarrow \sum_{x=1}^\xi r_x \cdot B_x \in \mathbb{F}_q^{\xi+m}$.

Theorem 3. Given $\{w_1, \dots, w_m\} \in \mathbb{F}_q^{\xi+m}$, the number of basis vectors of π_M^\perp is ξ .

Proof. $\text{rank}(M) = m$. Let π_M be the space spanned by the rows of M . For any $m \times (\xi+m)$ matrix, the rank-nullity theorem gives:

$$\text{rank}(M) + \text{nullity}(M) = \xi + m \quad (3.7)$$

where $\text{nullity}(M)$ is the dimension of π_M^\perp . Thus, we have:

$$\dim(\pi_M^\perp) = (\xi + m) - m = \xi \quad (3.8)$$

Therefore, the number of basis vectors of π_M^\perp is ξ . In the **OrthogonalGen-SS** algorithm, we denoted the basis vectors by B_1, \dots, B_ξ . \square

b) Multiple Source Nodes. Suppose that there are s source nodes, denoted by $\{\mathcal{C}_1, \dots, \mathcal{C}_s\}$. Each client \mathcal{C}_i where $i \in \{1, \dots, s\}$ owns a file F_i which consists of g file blocks: $F_i = v_{i1} || \dots || v_{ig}$. $v_{ij} \in \mathbb{F}_q^\xi$ where $i \in \{1, \dots, s\}, j \in \{1, \dots, g\}$. Suppose that all the clients have the same number of file blocks (g is the same for all $\mathcal{C}_1, \dots, \mathcal{C}_s$). Let $m = s \cdot g$. The set of all m file blocks is: $\{v_{11}, \dots, v_{1g}, \dots, v_{s1}, \dots, v_{sg}\}$. From these m file blocks, the m augmented blocks $\{w_{11}, \dots, w_{1g}, \dots, w_{s1}, \dots, w_{sg}\}$ are created as follows:

$$w_{ij} = (v_{ij}, \underbrace{0, \dots, 0}_{g(i-1)}, \underbrace{0, \dots, 0, 1, 0, \dots, 0}_g, \underbrace{0, \dots, 0}_{g(s-i)}) \in \mathbb{F}_q^{\xi+m} \quad (3.9)$$

$m = s \cdot g$

where $i \in \{1, \dots, s\}, j \in \{1, \dots, g\}$, $m = s \cdot g$. The tuple of algorithms (**KeyGen-MS**, **Tag-MS**, **Verify-MS**) is given as follows:

- **KeyGen-MS** $(1^\lambda, \{w_{11}, \dots, w_{1g}, \dots, w_{s1}, \dots, w_{sg}\}) \rightarrow \{k_1, \dots, k_s\}$: This algorithm takes a security parameter λ and the set of $m = s \cdot g$ augmented blocks $\{w_{11}, \dots, w_{1g}, \dots, w_{s1}, \dots, w_{sg}\}$ as the inputs, and outputs a set of s secret keys $\{k_1, \dots, k_s\}$ such that $w_{ij} \cdot k_p = 0$ for all $i, p \in \{1, \dots, s\}, j \in \{1, \dots, g\}$ and $p \neq i$.
- **Tag-MS** $(\{w_{11}, \dots, w_{1g}, \dots, w_{s1}, \dots, w_{sg}\}, \{k_1, \dots, k_s\}) \rightarrow \{t_{11}, \dots, t_{1g}, \dots, t_{s1}, \dots, t_{sg}\}$: This algorithm takes the set of $m = s \cdot g$ augmented blocks $\{w_{11}, \dots, w_{1g}, \dots, w_{s1}, \dots, w_{sg}\}$ and the set of s secret keys $\{k_1, \dots, k_s\}$ as the inputs, and outputs a set of m tags such that $t_{ij} = w_{ij} \cdot k_i$ for all $i \in \{1, \dots, s\}$ and $j \in \{1, \dots, g\}$.
- **Verify-MS** $(c, t, k_1 + \dots + k_s) \rightarrow \{0, 1\}$: This algorithm takes c , t and $(k_1 + \dots + k_s)$ as the inputs where c and t are the linear combinations of $\{w_{11}, \dots, w_{1g}, \dots, w_{s1}, \dots, w_{sg}\}$ and $\{t_{11}, \dots, t_{1g}, \dots, t_{s1}, \dots, t_{sg}\}$, respectively. In other words, $c = \sum_{i=1}^m \alpha_i w_i$ and $t = \sum_{i=1}^m \alpha_i t_i$ where α_i denotes a coding coefficient. This algorithm outputs 1 if t is a valid tag and 0 otherwise.

Similar to the **KeyGen-SS**, the **KeyGen-MS** also introduces a challenge that how to generate k_p where $p \in \{1, \dots, s\}$ such that it is orthogonal to the augmented blocks which do not belong to the source node \mathcal{C}_p . Formally, $w_{ij} \cdot k_p = 0$ for all $i, p \in \{1, \dots, s\}$, $j \in \{1, \dots, g\}$ and $i \neq p$. The algorithm to generate k_p is given as follows.

- **OrthogonalGen-MS** $(p, \{w_{ij} \in \mathbb{F}_q^{\xi+m} | i = 1, \dots, s; i \neq p; j = 1, \dots, g\}) \rightarrow k_p$:
 - Find the span π of the set $\{w_{ij} \in \mathbb{F}_q^{\xi+m} | i = 1, \dots, s; i \neq p; j = 1, \dots, g\}$. The set consists of $(m - g)$ elements.
 - Construct the matrix M in which the above $(m - g)$ elements in the set are the rows of M .
 - Find the null-space of M , denoted by π_M^\perp , which is the set of all vectors $u \in \mathbb{F}_q^{\xi+m}$ such that $M \cdot u^T = 0$.
 - Find the basis vectors of π_M^\perp , denoted by $B_1, \dots, B_{\xi+g} \in \mathbb{F}_q^{\xi+m}$ // Theorem 4 will explain why the number of the basis vectors is $(\xi + g)$.
 - Compute $k_p \leftarrow \mathbf{Kg-MS}(B_1, \dots, B_{\xi+g})$.
- **Kg-MS** $(B_1, \dots, B_{\xi+g}) \rightarrow k_p$: this is the sub-algorithm used in the **OrthogonalGen-MS** algorithm.
 - Let f be a pseudo-random function such that $\mathcal{K} \times [1, \xi + g] \rightarrow \mathbb{F}_q$.
 - Generate $r_x \leftarrow f(k_{PRF}, x) \in \mathbb{F}_q, \forall x \in \{1, \dots, \xi + g\}$ where $k_{PRF} \in \mathcal{K}$.
 - Compute $k_p \leftarrow \sum_{x=1}^{\xi+g} r_x \cdot B_x \in \mathbb{F}_q^{\xi+m}$.

Theorem 4. *Given an integer $p \in \{1, \dots, s\}$ and the set $\{w_{ij} \in \mathbb{F}_q^{\xi+m} | i = 1, \dots, s; i \neq p; j = 1, \dots, g\}$, the number of basis vectors of π_M^\perp is $(\xi + g)$.*

Proof. $\text{rank}(M) = m - g$. Let π_M be the space spanned by the rows of M . For any $(m - g) \times (\xi + m)$ matrix, the rank-nullity theorem gives:

$$\text{rank}(M) + \text{nullity}(M) = \xi + m \quad (3.10)$$

where $\text{nullity}(M)$ is the dimension of π_M^\perp . Thus, we have:

$$\dim(\pi_M^\perp) = (\xi + m) - (m - g) = \xi + g \quad (3.11)$$

Therefore, the number of basis vectors of π_M^\perp is $(\xi + g)$. In the **OrthogonalGen-MS** algorithm, we denoted the basis vectors by $B_1, \dots, B_{\xi+g}$. \square

3.4 Dispersal Coding

To prevent the small corruption attack and to allow the client to repair the data with a high probability, the dispersal coding is proposed [26] with a minimal additional storage overhead.

3.4.1 Building Block

The dispersal coding is constructed into a single primitive called *codeword* using the following building blocks.

Universal Hash Function (UHF). A UHF [93] is a function $h: \mathcal{K} \times \mathcal{I}^l \rightarrow \mathcal{I}$ where \mathcal{I} denotes a field with operations $(+, \times)$. This UHF compresses a message $m \in \mathcal{I}^l$ into a compact digest based on a key $\kappa \in \mathcal{K}$ such that the hash of two different messages is different with an overwhelming probability over keys. A common UHF is almost XOR universal (AXU) which satisfies:

- h is an ϵ -UHF family if:

$$\forall x \neq y \in \mathcal{I}^l : \Pr_{\kappa \leftarrow \mathcal{K}}[h_\kappa(x) = h_\kappa(y)] \leq \epsilon \quad (3.12)$$

- h is an ϵ -AXU family if:

$$\forall x \neq y \in \mathcal{I}^l, \forall z \in \mathcal{I} : \Pr_{\kappa \leftarrow \mathcal{K}}[h_\kappa(x) \oplus h_\kappa(y) = z] \leq \epsilon \quad (3.13)$$

- If a UHF is linear, for any message pair (m_1, m_2) :

$$h_\kappa(m_1) + h_\kappa(m_2) = h_\kappa(m_1 + m_2) \quad (3.14)$$

Error-Correcting Code (ECC). An ECC [94–96] is used to express a sequence of the original data and the parity data such that any errors can be detected and corrected. An ECC has two parameters (n, l) where l denotes the number of the original blocks, and n denotes the number of blocks after adding $(n - l)$ redundant blocks. There exists (n, l) -ECC codes that can correct up to $t = \frac{n-l+1}{2}$ errors. The Reed-Solomon code (RS) [97, 98] is a kind of ECC which uses a special polynomial:

$$g(x) = (x - a^i)(x - a^{i+1}) \cdots (x - a^{i+2t}) \quad (3.15)$$

The codeword of the RS code is:

$$c(x) = g(x) \cdot i(x) \quad (3.16)$$

where $g(x)$ is the generator polynomial over \mathbb{F}_p , $i(x)$ is the information block, and a is a primitive element of the field.

- *Encoder*: The $2t$ parity symbols are given by:

$$p(x) = i(x) \cdot x^{n-l} \mod g(x) \quad (3.17)$$

- *Decoder*: Given a codeword $r(x)$ which is the original codeword $c(x)$ plus errors:

$$r(x) = c(x) + e(x) \quad (3.18)$$

the RS decoder identifies the position and magnitude of up to t and corrects the errors.

- Calculating the syndrome: The RS codeword has $2t$ syndromes that depend on errors. The syndromes are calculated by substituting the $2t$ roots of $g(x)$ into $r(x)$.
- Finding the symbol error locations: This involves solving the equations with t unknowns. The first step is to find an error locator polynomial using the Berlekamp-Massey algorithm or the Euclid algorithm. The second step is to find the roots of this polynomial using the Chien search algorithm.
- Finding symbol error values: This involves solving the equations with t unknowns using the Forney algorithm.

Universal Hash Function which is constructed using the Reed-Solomon code (RS-UHF). A RS-UHF [99] is constructed in a way as follows. Suppose that a message m is a vector $\vec{m} = (m_1, \dots, m_l)$ where $m_i \in \mathcal{I}$ and suppose that an (n, l) -RS code over \mathcal{I} is used. \vec{m} is viewed as a polynomial representation of the form:

$$p_{\vec{m}} = m_l x^{l-1} + m_{l-1} x^{l-2} + \dots + m_1 \quad (3.19)$$

A RS code can be defined as a vector $\vec{k} = (k_1, \dots, k_n)$. The codeword of a message \vec{m} is the evaluation of polynomial $p_{\vec{m}}$ at point (k_1, \dots, k_n) : $(p_{\vec{m}}(k_1), \dots, p_{\vec{m}}(k_n))$. A UHF is $h_{\kappa}(m) = p_{\vec{m}}(\kappa)$ where κ is the key.

Message Authentication Code (MAC). A MAC [100] is used to authenticate a message and to detect message tampering and forgery. A MAC is a tuple of (MGen, MTag, MVer):

- MGen(1^λ): generates a secret key κ given a security parameter λ .
- MTag $_{\kappa}(m)$: computes a tag τ for the message m with the key κ .
- MVer $_{\kappa}(m, \tau)$: outputs 1 if τ is a valid tag, and 0 otherwise.

Pseudo-random Function (PRF). A PRF [101] is used to generate exponentially many random bits in a way that behaves like a random function. A PRF is a keyed family of a function $g : \mathcal{K}_{\text{PRF}} \times \mathcal{L} \rightarrow \mathcal{I}$ which is indistinguishable from a random family of functions from \mathcal{L} to \mathcal{I} . A PRF can be constructed from any pseudorandom generator as follows:

- Let $G : \{0, 1\}^n \rightarrow \{0, 1\}^{2n}$ be a length-doubling pseudorandom generator.
- Define $G_0 : \{0, 1\}^n \rightarrow \{0, 1\}^n$ such that $G_0(x)$ is the first n bits of $G(x)$.
- Define $G_1 : \{0, 1\}^n \rightarrow \{0, 1\}^n$ such that $G_1(x)$ is the last n bits of $G(x)$.
- For the key $\mathcal{K}_{\text{PRF}} \in \{0, 1\}^n$ and an input $x \in \{0, 1\}^n$, the PRF is constructed as:

$$F_{\mathcal{K}_{\text{PRF}}}(x) = G_{x_n}(G_{x_{n-1}}(\dots G_{x_2}(G_{x_1}(\mathcal{K}_{\text{PRF}})) \dots)) \quad (3.20)$$

where $x_i \in \{0, 1\}$ ($i = 1, \dots, n$) are the elements of x ($x = \{x_1, \dots, x_n\} \in \{0, 1\}^n$).

MAC based on Universal Hash Function (UMAC). A UMAC [99] can be constructed as the composition of a UHF with a PRF. Given a UHF family $h : \mathcal{K}_{\text{UHF}} \times \mathcal{I}^l \rightarrow \mathcal{I}$ and a PRF family $g : \mathcal{K}_{\text{PRF}} \times \mathcal{L} \rightarrow \mathcal{I}$, the UMAC is a tuple of $\text{UMAC} = (\text{UGen}, \text{UTag}, \text{UVer})$:

- $\text{UGen}(1^\lambda)$: generates key (κ, κ') uniformly at random from $\mathcal{K}_{\text{UHF}} \times \mathcal{K}_{\text{PRF}}$.
- $\text{UTag}_{\kappa, \kappa'}(m)$: works in space $\mathcal{K}_{\text{UHF}} \times \mathcal{K}_{\text{PRF}} \times \mathcal{I}^l \rightarrow \mathcal{L} \times \mathcal{I}$, outputs $(r, h_\kappa(m) + g_{\kappa'}(r))$ in which a unique counter $r \in \mathcal{L}$ is increased in each execution.
- $\text{UVer}_{\kappa, \kappa'}(m, (c_1, c_2))$: works in space $\mathcal{K}_{\text{UHF}} \times \mathcal{K}_{\text{PRF}} \times \mathcal{I}^l \times \mathcal{L} \times \mathcal{I}$, outputs 1 if and only if $h_\kappa(m) + g_{\kappa'}(c_1) = c_2$.

Dispersal coding. The dispersal coding [26] is constructed as follows: To tag a message, the message is encoded under an (n, l) -RS code; and a PRF is then applied to the last s code symbols ($s \in \{1, \dots, n\}$). A MAC is obtained on each of those s code symbols using UMAC. A codeword is valid if at least one of the last s symbols is the valid MAC.

- $\text{KGenECC}(1^\lambda)$: selects key $\vec{\kappa} = \{\{\kappa_i\}_{i=1}^n, \{\kappa'_i\}_{i=n-s+1}^n\}$ randomly from space $\mathcal{K} = \mathcal{I}^n \times (\mathcal{K}_{\text{PRF}})^s$. The keys $\{\kappa_i\}_{i=1}^n$ are used for the RS code. The keys $\{\kappa'_i\}_{i=n-s+1}^n$ are used for the PRF in the UMAC.
- $\text{MTagECC}_{\vec{\kappa}}(m_1, \dots, m_l)$: outputs (c_1, \dots, c_n) in which $c_i = \text{RS-UHF}_{\kappa_i}(\vec{m})$ when $i \in \{1, \dots, n-s\}$ and $c_i = \text{UTag}_{\kappa_i, \kappa'_i}(m_1, \dots, m_l) = (r_i, \text{RS-UHF}_{\kappa_i}(\vec{m}) + g_{\kappa'_i}(r_i))$ when $i \in \{n-s+1, \dots, n\}$.
- $\text{MVerECC}_{\vec{\kappa}}(c_1, \dots, c_n)$: strips off the PRF from (c_{n-s+1}, \dots, c_n) as: $c'_i = c_i - g_{\kappa'_i}(r_i)$ where $i \in \{n-s+1, \dots, n\}$, and then decodes $(c_1, \dots, c_{n-s}, c'_{n-s+1}, \dots, c'_n)$ using the RS decoder to obtain the message $\vec{m} = (m_1, \dots, m_l)$. If the RS decoder fails at the point $\{\kappa_i\}_{i=1}^n$ (when the number of corruptions is more than $\frac{n-l+1}{2}$), MVerECC outputs $(\perp, 0)$. If one of the last s symbols of (c_1, \dots, c_n) is a valid MAC on \vec{m} under UMAC, MVerECC outputs $(\vec{m}, 1)$, otherwise it outputs $(\vec{m}, 0)$.

Because the dispersal coding uses the RS code in MTagECC to tag the message and uses the RS decoder in MVerECC to verify, the dispersal coding can prevent the small corruption attack.

3.5 Shamir SSS

The Shamir-SSS [110, 111] consists of n participants $\mathcal{P} = \{\mathcal{P}_1, \dots, \mathcal{P}_n\}$ and a dealer. Two algorithms ShareGen and Reconst are run by the dealer. The share generation ShareGen algorithm takes a secret S as the input and outputs a set $C = \{c_1, \dots, c_n\}$. c_i where $i \in \{1, \dots, n\}$ is called a *share* and is given to a participant \mathcal{P}_i . The secret reconstruction algorithm Reconst inputs a set of m shares and outputs the secret S . A (m, n) -Shamir-SSS has the following properties:

- perfect SSS: any $m \leq n$ participants or more can reconstruct the secret S and no $(m - 1)$ participants or less can learn any information of the secret S . Formally, let $H(S)$ and $H(A)$ denote the entropy of the secret S and a set of shares $A \subseteq C$, respectively.

$$H(S|A) = \begin{cases} 0, & \text{if } |A| \geq m \\ H(S), & \text{if } |A| < m \end{cases} \quad (3.21)$$

- ideal SSS: the size of a share is the same as the size of the secret: $|c_i| = |S|$.

Example 3.5.1. Suppose the secret $S = 13$. All operations work in $\mathbb{F}_p = \mathbb{F}_{17}$. The thresholds $(m, n) = (3, 5)$. The polynomial which is used to generate the shares has the following form:

$$f(x) = S + \sum_{i=1}^{m-1} a_i x^i \pmod{p} \quad (3.22)$$

where $a_i \xleftarrow{\text{rand}} \mathbb{F}_p$. Because $n = 5$, the dealer chooses 5 inputs of the polynomial to compute 5 shares. Suppose that these inputs are: $id = [x_1, x_2, x_3, x_4, x_5] = [1, 2, 3, 4, 5]$.

- *Share generation:* The dealer chooses the coefficients $a_1 = 10 \in \mathbb{F}_{17}$ and $a_2 = 2 \in \mathbb{F}_{17}$. The polynomial is:

$$f(x) = 13 + 10x + 2x^2 \pmod{17} \quad (3.23)$$

Because $n = 5$, the dealer then computes 5 shares as follows:

- For $x_1 = 1$, compute $f(x_1) = 8$, then sends the pair of $[x_1, f(x_1)] = [1, 8]$ to the participant \mathcal{P}_1 .
- For $x_2 = 2$, compute $f(x_2) = 7$, then sends the pair of $[x_2, f(x_2)] = [2, 7]$ to the participant \mathcal{P}_2 .
- For $x_3 = 3$, compute $f(x_3) = 10$, then sends the pair of $[x_3, f(x_3)] = [3, 10]$ to the participant \mathcal{P}_3 .
- For $x_4 = 4$, compute $f(x_4) = 0$, then sends the pair of $[x_4, f(x_4)] = [4, 0]$ to the participant \mathcal{P}_4 .
- For $x_5 = 5$, compute $f(x_5) = 11$, then sends the pair of $[x_5, f(x_5)] = [5, 11]$ to the participant \mathcal{P}_5 .
- *Secret reconstruction:* To reconstruct the secret S , the dealer requires $m = 3$ participants to provide their shares. Suppose that $\{\mathcal{P}_1, \mathcal{P}_3, \mathcal{P}_5\}$ are required for reconstructing S . After obtaining $[x_1, f(x_1)]$, $[x_3, f(x_3)]$ and $[x_5, f(x_5)]$, the dealer solves S using the following equation system:

$$\begin{cases} f(x_1) = s + a_1x_1 + a_2(x_1)^2 \pmod{17} \\ f(x_3) = s + a_1x_3 + a_2(x_3)^2 \pmod{17} \\ f(x_5) = s + a_1x_5 + a_2(x_5)^2 \pmod{17} \end{cases} \quad (3.24)$$

Because $[x_1, f(x_1)] = [1, 8]$, $[x_3, f(x_3)] = [3, 10]$ and $[x_5, f(x_5)] = [5, 11]$, the equation system becomes:

$$\begin{cases} 8 = s + a_1 \cdot 1 + a_2 \cdot 1^2 \pmod{17} \\ 10 = s + a_1 \cdot 3 + a_2 \cdot 3^2 \pmod{17} \\ 11 = s + a_1 \cdot 5 + a_2 \cdot 5^2 \pmod{17} \end{cases} \quad (3.25)$$

The solution of the equation system is:

$$\begin{cases} s = 13 \\ a_1 = 10 \\ a_2 = 2 \end{cases} \quad (3.26)$$

Not only the secret S is reconstructed, all the coefficients (a_1 and a_2) are also reconstructed.

3.6 Ramp SSS

The Ramp-SSS [114–118] was proposed to improve the coding efficiency of the Shamir-SSS. The Ramp-SSS has three parameters (m, L, n) instead of two parameters (m, n) like the Shamir-SSS. The Ramp-SSS is constructed in a way that any m shares or more can reconstruct the secret S ; any set of $(m - t)$ shares where $t \in \{1, \dots, m - L\}$ can learn a partial information of the secret S ; and any L shares or less cannot obtain any information of the secret S . Formally,

$$H(S|A) = \begin{cases} H(S), & \text{if } |A| < L \\ \frac{m-|A|}{m} H(S), & \text{if } L \leq |A| < m \\ 0, & \text{if } |A| \geq m \end{cases} \quad (3.27)$$

where $H(S)$ and $H(A)$ denote the entropy of the secret S and a set of shares $A \subseteq C$, respectively. The Ramp-SSS is more efficient than the Shamir-SSS because in any (m, L, n) -Ramp-SSS, $H(C_i) = H(S)/(m - L)$ [114, 115].

Example 3.6.1. Suppose that the secret $S = [S_1, S_2] = [83, 102]$. All operations work in $\mathbb{F}_p = \mathbb{F}_{10729}$. Suppose that $(m, L, n) = (3, 2, 7)$. The polynomial which is used to generate the shares has the following form:

$$f(x) = \sum_{i=0}^{m-1} a_i x^i \pmod{p} \quad (3.28)$$

where $a_i \xleftarrow{\text{rand}} \mathbb{F}_p$. Because $n = 7$, the dealer chooses 7 inputs of the polynomial to compute 7 shares. Suppose that these inputs are: $id = [x_1, x_2, x_3, x_4, x_5, x_6, x_7] = [1, 2, 3, 4, 5, 6, 7]$. Besides these inputs id , the dealer chooses m more inputs for the polynomial: $sid = [x_8, x_9, x_{10}] = [8, 9, 10]$. The dealer chooses $(m - L) = (3 - 2) = 1$ value in \mathbb{F}_{10729} (suppose the value is $R = 123$).

- *Share generation:* Because $m = 3$, the polynomial is:

$$f(x) = a_0 + a_1x + a_2x^2 \pmod{10729} \quad (3.29)$$

Before computing the shares, the dealer finds the coefficients of the polynomial by solving the following equation system:

$$\begin{cases} S_1 = a_0 + a_1x_8 + a_2(x_8)^2 \\ S_2 = a_0 + a_1x_9 + a_2(x_9)^2 \\ R = a_0 + a_1x_{10} + a_2(x_{10})^2 \end{cases} \quad (3.30)$$

Replace $(S_1 = 83, x_8 = 8), (S_2 = 102, x_9 = 9), (R = 123, x_{10} = 10)$:

$$\begin{cases} 83 = a_0 + a_18 + a_2(8)^2 \\ 102 = a_0 + a_19 + a_2(9)^2 \\ 123 = a_0 + a_110 + a_2(10)^2 \end{cases} \quad (3.31)$$

The coefficients are solved as follows:

$$\begin{cases} a_0 = 3 \\ a_1 = 2 \\ a_2 = 1 \end{cases} \quad (3.32)$$

Thus, the polynomial becomes:

$$f(x) = 3 + 2x + 1x^2 \pmod{10729} \quad (3.33)$$

The dealer is now ready for computing $n = 7$ shares as follows:

- For $x_1 = 1$, compute $f(x_1) = 6$, then sends the pair of $[x_1, f(x_1)] = [1, 6]$ to the participant \mathcal{P}_1 .
- For $x_2 = 2$, compute $f(x_2) = 11$, then sends the pair of $[x_2, f(x_2)] = [2, 11]$ to the participant \mathcal{P}_2 .
- For $x_3 = 3$, compute $f(x_3) = 18$, then sends the pair of $[x_3, f(x_3)] = [3, 18]$ to the participant \mathcal{P}_3 .
- For $x_4 = 4$, compute $f(x_4) = 27$, then sends the pair of $[x_4, f(x_4)] = [4, 27]$ to the participant \mathcal{P}_4 .
- For $x_5 = 5$, compute $f(x_5) = 38$, then sends the pair of $[x_5, f(x_5)] = [5, 38]$ to the participant \mathcal{P}_5 .

- For $x_6 = 6$, compute $f(x_6) = 51$, then sends the pair of $[x_6, f(x_6)] = [6, 51]$ to the participant \mathcal{P}_6 .
- For $x_7 = 7$, compute $f(x_7) = 66$, then sends the pair of $[x_7, f(x_7)] = [7, 66]$ to the participant \mathcal{P}_7 .
- *Secret reconstruction:* To reconstruct the secret S , the dealer requires $m = 3$ participants to provide their shares. Suppose that $\{\mathcal{P}_1, \mathcal{P}_3, \mathcal{P}_4\}$ are required for reconstructing S . After obtaining $[x_1, f(x_1)]$, $[x_3, f(x_3)]$ and $[x_4, f(x_4)]$, the dealer solves S using the following equation system:

$$\begin{cases} f(x_1) = a_0 + a_1x_1 + a_2(x_1)^2 & (\text{mod } 10729) \\ f(x_3) = a_0 + a_1x_3 + a_2(x_3)^2 & (\text{mod } 10729) \\ f(x_4) = a_0 + a_1x_4 + a_2(x_4)^2 & (\text{mod } 10729) \end{cases} \quad (3.34)$$

Because $[x_1, f(x_1)] = [1, 6]$, $[x_3, f(x_3)] = [3, 18]$ and $[x_4, f(x_4)] = [4, 27]$, the equation system becomes:

$$\begin{cases} 6 = a_0 + a_1 \cdot 1 + a_2 \cdot 1^2 & (\text{mod } 10729) \\ 18 = a_0 + a_1 \cdot 3 + a_2 \cdot 3^2 & (\text{mod } 10729) \\ 27 = a_0 + a_1 \cdot 4 + a_2 \cdot 4^2 & (\text{mod } 10729) \end{cases} \quad (3.35)$$

The solution of the equation system is:

$$\begin{cases} a_0 = 3 \\ a_1 = 2 \\ a_2 = 1 \end{cases} \quad (3.36)$$

Thus, the dealer can recover the polynomial as:

$$f(x) = a_0 + a_1x + a_2x^2 \quad (\text{mod } 10729) \quad (3.37)$$

Then the secret S is reconstructed as follows:

- For $x_8 = 8$, compute $f(x_8) = 83$. Thus, the first secret is $S_1 = 83$.
- For $x_9 = 9$, compute $f(x_9) = 102$. Thus, the second secret is $S_2 = 102$.

Lastly, the secret is $S = [S_1, S_2] = [83, 102]$.

3.7 SWC

The SWC was firstly proposed by Slepian and Wolf in 1973 [140] to compress data in a network. While the possibility for the practical implementation of SWC was suggested in [141–143], it was not until late 90s that SWC was actually implemented. The SWC has

several approaches: syndrome-based, binning idea, Low Density Parity Check (LDPC)-based and parity-based [144–151].

For the most efficient computation, we use the *binning idea*. While the concept of binning is simple, it is extremely powerful. In essence, the source space is partitioned into bins and the encoder identifies the bin in which a input source belongs to. The label of the bin instead of the source itself is passed to the decoder, which will then estimate the original source based on the bin information and the correlation among sources. More concretely, suppose that a source node has two data b_1 and b_2 which have the same size. To compress, b_1 is divided into a number of bins. During encoding, the index of the bin that the input belongs to is transmitted to the receiver node instead of the input itself. For example, $|b_1|$ is divided into k bins. Each bin contains $\frac{|b_1|}{k}$ elements. If the SWC is not used, $\log_2 |b_1|$ bits are required to transmit the input to the receiver node. If the SWC is used, only $\log_2 k$ bits are required to transmit the input to the receiver node. The receiver node cannot decode b_1 if b_2 does not present because the receiver node cannot know the corresponding element of the bin index. If b_2 is obtained, the receiver node can decode b_1 by picking the element in the bin that is best matched with b_2 .

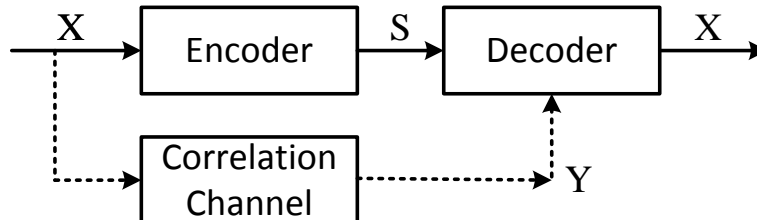


Figure 3.2: SWC

To consolidate the idea, we consider two numerical examples that were introduced by Pradhan et al. [152] and then re-presented in [149].

Example 3.7.1. This example is a compression of a 3-bit sequence with side information. Consider two binary sequences of length 3 and the two sequences are correlated in such a way that they can differ no more than 1 bit. Denote the two sequences as X and Y , respectively. Assuming that Y is given to both the encoder and the decoder, apparently we can compress X into 2 bits since we can represent and transmit $X - Y \in \{[000]^T, [001]^T, [010]^T, [100]^T\}$ with $\log_2 4 = 2$ bits instead of transmitting X directly. At the decoder, given XY and the side information Y , X can be recovered as $(XY) + Y$.

Example 3.7.2. This example is an asymmetric SWC of a 3-bit sequence. Continuing with the above example, the side information Y is now given only to the decoder but not

the encoder. So the encoder cannot compute XY any more. There is a challenge that how can we still compress X into two bits.

Let us partition all possibilities of X into 4 bins as follows:

- Bin 0: $\{[000]^T, [111]^T\}$
- Bin 1: $\{[001]^T, [110]^T\}$
- Bin 2: $\{[010]^T, [101]^T\}$
- Bin 3: $\{[100]^T, [011]^T\}$

Now, let say $X = [001]^T$ and $Y = [000]^T$. At the decoder, the index of the bin that $X = [001]^T$ lies into, i.e., 1, will be transmitted to the decoder. Since there are only 4 bins, we only need $\log_2 4 = 2$ bits to represent the bin index. From the bin index, the decoder knows that X is either $[001]^T$ or $[110]^T$. Since it is given that X and Y can differ no more than 1 bit. We know that X is $[001]^T$.

Chapter 4

MD-POR: Multi-client and Direct Repair for POR

4.1 System Model

As depicted in Figure 4.1, the system model of our proposed MD-POR scheme consists of four types of entities:

- **Key manager:** This entity is fully trusted, and has the responsibility to generate the keys for the other entities.
- **Clients:** There are multiple clients who can be either enterprises or individual customers. Each client owns his/her data and wants to store the data in the cloud servers. The clients rely on the cloud for data storage, computation, and maintenance.
- **Servers:** The servers are managed and monitored by a cloud service provider to accommodate a service of data storage and have significant and unlimited storage space and computation resources. In the cloud storage service, the clients can store their data into a set of servers in a simultaneous and distributed manner.
- **TPA:** This entity is delegated the responsibility of checking the servers on behalf of the clients. The TPA is assumed to be semi-trusted (trusted in checking the servers periodically, and untrusted in learning the secret keys of the clients).

Originally, the system model which consists of only the client and the servers without the TPA is enough for data check. To enable the public authentication feature, the TPA is employed with the assumption that the TPA is a honest-but-curious entity. Several previous papers also use the same assumption of the TPA, e.g, [62, 68–70, 74, 75].

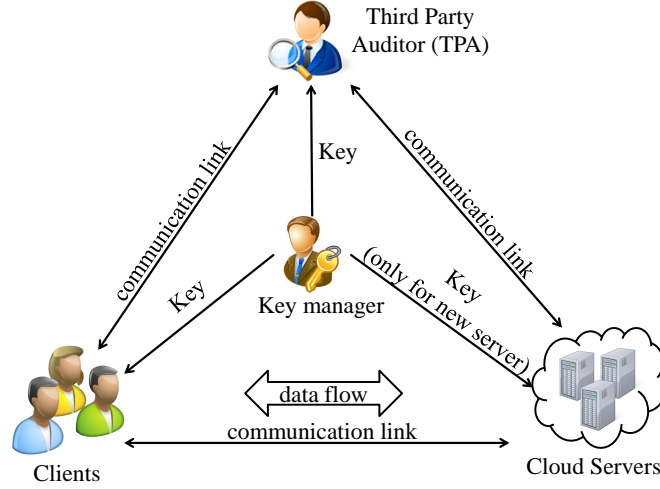


Figure 4.1: System model of the MD-POR scheme

To be suitable for our system model, we modify the POR such that the verifier is the TPA and there are multiple clients as follows:

1. **keygen**(1^λ): The key manager runs this algorithm which takes a security parameter λ as the input, and outputs a set of secret keys $\{sk_i\}_{i \in \{1, \dots, s\}}$ for s clients, a secret key κ for the TPA and a secret key κ' for a new server every when a repair phase is executed.
2. **encode**(sk_i, F_i): Each client i uses his secret key sk_i to encode his original file F_i to an encoded file F'_i , then sends F'_i to the servers. Each server then linearly combines all F'_i ($i \in \{1, \dots, s\}$) and stores the combined blocks of the files.
3. **check**(κ): The TPA uses his key κ to generate a challenge c and sends c to the servers. Each server then computes a response r and sends r back to the TPA. Finally, the TPA verifies whether each F_i is available and intact based on c and r .
4. **repair**(): This algorithm is executed when a failure is detected in the check phase. The technique of the repair phase depends on each specific scheme.

4.2 Adversarial Model

In this scheme, the key manager and the clients are trusted. The following entities are considered to be untrusted:

- Adversaries outside the system.
- The servers in the system.

- The TPA in the system.

There are two assumptions:

- The TPA is assumed to not collude with the servers.
- All the data and the keys are transmitted via a secure channel.

Concretely, the adversaries can perform the following the attacks:

Mobile Attack. This attack is also considered in several papers [26, 76–78]. This attack can be performed by an adversary outside the system who may potentially corrupt all the servers across the full system lifetime. A restriction for the adversary is that it can attack at most $(n - l)$ out of n servers in any given time step. Let *epoch* denote a given time step. In each epoch, the servers are checked and if a corruption in a certain server is detected, the data stored in that corrupted server will be repaired from redundancy in the intact servers. Without the server checks, the adversary can corrupt all n the servers and can destroy the system in $n/(n - l)$ epochs.

Curious Adversary. This attack can be performed by the untrusted entities in order to search the secret keys of the clients. This attack is a specific attack in our scheme. Namely, the attackers can behave as follows:

- In order to check the data stored in each server during the check phase, the TPA is given a key which is constructed from the secret keys of the clients. Given the key, the TPA may try to search the secret keys of the clients using the brute force search.
- In order to check the data provided from other servers during the repair phase, the new server is also given another key which is also constructed from the secret keys of the clients. Given the key, the new server may try to search the secret keys of the clients using the brute force search.
- Anyone who has the access to the keygen algorithm may also try to search these secret keys of the clients.

Response Forgery. This forgery is performed by the servers. This attack is a specific attack in our scheme. Namely, the forgery happens as follows:

- The TPA requires each server to provide a response to ensure the data stored in that server is healthy.
- Each server must respond a valid pair of the aggregated coded block (says, c) and the aggregated tag (says, t) to the TPA in order to demonstrate that the server is healthy. However, the malicious server responds $\{c_{forge}, t_{forge}\}$ instead of $\{c, t\}$ to the TPA. If $\{c_{forge}, t_{forge}\}$ holds the verification, that server can pass the check phase.

Pollution Attack. This attack is performed by the servers. This attack is also considered in several papers [79, 82, 86–92, 137]. The purpose of this attack is to break the linear independence of the encoded blocks by injecting invalid packets to prevent data recover. In the network scenario, a malicious node may forward invalid package to the receiver node. Therefore, when the receiver node obtain multiple packets, the receiver node cannot tell which of the received packets are corrupt. In the distributed storage system scenario, this attack happens when the malicious server provides a valid response to pass the check phase, but then provides an invalid response during the repair phase. An example is given as follows:

- **Encode:** the client encodes the augmented blocks (w_1, w_2, w_3) to six coded blocks: c_{11}, c_{12} (stored in the server \mathcal{S}_1), c_{21}, c_{22} (stored in the server \mathcal{S}_2), and c_{31}, c_{32} (stored in the server \mathcal{S}_3). Suppose that \mathcal{S}_1 is malicious and will perform a pollution attack.
- **Check:** suppose that \mathcal{S}_3 is detected as a corrupted server.
- **Repair:** \mathcal{S}_3 should be repaired by two coded blocks: c'_{31} (which is a linear combination of c_{11} and c_{12}) and c'_{32} (which is a linear combination of c_{21} and c_{22}). However, \mathcal{S}_1 is not detected because this time is the repair phase, not the check phase. The client still thinks \mathcal{S}_1 is healthy, and thus the client requests the coded blocks from \mathcal{S}_1 and \mathcal{S}_2 . \mathcal{S}_1 will provide an invalid coded block c''_{31} to the client instead of c'_{31} .

Because the valid coded blocks c'_{31} is not a linear combination of (w_1, w_2, w_3) , the original file cannot be recovered from any $m = 3$ coded blocks.

4.3 Proposed MD-POR Scheme

Before describing the MD-POR scheme in detail, we firstly introduce the technical roadmap, the key idea, the notations and the structure as follows:

- *Technical Roadmap.* We depict the technical roadmap in Figure 4.2. The input is the file blocks. Firstly, the file blocks are used to generate the augmented blocks. Then, the augmented blocks are combined with random values to compute the keys. Meanwhile, the augmented blocks are linearly combined into the coded blocks using the network coding. Finally, the coded blocks are tagged using the keys. The coded blocks and the tags are the outputs. The network coding is used because it is related to the data repair. The inter MAC is used because it is related to the multi-user, direct repair and public authentication features. The network coding and inter MAC are suitable to combine together in the MD-POR scheme.

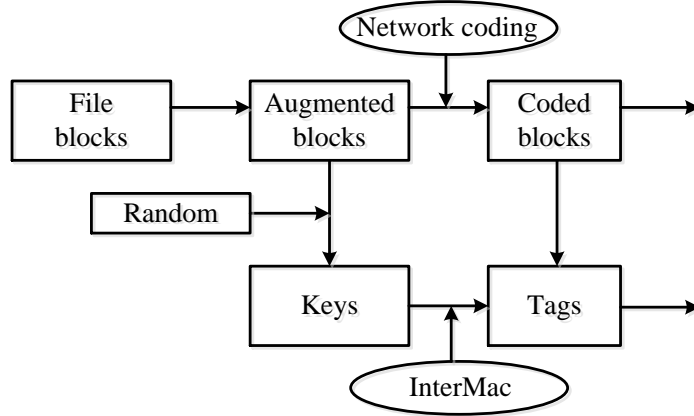


Figure 4.2: Technical roadmap

- *Key Idea.* In our scheme, multiple clients are simultaneously supported. Each client owns a different secret key. The data of each client cannot be checked alone; instead, each client uses his/her secret key to compute additional information which is MAC tag for each augmented block. Each client then transmits the aggregated augmented block and the corresponding tag to the servers. The servers will linearly combine the aggregated augmented blocks and the aggregated tags. *Herein lies a challenge that how to enable the TPA to check the servers during the check phase without any information of the secret keys of the clients; and how to enable the new server to check the other servers during the repair phase without any information of the secret keys of the clients.* The traditional MACs are inadequate to solve this task. Some recent papers related to this problem have been proposed (e.g., [82–84]). However, as mentioned in Section 2.1, they all use an asymmetric key setting. The inter MAC technique is a suitable technique to generate such secret keys for multiple clients. Using this technique, we can generate the keys for the system as follows:
 - The key of a client is generated such that it is orthogonal to all the augmented blocks which do not belong to that client.
 - The key of the TPA is generated such that it is the summation of the secret keys of the clients. The TPA can check the servers during the check phase without the information of the secret keys.
 - The key of the new server is generated such that it is the summation between the key of the TPA and an additional key which is orthogonal to all augmented blocks of all the clients. The new server can also check the other servers used in the repair phase without the information of the secret keys.
- *Notations.* The notations used throughout the MD-POR scheme are given in Table 4.1.

Table 4.1: List of notations in the MD-POR scheme

Notation	Description
s	number of clients
i	client index ($i \in \{1, \dots, s\}$)
\mathcal{C}_i	client ($i \in \{1, \dots, s\}$)
k_i	secret key of \mathcal{C}_i
F_i	original file of \mathcal{C}_i
g	number of file blocks in F_i of each client (g is the same for all clients)
j	file block index ($j \in \{1, \dots, g\}$)
v_{ij}	file block ($i \in \{1, \dots, s\}, j \in \{1, \dots, g\}$)
w_{ij}	augmented block of v_{ij} ($i \in \{1, \dots, s\}, j \in \{1, \dots, g\}$)
\mathbb{F}_q^ξ	a ξ -dimensional finite field \mathbb{F}_q of a prime order q .
m	$m = s \cdot g$
n	number of servers
l	number of healthy servers which are used to repair a corrupted server during repair phase
d	number of coded blocks in each server
x	server index ($x \in \{1, \dots, n\}$)
y	coded block index in each server ($y \in \{1, \dots, d\}$)
\mathcal{S}_x	server ($x \in \{1, \dots, n\}$)
c_{xy}	coded block ($x \in \{1, \dots, n\}, y \in \{1, \dots, d\}$)
t_{xy}	tag of c_{xy} ($x \in \{1, \dots, n\}, y \in \{1, \dots, d\}$)
κ	key of the TPA
κ'	key of a new server
\mathcal{A}	adversary
$spotcheck$	number of coded blocks in a server which are checked during the check and repair phases ($spotcheck \in \{1, \dots, d\}$)

- *Structure.* Let $\mathcal{C}_1, \dots, \mathcal{C}_s$ denote the set of s clients. Each client \mathcal{C}_i where $i \in \{1, \dots, s\}$ owns a file $F_i = v_{i1} || \dots || v_{ig}$ where g is the number of file blocks. Suppose that for all the clients, g is the same and each file block $v_{ij} \in \mathbb{F}_q^\xi$ where $j \in \{1, \dots, g\}$. \mathcal{C}_i creates g augmented blocks $\{w_{i1}, \dots, w_{ig}\}$ from g file blocks $\{v_{i1}, \dots, v_{ig}\}$. Each augmented block w_{ij} has the following form:

$$w_{ij} = (v_{ij}, \underbrace{0, \dots, 0}_{g(i-1)}, \underbrace{0, \dots, 0, 1, 0, \dots, 0}_g, \underbrace{0, \dots, 0}_{g(s-i)}) \in \mathbb{F}_q^{\xi+m} \quad (4.1)$$

$m = s \cdot g$

where $i \in \{1, \dots, s\}, j \in \{1, \dots, g\}, m = s \cdot g$. Each client \mathcal{C}_i is given a secret key k_i by the key manager. \mathcal{C}_i uses his secret key k_i to compute a tag t_{ij} for each augmented blocks w_{ij} . The augmented blocks and the tags are then linearly combined and

transmitted to all the servers via a secure channel. The TPA will check each server every epoch. Each server must linearly combine its coded blocks and the tags. Each server then responds the aggregated coded block and the aggregated tag to the TPA. The TPA finally verifies each server based on the response even though the TPA does not know any secret key k_i for all $i \in \{1, \dots, s\}$.

We are now ready to describe the proposed scheme via each phase of the POR (KeyGen, Encode, Check, Repair).

4.3.1 Keygen

a) Keys for the clients (KeyGen1). The key manager generates a set of s secret keys $\{k_1, \dots, k_s\}$ for s clients. Each key k_p of the client \mathcal{C}_p where $p \in \{1, \dots, s\}$ is constructed in a way that k_p is orthogonal to all the augmented blocks which do not belong to \mathcal{C}_p . In a formal statement, k_p is constructed such that:

$$\forall i \in \{1, \dots, s\}, i \neq p, p \in \{1, \dots, s\}, w_{ij} \cdot k_p = 0 \quad (4.2)$$

Concretely, the key manager will perform the following algorithm:

- **KeyGen1**(w_{11}, \dots, w_{sg}) $\rightarrow \{k_1, \dots, k_s\}$:
 - For each $p \in \{1, \dots, s\}$, compute $k_p \in \mathbb{F}_q^{\xi+m}$:
$$k_p \leftarrow \text{OrthogonalGen-MS}(p, \{w_{ij} \in \mathbb{F}_q^{\xi+m} | i = 1, \dots, s; i \neq p; j = 1, \dots, g\}) \quad (4.3)$$
 - After the set $\{k_1, \dots, k_s\}$ is computed, assign $k_p \in \mathbb{F}_q^{\xi+m}$ to the client \mathcal{C}_p as his/her secret key via a secure channel.

Each client will use his/her secret keys to compute the tags for his/her own augmented blocks.

b) Key for the TPA (KeyGen2). The key manager will perform the following algorithm:

- **KeyGen2**(k_1, \dots, k_s) $\rightarrow \kappa$:
 - Compute the key:

$$\kappa = k_1 + \dots + k_s \in \mathbb{F}_q^{\xi+m} \quad (4.4)$$
 - Assign κ to the TPA via a secure channel.

The TPA will use κ to verify the servers during the check phase. We can see that the TPA is only given the sum κ without each component k_i where $i = \{1, \dots, s\}$. We will prove the security in Section 4.5.

c) One-time key for a new server (KeyGen3). When the repair phase is executed, the key manager will compute a key κ' which is a summation between the key of the TPA (κ) and another key k_{repair} . The new server will use κ' to check pollution attack during the repair phase. κ is already computed in **KeyGen2** as a static key. k_{repair} must be re-computed every repair time in order to ensure that an adversary cannot attack the new server to obtain k_{repair} for passing the pollution attack check in the later repair phases (We thereafter explain it in Section 4.5.4). k_{repair} is constructed such that it is orthogonal to all augmented blocks of all the clients. Namely, the key manager will perform the following algorithm:

- **KeyGen3**($\kappa, \{w_{11}, \dots, w_{sg}\}$) $\rightarrow \kappa'$:
 - Compute $k_{repair} \leftarrow \text{OrthogonalGen-New}(w_{11}, \dots, w_{sg})$. $k_{repair} \in \mathbb{F}_q^{\xi+m}$
 - Compute the key:

$$\kappa' = \kappa + k_{repair} = (k_1 + \dots + k_s) + k_{repair} \in \mathbb{F}_q^{\xi+m} \quad (4.5)$$
 - Assign $\kappa' \in \mathbb{F}_q^{\xi+m}$ to the new server when a repair phase is executed.
- **OrthogonalGen-New**(w_{11}, \dots, w_{sg}) $\rightarrow k_{repair}$: this is the sub-algorithm used in the **KeyGen3** algorithm:
 - Find the span π of $\{w_{11}, \dots, w_{sg}\}$. Each $w_{ij} \in \mathbb{F}_q^{\xi+m}$.
 - Construct the matrix M in which w_{11}, \dots, w_{sg} are the rows of M .
 - Find the null-space of M , denoted by π_M^\perp , which is the set of all vectors $u \in \mathbb{F}_q^{\xi+m}$ such that $M \cdot u^T = 0$.
 - Find the basis vectors of π_M^\perp , denoted by $B_1, \dots, B_\xi \in \mathbb{F}_q^{\xi+m}$ // Theorem 5 will explain why the number of the basis vectors is ξ .
 - Compute $k_{repair} \leftarrow \text{Kg-New}(B_1, \dots, B_\xi)$.
- **Kg-New**(B_1, \dots, B_ξ) $\rightarrow k_{repair}$: this is the sub-algorithm which is used in the **OrthogonalGen-New** algorithm.
 - Let f be a pseudo-random function such that $\mathcal{K} \times [1, \xi] \rightarrow \mathbb{F}_q$.
 - Generate $r_x \leftarrow f(k_{PRF}, x) \in \mathbb{F}_q, \forall x \in \{1, \dots, \xi\}$ where $k_{PRF} \in \mathcal{K}$.
 - Compute $k_{repair} \leftarrow \sum_{x=1}^{\xi} r_x \cdot B_x \in \mathbb{F}_q^{\xi+m}$.

Theorem 5. Given $\{w_{11}, \dots, w_{sg}\}$ where each $w_{ij} \in \mathbb{F}_q^{\xi+m}$, the number of basis vectors of π_M^\perp is ξ .

Proof. $\text{rank}(M) = s \cdot g = m$. Let π_M be the space spanned by the rows of M . For any $m \times (\xi + m)$ matrix, the rank-nullity theorem gives:

$$\text{rank}(M) + \text{nullity}(M) = \xi + m \quad (4.6)$$

where $\text{nullity}(M)$ is the dimension of π_M^\perp . Thus, we have:

$$\dim(\pi_M^\perp) = (\xi + m) - m = \xi \quad (4.7)$$

Therefore, the number of basis vectors of π_M^\perp is ξ . In the **OrthogonalGen-New** algorithm, we denoted the basis vectors by B_1, \dots, B_ξ . \square

Note that when k_{repair} is constructed in the first time, the basis vectors B_1, \dots, B_ξ are computed and saved for reusing in the later times. In the next repair times, the basis vectors will be re-used for computation cost-effective, and only the random coefficient r_x is re-generated again to compute k_{repair} . The **KeyGen3** algorithm is only executed and κ' is given to a new server if only if a repair phase is executed. The key κ is already computed in the **KeyGen1** algorithm as a static information, only k_{repair} is different each repair time.

4.3.2 Encode

1. Each client \mathcal{C}_i where $i \in \{1, \dots, s\}$ computes g tags for g augmented blocks as follows:

For $\forall i \in \{1, \dots, s\}, \forall j \in \{1, \dots, g\}$:

$$t_{ij} = w_{ij} \cdot k_i \in \mathbb{F}_q \quad (4.8)$$

2. Each client \mathcal{C}_i linearly combines the augmented blocks and the corresponding tags as follows:

For $\forall i \in \{1, \dots, s\}$:

- \mathcal{C}_i generates g coefficients: $\alpha_{ij} \xleftarrow{\text{rand}} \mathbb{F}_q$ for all $j \in \{1, \dots, g\}$.
- \mathcal{C}_i computes coded block:

$$w_{\mathcal{C}_i} = \sum_{j=1}^g \alpha_{ij} \cdot w_{ij} \in \mathbb{F}_q^{\xi+m} \quad (4.9)$$

- \mathcal{C}_i computes tag:

$$t_{\mathcal{C}_i} = \sum_{j=1}^g \alpha_{ij} \cdot t_{ij} \in \mathbb{F}_q \quad (4.10)$$

- \mathcal{C}_i sends the pair of $\{w_{\mathcal{C}_i}, t_{\mathcal{C}_i}\}$ to all n servers $\{\mathcal{S}_1, \dots, \mathcal{S}_n\}$.

3. Each server \mathcal{S}_x where $x \in \{1, \dots, n\}$ creates d pairs of coded block c_{xy} and corresponding tag t_{xy} where $y \in \{1, \dots, d\}$ as follows:

For $\forall x \in \{1, \dots, n\}, \forall y \in \{1, \dots, d\}$:

- \mathcal{S}_x generates s coefficients: $\beta_{xyi} \xleftarrow{\text{rand}} \mathbb{F}_q$ for all $i \in \{1, \dots, s\}$.

- \mathcal{S}_x computes coded block:

$$c_{xy} = \sum_{i=1}^s \beta_{xyi} \cdot w_{c_i} \in \mathbb{F}_q^{\xi+m} \quad (4.11)$$

- \mathcal{S}_x computes tag:

$$t_{xy} = \sum_{i=1}^s \beta_{xyi} \cdot t_{c_i} \in \mathbb{F}_q \quad (4.12)$$

4.3.3 Check

The TPA challenges each server. Each server must provide its corresponding proof to the TPA. The TPA then uses its key κ to check whether the server is healthy or not based on the proof. The TPA has responsibility to check all the n servers periodically.

1. The TPA challenges each server:

- The TPA generates a challenge *chall* which consists of *spotcheck* pairs of index and coefficient: $chall = \{(y_1, \gamma_1), \dots, (y_{spotcheck}, \gamma_{spotcheck})\}$ where $y_{sp} \xleftarrow{rand} \{1, \dots, d\}$ and $\gamma_{sp} \xleftarrow{rand} \mathbb{F}_q$ for $sp \in \{1, \dots, spotcheck\}$.
- The TPA sends *chall* to all the servers.

2. Each server \mathcal{S}_x where $x \in \{1, \dots, n\}$ provides its proof as follows:

- \mathcal{S}_x combines coded blocks:

$$c_x = \sum_{sp=1}^{spotcheck} \gamma_{sp} \cdot c_{xy_{sp}} \in \mathbb{F}_q^{\xi+m} \quad (4.13)$$

- \mathcal{S}_x combines tags:

$$t_x = \sum_{sp=1}^{spotcheck} \gamma_{sp} \cdot t_{xy_{sp}} \in \mathbb{F}_q \quad (4.14)$$

- \mathcal{S}_x sends $\{c_x, t_x\}$ to the TPA.

3. The TPA verifies \mathcal{S}_x as follows:

- TPA computes:

$$t'_x = c_x \cdot \kappa \in \mathbb{F}_q \quad (4.15)$$

- TPA verifies iff:

$$t_x = t'_x \quad (4.16)$$

If the equality holds, the TPA will return **true** (this means that \mathcal{S}_x is healthy), otherwise the TPA will return **false** (this means that \mathcal{S}_x is corrupted).

4.3.4 Repair

Suppose that the server \mathcal{S}_r is detected as a corrupted server in the check phase. \mathcal{S}_r will be replaced by a new server \mathcal{S}'_r . The new server \mathcal{S}'_r challenges and requires l healthy servers $\mathcal{S}_{x_1}, \dots, \mathcal{S}_{x_l}$ to provide the aggregated coded blocks and the aggregated tags. \mathcal{S}'_r will check each of these l servers using the key κ' , which is generated by the **KeyGen3** algorithm. We will explain how to choose l in Section 4.5.

1. The new server \mathcal{S}'_r challenges l healthy servers:

- \mathcal{S}'_r generates a challenge *chall* which consists of *spotcheck* pairs of index and co-efficient: $chall = \{(y_1, \gamma_1), \dots, (y_{spotcheck}, \gamma_{spotcheck})\}$ where $y_{sp} \xleftarrow{rand} \{1, \dots, d\}$ and $\gamma_{sp} \xleftarrow{rand} \mathbb{F}_q$ for $sp \in \{1, \dots, spotcheck\}$.
- \mathcal{S}'_r sends *chall* to l healthy servers.

2. Each server \mathcal{S}_x where $x \in \{x_1, \dots, x_l\}$ linearly combines its *spotcheck* coded blocks and linearly combines its *spotcheck* tags as follows:

For $\forall x \in \{x_1, \dots, x_l\}$:

- \mathcal{S}_x combines coded blocks:

$$c_x = \sum_{sp=1}^{spotcheck} \gamma_{sp} \cdot c_{xy_{sp}} \in \mathbb{F}_q^{\xi+m} \quad (4.17)$$

- \mathcal{S}_x combines tags:

$$t_x = \sum_{sp=1}^{spotcheck} \gamma_{sp} \cdot t_{xy_{sp}} \in \mathbb{F}_q \quad (4.18)$$

- \mathcal{S}_x sends $\{c_x, t_x\}$ to \mathcal{S}'_r .

3. The new server \mathcal{S}'_r checks whether each server \mathcal{S}_x where $x \in \{x_1, \dots, x_l\}$ provides a valid packet (pollution attack), using the key $\kappa' = (k_1 + \dots + k_s) + k_{repair}$ as follows:

- \mathcal{S}'_r computes:

$$t'_x = c_x \cdot \kappa' \in \mathbb{F}_q \quad (4.19)$$

- \mathcal{S}'_r checks iff:

$$t_x = t'_x \quad (4.20)$$

4. The new server \mathcal{S}'_r computes d coded blocks and d tags for itself as follows:

For $\forall y \in \{1, \dots, d\}$:

- \mathcal{S}'_r generates l coefficients $\theta_{xy} \xleftarrow{rand} \mathbb{F}_q$ for all $x \in \{x_1, \dots, x_l\}$.

- \mathcal{S}'_r computes new coded blocks:

$$c_{ry} = \sum_{x=x_1}^{x_l} \theta_{xy} \cdot c_x \in \mathbb{F}_q^{\ell+m} \quad (4.21)$$

- \mathcal{S}'_r computes new tags:

$$t_{ry} = \sum_{x=x_1}^{x_l} \theta_{xy} \cdot t_x \in \mathbb{F}_q \quad (4.22)$$

4.4 Correctness

1. The correctness of Equation 4.16 is proven as follows:

Proof.

$$\begin{aligned}
t_x &= \sum_{sp=1}^{spotcheck} \gamma_{sp} \cdot t_{xy_{sp}} // \text{ because of Equation 4.14} \\
&= \sum_{sp=1}^{spotcheck} \gamma_{sp} (\sum_{i=1}^s \beta_{xy_{sp}i} t_{c_i}) // \text{ because of Equation 4.12} \\
&= \sum_{sp=1}^{spotcheck} \sum_{i=1}^s \gamma_{sp} \beta_{xy_{sp}i} t_{c_i} \\
&= \sum_{sp=1}^{spotcheck} \sum_{i=1}^s \gamma_{sp} \beta_{xy_{sp}i} (\sum_{j=1}^g \alpha_{ij} t_{ij}) // \text{ because of Equation 4.10} \\
&= \sum_{sp=1}^{spotcheck} \sum_{i=1}^s \sum_{j=1}^g \gamma_{sp} \beta_{xy_{sp}i} \alpha_{ij} t_{ij} \\
&= \sum_{sp=1}^{spotcheck} \sum_{i=1}^s \sum_{j=1}^g \gamma_{sp} \beta_{xy_{sp}i} \alpha_{ij} (w_{ij} k_i) // \text{ because of Equation 4.8} \\
&= \sum_{sp=1}^{spotcheck} \sum_{i=1}^s \sum_{j=1}^g \gamma_{sp} \beta_{xy_{sp}i} \alpha_{ij} w_{ij} k_i \\
t'_x &= c_x \cdot \kappa // \text{ because of Equation 4.15} \\
&= c_x \cdot (k_1 + \dots + k_s) // \text{ because of Equation 4.4} \\
&= \sum_{sp=1}^{spotcheck} \gamma_{sp} c_{xy_{sp}} (k_1 + \dots + k_s) // \text{ because of Equation 4.13} \\
&= \sum_{sp=1}^{spotcheck} \gamma_{sp} (\sum_{i=1}^s \beta_{xy_{sp}i} w_{c_i}) (k_1 + \dots + k_s) // \text{ because of Equation 4.11} \\
&= \sum_{sp=1}^{spotcheck} \sum_{i=1}^s \gamma_{sp} \beta_{xy_{sp}i} w_{c_i} (k_1 + \dots + k_s) \\
&= \sum_{sp=1}^{spotcheck} \sum_{i=1}^s \gamma_{sp} \beta_{xy_{sp}i} (\sum_{j=1}^g \alpha_{ij} w_{ij}) (k_1 + \dots + k_s) // \text{ because of Equation 4.9} \\
&= \sum_{sp=1}^{spotcheck} \sum_{i=1}^s \sum_{j=1}^g \gamma_{sp} \beta_{xy_{sp}i} \alpha_{ij} w_{ij} (k_1 + \dots + k_s)
\end{aligned}$$

Because k_1, \dots, k_s are constructed such that $w_{ij} \cdot k_p = 0$ for all $i, p \in \{1, \dots, s\}$ and $i \neq p$ using the **KeyGen1** algorithm, then we have:

$$\begin{aligned}
t'_x &= \sum_{sp=1}^{spotcheck} \sum_{i=1}^s \sum_{j=1}^g \gamma_{sp} \beta_{xy_{sp}i} \alpha_{ij} w_{ij} k_i \\
&= t_x
\end{aligned}$$

This completes the proof. \square

2. The correctness of Equation 4.20 is proven as follows:

Proof. The way to prove the correctness of Equation 4.20 is similar to the correctness of Equation 4.16 in the check phase. The only different thing is that in Equation 4.16, not only k_1, \dots, k_s but also k_{repair} participates in linearly combining the coded blocks and the tags. Concretely:

$$\begin{aligned}
t_x &= \sum_{sp=1}^{spotcheck} \gamma_{sp} \cdot t_{xy_{sp}} // \text{ because of Equation 4.18} \\
&= \sum_{sp=1}^{spotcheck} \gamma_{sp} (\sum_{i=1}^s \beta_{xy_{sp}i} t_{c_i}) // \text{ because of Equation 4.12} \\
&= \sum_{sp=1}^{spotcheck} \sum_{i=1}^s \gamma_{sp} \beta_{xy_{sp}i} t_{c_i} \\
&= \sum_{sp=1}^{spotcheck} \sum_{i=1}^s \gamma_{sp} \beta_{xy_{sp}i} (\sum_{j=1}^g \alpha_{ij} t_{ij}) // \text{ because of Equation 4.10} \\
&= \sum_{sp=1}^{spotcheck} \sum_{i=1}^s \sum_{j=1}^g \gamma_{sp} \beta_{xy_{sp}i} \alpha_{ij} t_{ij} \\
&= \sum_{sp=1}^{spotcheck} \sum_{i=1}^s \sum_{j=1}^g \gamma_{sp} \beta_{xy_{sp}i} \alpha_{ij} (w_{ij} k_i) // \text{ because of Equation 4.8} \\
&= \sum_{sp=1}^{spotcheck} \sum_{i=1}^s \sum_{j=1}^g \gamma_{sp} \beta_{xy_{sp}i} \alpha_{ij} w_{ij} k_i \\
t'_x &= c_x \cdot \kappa' // \text{ because of Equation 4.19} \\
&= c_x (k_1 + \dots + k_s + k_{repair}) // \text{ because of Equation 4.5} \\
&= \sum_{sp=1}^{spotcheck} \gamma_{sp} c_{xy_{sp}} (k_1 + \dots + k_s + k_{repair}) // \text{ because of Equation 4.17} \\
&= \sum_{sp=1}^{spotcheck} \gamma_{sp} (\sum_{i=1}^s \beta_{xy_{sp}i} w_{c_i}) (k_1 + \dots + k_s + k_{repair}) // \text{ because of Equation 4.11} \\
&= \sum_{sp=1}^{spotcheck} \sum_{i=1}^s \gamma_{sp} \beta_{xy_{sp}i} w_{c_i} (k_1 + \dots + k_s + k_{repair}) \\
&= \sum_{sp=1}^{spotcheck} \sum_{i=1}^s \gamma_{sp} \beta_{xy_{sp}i} (\sum_{j=1}^g \alpha_{ij} w_{ij}) (k_1 + \dots + k_s + k_{repair}) // \text{ because of Equation 4.9} \\
&= \sum_{sp=1}^{spotcheck} \sum_{i=1}^s \sum_{j=1}^g \gamma_{sp} \beta_{xy_{sp}i} \alpha_{ij} w_{ij} (k_1 + \dots + k_s + k_{repair})
\end{aligned}$$

Because k_1, \dots, k_s are constructed such that $k_p \cdot w_{ij} = 0$ for all $i, p \in \{1, \dots, s\}$ and $i \neq p$ using the **KeyGen1** algorithm, then we have:

$$t'_x = \sum_{sp=1}^{spotcheck} \sum_{i=1}^s \sum_{j=1}^g \gamma_{sp} \beta_{xy_{sp}i} \alpha_{ij} w_{ij} (k_i + k_{repair})$$

Because k_{repair} is constructed such that $w_{ij} \cdot k_{repair}$ for all $i \in \{1, \dots, s\}$ and for all $j \in \{1, \dots, g\}$ using the **KeyGen3** algorithm, then we have:

$$\begin{aligned}
t'_x &= \sum_{sp=1}^{spotcheck} \sum_{i=1}^s \sum_{j=1}^g \gamma_{sp} \beta_{xy_{sp}i} \alpha_{ij} w_{ij} k_i \\
&= t_x
\end{aligned}$$

This completes the proof. \square

4.5 Security Analysis

4.5.1 Mobile Attack

To prevent the mobile attack, a data repair condition is given as follows:

Theorem 6. *The original files F_1, \dots, F_s of the clients can be reconstructed if in any epoch, at least l out of n servers collectively store $m = s \cdot g$ coded blocks which are linearly independent combinations of m original file blocks; and the matrix consisting of the accumulated coefficients has full rank (i.e., rank m).*

Proof. Each server \mathcal{S}_x where $x \in \{1, \dots, n\}$ contains d coded blocks c_{xy} where $y \in \{1, \dots, d\}$. Each coded block c_{xy} is computed from $m = s \cdot g$ augmented blocks w_{ij} where $i \in \{1, \dots, s\}, j \in \{1, \dots, g\}$ using the linear combination $c_{xy} = \sum_{i=1}^s \sum_{j=1}^g \beta_{xyi} \alpha_{ij} w_{ij}$. To reconstruct the original files, m augmented blocks $w_{11}, \dots, w_{1g}, \dots, w_{s1}, \dots, w_{sg}$ are viewed as the variables that need to be solved. To solve these m variables, at least m coded blocks are required to make the coefficient matrix has full rank because the number

of variables in an equation system must be less than or equal to the number of independent equations.

$$\begin{cases} c_{xy_1} = \sum_{i=1}^s \sum_{j=1}^g \beta_{xy_{i_1}} \cdot \alpha_{ij_1} \cdot w_{ij} \\ c_{xy_2} = \sum_{i=1}^s \sum_{j=1}^g \beta_{xy_{i_2}} \cdot \alpha_{ij_2} \cdot w_{ij} \\ \dots \\ c_{xy_m} = \sum_{i=1}^s \sum_{j=1}^g \beta_{xy_{i_m}} \cdot \alpha_{ij_m} \cdot w_{ij} \end{cases} \quad (4.23)$$

Therefore, at least l servers which collectively store $m = s \cdot g$ coded blocks in each epoch are required. $\lceil \frac{m}{d} \rceil \leq l < n$. \square

4.5.2 Curious Attack

The following theorems describes the probabilities for the adversaries to search the secret keys of the clients.

Theorem 7. *Given the key κ , the TPA cannot derive the secret keys of the clients k_1, \dots, k_s via the brute force search.*

Proof. The TPA is given the key κ to verify n servers $\mathcal{S}_1, \dots, \mathcal{S}_n$ during the check phase. Because $\kappa = k_1 + \dots + k_s$ as computed in the **KeyGen2** algorithm, the security problem is now the problem of solving s variables k_1, \dots, k_s given a single equation. The brute force search to solve these variables is to try all possible variable sets, and test whether the sets satisfy the equation by using the trial-and-error method. Because each key $k_i \in \mathbb{F}_q^{\xi+m}$ where $i \in \{1, \dots, s\}$, the probability to search k_i is $\frac{1}{q^{\xi+m}}$. The probability to search $(s-1)$ keys is $\frac{1}{q^{(\xi+m)(s-1)}}$. Given κ , the TPA can search $(s-1)$ keys and then obtains the last key by subtracting the $(s-1)$ keys from κ . Therefore, the probability for the TPA to solve all s keys (k_1, \dots, k_s) is $\frac{1}{q^{(\xi+m)(s-1)}}$. In a formal statement:

$$\Pr[\{k_1, \dots, k_s\} \leftarrow TPA(\kappa)] = \frac{1}{q^{(\xi+m)(s-1)}} \quad (4.24)$$

If q is chosen as a large prime (e.g., 160 bits), this probability is $\frac{1}{2^{160(\xi+m)(s-1)}}$, which is negligible. k_1, \dots, k_s and k_{repair} cannot be solved in a polynomial time. \square

Theorem 8. *Given the key κ' , the new server cannot derive the secret keys of the clients k_1, \dots, k_s via the brute force search.*

Proof. The way to prove this theorem is similar to the way to prove Theorem 7. Concretely, the new server \mathcal{S}'_r is given the key κ' to verify l servers $\mathcal{S}_{x_1}, \dots, \mathcal{S}_{x_l}$ during the repair phase. Because $\kappa' = (k_1 + \dots + k_s) + k_{repair}$ as computed in the **KeyGen3** algorithm, the security problem is now the problem of solving $(s+1)$ variables: k_1, \dots, k_s and k_{repair} given a single equation. The brute force search to solve these variables is to try all possible variable sets, and test whether the sets satisfy this equation by using the trial-and-error method. Because each key k_i and k_{repair} belong to $\mathbb{F}_q^{\xi+m}$, the probability to search k_i or k_{repair} is $\frac{1}{q^{\xi+m}}$. The probability to search s keys is $\frac{1}{q^{(\xi+m)s}}$. Given κ' , the new server \mathcal{S}'_r can

search s keys and then obtains the last key by subtracting the s keys from κ' . Therefore, the probability for the new server \mathcal{S}'_r to solve all $(s + 1)$ keys: k_1, \dots, k_s and k_{repair} is $\frac{1}{q^{(\xi+m)s}}$. In a formal statement:

$$\Pr[\{k_1, \dots, k_s, k_{repair}\} \leftarrow \mathcal{S}'_r(\kappa')] = \frac{1}{q^{(\xi+m)s}} \quad (4.25)$$

If q is chosen as a large prime (e.g., 160 bits), this probability is $\frac{1}{2^{160(\xi+m)s}}$, which is negligible. k_1, \dots, k_s and k_{repair} cannot be solved in a polynomial time. \square

Theorem 9. *The secret keys of the clients k_1, \dots, k_s cannot be derived by any entity who has an access to the KeyGen1 algorithm, which is used to compute the secret keys of the clients.*

Proof. In the KeyGen1 algorithm, each key k_p where $p \in \{1, \dots, s\}$ is computed as:

$$k_p \leftarrow \text{OrthogonalGen-MS}(p, \{w_{ij} \in \mathbb{F}_q^{\xi+m} | i = 1, \dots, s; i \neq p; j = 1, \dots, g\}). \quad (4.26)$$

More concretely, in the OrthogonalGen-MS algorithm, after finding the basis vectors $B_1, \dots, B_{\xi+g}$, k_p is computed as:

- $r_x \leftarrow f(k_{PRF}, x) \in \mathbb{F}_q, \forall x \in \{1, \dots, \xi + g\}$.
- $k_p \leftarrow \sum_{x=1}^{\xi+g} r_x \cdot B_x \in \mathbb{F}_q^{\xi+m}$.

where f is a pseudo-random function. The probability to find each r_x is $(\Pr[f] + \frac{1}{q})$. The probability to find all $r_1, \dots, r_{\xi+g}$ is $(\Pr[f] + \frac{1}{q^{\xi+g}})$. Note that it is not $((\xi + g)\Pr[f] + \frac{1}{q^{\xi+g}})$ because $\Pr[f]$ can be re-used for finding other r_i . This is also the probability to find one key k_q . The probability to find all s keys: k_1, \dots, k_s is $(\Pr[f] + \frac{1}{q^{(\xi+g)s}})$. Again, note that $\Pr[f]$ can be re-used for finding other keys; thus, it is not $(s\Pr[f] + \frac{1}{q^{(\xi+g)s}})$. In a formal statement:

$$\Pr[k_1, \dots, k_s \leftarrow \text{KeyGen1}] = \Pr[f] + \frac{1}{q^{(\xi+g)s}} \quad (4.27)$$

If the pseudo-random function is supposed to be unforgeable and q is chosen large enough (e.g., 160 bits), this probability is negligible. \square

4.5.3 Response Forgery

Suppose that a malicious server which performs this forgery is \mathcal{S}_x . In the check phase, instead of sending a pair of aggregated coded block and aggregated tag $\{c_x, t_x\}$ (as computed in Equation 4.13 and Equation 4.14) to the TPA, \mathcal{S}_x sends a pair of forged coded block and forged tag (c''_x, t''_x) to the TPA.

Theorem 10. *\mathcal{S}_x cannot pass the check phase with the response forgery.*

Proof. The purpose of \mathcal{S}_x is to generate (c''_x, t''_x) which holds the verification $t''_x = c''_x \cdot \kappa$. Because the TPA is assumed to not collude with any server and κ is sent to the TPA via a secure channel, a possible way for \mathcal{S}_x is to pass the verification is to obtain κ .

- Using the brute force search: the probability to find κ is $\frac{1}{q^{\xi+m}}$ because $\kappa \in \mathbb{F}_q^{\xi+m}$. Formally:

$$\Pr_{\text{BruteForce}}[\mathcal{S}_x \rightarrow \kappa] = \frac{1}{q^{\xi+m}} \quad (4.28)$$

- Using the access to the **KeyGen2** algorithm: because $\kappa = k_1 + \dots + k_s$, the problem to find κ now becomes the problem to find k_1, \dots, k_s via the **KeyGen1** algorithm. As Theorem 9, the probability to find k_1, \dots, k_s is $(\Pr[f] + \frac{1}{q^{(\xi+g)s}})$. Formally:

$$\Pr_{\text{KeyGen2}}[\mathcal{S}_x \rightarrow \kappa] = \Pr[f] + \frac{1}{q^{(\xi+g)s}} \quad (4.29)$$

From Equation 4.28, Equation 4.29, the probability for \mathcal{S}_x to pass the check phase is:

$$\begin{aligned} \Pr[\mathcal{S}_x \rightarrow \text{verify}(\text{CheckPhase}) = 1] &= \Pr_{\text{BruteForce}}[\mathcal{S}_x \rightarrow \kappa] + \Pr_{\text{KeyGen2}}[\mathcal{S}_x \rightarrow \kappa] \\ &= \frac{1}{q^{\xi+m}} + \Pr[f] + \frac{1}{q^{(\xi+g)s}} \end{aligned} \quad (4.30)$$

If the pseudo-random function f is unforgeable and q is chosen large enough (e.g., 160 bits), the probability for \mathcal{S}_x to pass the check phase is negligible. \square

4.5.4 Pollution Attack

In the check phase, the server \mathcal{S}_r is detected as a corrupted server. Then, a set of l servers $\mathcal{S}_{x_1}, \dots, \mathcal{S}_{x_l}$ are required to provide their responses which consist of aggregated coded blocks (as in Equation 4.17) and aggregated tags (as in Equation 4.18) to the new server \mathcal{S}'_r for repairing \mathcal{S}_r . Suppose that \mathcal{S}_x , which is a server in the set of the l servers $\mathcal{S}_{x_1}, \dots, \mathcal{S}_{x_l}$, is the malicious server which performs the pollution attack. Instead of sending the valid pair of aggregated coded block and aggregated tag $\{c_x, t_x\}$ to the new server \mathcal{S}'_r , \mathcal{S}_x sends a pair of forged coded block and forged tag (c''_x, t''_x) to the new server \mathcal{S}'_r .

Theorem 11. \mathcal{S}_x cannot pass the verification in the repair phase with the pollution attack.

Proof. The key idea here is that \mathcal{S}'_r always checks each aggregated coded block which is provided from each of the servers $\mathcal{S}_{x_1}, \dots, \mathcal{S}_{x_l}$. Although \mathcal{S}_x already passed the check phase, \mathcal{S}_x must be checked again by \mathcal{S}'_r in the repair phase before \mathcal{S}'_r uses the aggregated coded block of \mathcal{S}_x for repairing \mathcal{S}_r . Namely, we analyse the probability of \mathcal{S}_x as follows. (c''_x, t''_x) holds the verification $t''_x = c''_x \cdot \kappa'$ if \mathcal{S}_x can obtain κ' because the new server \mathcal{S}'_r is assumed to not collude with the other servers and κ' is sent to \mathcal{S}'_r via a secure channel.

- Using the brute force search: the probability to find κ' is $\frac{1}{q^{\xi+m}}$ because $\kappa' \in \mathbb{F}_q^{\xi+m}$. Formally:

$$\Pr_{\text{BruteForce}}[\mathcal{S}_x \rightarrow \kappa'] = \frac{1}{q^{\xi+m}} \quad (4.31)$$

- Using the access to the **KeyGen3** algorithm: because $\kappa' = k_1 + \dots + k_s + k_{repair}$, the problem to find κ' now becomes the problem to find k_1, \dots, k_s via the **KeyGen1** algorithm and to find k_{repair} in the **KeyGen3** algorithm. As Theorem 9, the probability to find k_1, \dots, k_s is $(\Pr[f] + \frac{1}{q^{(\xi+g)s}})$. Formally:

$$\Pr_{\text{KeyGen1}}[\mathcal{S}_x \rightarrow k_1, \dots, k_s] = \Pr[f] + \frac{1}{q^{(\xi+g)s}} \quad (4.32)$$

Now we find the probability to find k_{repair} as follows. In the **KeyGen3** algorithm, k_{repair} is computed as:

$$k_{repair} \leftarrow \text{OrthogonalGen-New}(w_{11}, \dots, w_{sg})$$

In the **OrthogonalGen-New** algorithm, after finding the basis vectors B_1, \dots, B_ξ , k_{repair} is computed as:

$$\begin{aligned} - r_x &\leftarrow f(k_{PRF}, x) \in \mathbb{F}_q, \forall x \in \{1, \dots, \xi\}. \\ - k_{repair} &\leftarrow \sum_{x=1}^{\xi} r_x \cdot B_x \in \mathbb{F}_q^{\xi+m}. \end{aligned}$$

where f is a pseudo-random function. The probability to find each r_x is $(\Pr[f] + \frac{1}{q})$. The probability to find all r_1, \dots, r_ξ is $(\Pr[f] + \frac{1}{q^\xi})$. It is not $(\xi\Pr[f] + \frac{1}{q^\xi})$ because $\Pr[f]$ can be re-used for finding other r_i . This is also the probability to find k_{repair} . Formally:

$$\Pr_{\text{KeyGen3}}[\mathcal{S}_x \rightarrow k_{repair}] = \Pr[f] + \frac{1}{q^\xi} \quad (4.33)$$

From Equation 4.32 and Equation 4.33, the probability for \mathcal{S}_x to find κ using the access to **KeyGen3** algorithm is as follows:

$$\begin{aligned} \Pr_{\text{KeyGen3}}[\mathcal{S}_x \rightarrow \kappa'] &= \Pr_{\text{KeyGen1}}[\mathcal{S}_x \rightarrow k_1, \dots, k_s] + \Pr_{\text{KeyGen3}}[\mathcal{S}_x \rightarrow k_{repair}] \\ &= 2\Pr[f] + \frac{1}{q^{(\xi+g)s}} + \frac{1}{q^\xi} \end{aligned} \quad (4.34)$$

From Equation 4.31 and Equation 4.34, the probability for \mathcal{S}_x to pass the verification of the new server \mathcal{S}'_r in the repair phase is as follows:

$$\begin{aligned} \Pr[\mathcal{S}_x \rightarrow \text{verify}(\text{RepairPhase}) = 1] &= \Pr_{\text{BruteForce}}[\mathcal{S}_x \rightarrow \kappa'] + \Pr_{\text{KeyGen3}}[\mathcal{S}_x \rightarrow \kappa'] \\ &= \frac{1}{q^{\xi+m}} + 2\Pr[f] + \frac{1}{q^{(\xi+g)s}} + \frac{1}{q^\xi} \end{aligned} \quad (4.35)$$

If the pseudo-random function f is unforgeable and q is chosen large enough (e.g., 160 bits), the probability for \mathcal{S}_x to pass the verification of the new server \mathcal{S}'_r in the repair phase is negligible. \square

We also consider that the new server \mathcal{S}'_r may become a malicious server. After repairing the corrupted server \mathcal{S}_r , the server \mathcal{S}'_r keeps the key κ' to perform the pollution attack in the next repair phase. However, because k_{repair} is re-computed every repair time as explained in the **KeyGen3** algorithm, \mathcal{S}'_r cannot re-use k_{repair} .

4.6 Efficiency Analysis

In Table 4.2, we compare the features and the efficiency between the MD-POR scheme and the previous schemes (the RDC-NC [61] and the NC-Audit [62] schemes). The RDC-NC and NC-Audit schemes are chosen for the comparison with our scheme because they have the same scenario as the MD-POR scheme at most. However, the different thing between our scheme and these two previous schemes is that these two schemes only consider a single client instead of multiple clients as our scheme. Therefore, for the fairness in the comparison, we assume that s clients can participate in the RDC-NC and NC-Audit schemes. However, these s clients in the RDC-NC and NC-Audit schemes can only perform in parallel instead of simultaneously combination as the MD-POR scheme. That parameter s in the RDC-NC and NC-Audit schemes does not affect the costs in the check phase and the repair phase because only one client can check and repair the servers. That s only affects the storage cost on server-side and the communication cost of the encode phase in the RDC-NC and NC-Audit schemes.

4.6.1 Storage Cost

Client-side. The storage costs on the client-side in the RDC-NC, NC-Audits and MD-POR schemes are given as follows:

- In the RDC-NC scheme, the client stores five secret keys in $\mathbb{F}_q^{\xi+g}$. Thus, the storage cost on the client-side is $5(\xi + g) \log_2 q$.
- In the NC-Audit scheme, the client stores only one secret key in $\mathbb{F}_q^{\xi+g}$. Thus, the storage cost on the client-side is $(\xi + g) \log_2 q$.
- In the MD-POR scheme, there are s keys for s clients. Each key belongs to $\mathbb{F}_q^{\xi+sg}$. Thus, the storage cost per client is $(\xi + sg) \log_2 q$.

Server-side. Because the file of each client is divided into g file block, the size of a file block is $|v_{ij}| = \frac{|F|}{g}$ where $i \in \{1, \dots, s\}$ and $j \in \{1, \dots, g\}$. As Equation 4.1, an augmented block has the following form:

$$w_{ij} = (v_{ij}, \underbrace{0, \dots, 0}_{g(i-1)}, \underbrace{0, \dots, 0, 1, 0, \dots, 0}_g, \underbrace{0, \dots, 0}_{g(s-i)}) \in \mathbb{F}_q^{\xi+m}$$

$\underbrace{\hspace{15em}}_{m=s \cdot g}$

The storage costs on the server-side in the RDC-NC, NC-Audit and MD-POR schemes are given as follows:

- In the RDC-NC scheme, the size of an augmented block is $|w_{ij}| = \frac{|F|}{g} + g$. The size of a coded block is $|c_{ij}| = |w_{ij}|$ because each coded block is a linear combination of all the augmented blocks. The number of servers is n . The number of coded blocks in each server is d . s clients are assumed to participate in the RDC-NC scheme in parallel. Therefore, the server storage on the server-side is $O(sdn(\frac{|F|}{g} + g))$.

Table 4.2: Efficiency comparison between the MD-POR and previous schemes

Feature		RDC-NC [61]	NC-Audit [62]	MD-POR
	Multi-client	no	no	yes
	Direct repair	no	not completed	yes
	Symmetric key	yes	yes	yes
	Public authentication	no	yes	yes
Storage complexity	Client-side	$5(\xi + g) \log_2 q$	$(\xi + g) \log_2 q$	$(\xi + sg) \log_2 q$
	Server-side	$O(sdn(\frac{ F }{g} + g))$	$O(sdn(\frac{ F }{g} + g))$	$O(dn(\frac{ F }{g} + sg))$
	TPA-side	N/A	$O((\xi + g + gdn) \log_2 q)$	$O((\xi + sg) \log_2 q)$
Encoding complexity	Computation (client)	$O(gdn)$	$O(gdn)$	$O(g)$
	Computation (server)	$O(1)$	$O(1)$	$O(sdn)$
	Computation (TPA)	N/A	$O(1)$	$O(1)$
	Communication	$O(sdn(\frac{ F }{g} + g))$	$O(sdn(\frac{ F }{g} + g) + sgdn)$	$O(ns(\frac{ F }{g} + sg))$
Checking complexity	Computation (client)	$O(n)$	$O(1)$	$O(1)$
	Computation (server)	$O(nd)$	$O(nd)$	$O(nd)$
	Computation (TPA)	N/A	$O(n)$	$O(n)$
	Communication	$O(n(\frac{ F }{g} + g))$	$O(n(\frac{ F }{g} + g))$	$O(n(\frac{ F }{g} + sg))$
Repairing complexity	Computation (client)	$O((l+1)d)$	$O(1)$	$O(1)$
	Computation (server)	$O(dl)$	$O(dl)$	$O(dl)$
	Computation (new server)	N/A	$O(dl)$	$O(dl)$
	Computation (TPA)	N/A	$O(l)$	$O(1)$
	Communication	$O((l+d)(\frac{ F }{g} + g))$	$O(l(\frac{ F }{g} + g) + ld)$	$O(l(\frac{ F }{g} + sg))$

- In the NC-Audit scheme, the analysis is the same as the analysis in the RDC-NC scheme. Thus, the storage cost on the server-side is also $O(sdn(\frac{|F|}{g} + g))$.
- In the MD-POR scheme, the size of an augmented block is $|w| = \frac{|F|}{g} + sg$. Similar to the RDC-NC and NC-Audit scheme, the size of a coded block is $|c_{ij}| = |w_{ij}|$ because each coded block is a linear combination of all the augmented blocks; the number of servers is n and the number of coded blocks in each server is d . However, s clients participate in our scheme simultaneously (not in parallel as the RDC-NC and NC-Audit schemes). Thus, the storage cost on the server-side in the MD-POR scheme is $O(dn(\frac{|F|}{g} + sg))$.

TPA-side. The storage costs on the TPA-side in the RDC-NC, NC-Audit and MD-POR schemes are given as follows:

- In the RDC-NC scheme, the TPA does not exist because this scheme does not support public authentication. The client must check the servers periodically. Therefore, the storage cost on the TPA-side is N/A, which means that it is not applicable due to the lack of support.
- In the NC-Audit scheme, the TPA not only stores a key in $\mathbb{F}_q^{\xi+g}$ which is used for verification ($O(\xi + g) \log_2 q$) but also stores the coding coefficients in \mathbb{F}_q which are used to compute all coded blocks ($O(gdn \log_2 q)$). Thus, the storage cost on the TPA-side is $O((\xi + g + gdn) \log_2 q)$.
- In the MD-POR scheme, the TPA stores the key $\kappa \in \mathbb{F}_q^{\xi+m}$ which is computed using the KeyGen2 algorithm. Because $m = s \cdot g$, we have $\kappa \in \mathbb{F}_q^{\xi+sg}$. Therefore, the storage cost on the TPA-side is $O((\xi + sg) \log_2 q)$.

4.6.2 Encode cost

Computation on Client-side. The computation costs on the client-side in the RDC-NC, NC-Audit and MD-POR schemes are given as follows:

- In the RDC-NC scheme, during the encode phase, each client combines g augmented blocks ($O(g)$) to create dn coded blocks in order to store d coded blocks in each of n servers. Thus, the computation cost on the client-side is $O(gdn)$.
- In the NC-Audit scheme, the encode phase is the same as the encode phase in the RDC-NC scheme. The computation cost on the client-side is also $O(gdn)$.
- In the MD-POR scheme, each client only needs to combine g augmented blocks ($O(g)$) and then distributes them to all the servers. The servers will create coded blocks by themselves. The cost in the MD-POR scheme is thus $O(g)$.

Computation on Server-side. The computation costs on the server-side in the RDC-NC, NC-Audit and MD-POR schemes are given as follows:

- In the RDC-NC scheme, the servers do not need to do anything. The servers only need to receive the coded blocks which are computed by the clients. Thus, the computation cost on the server-side is $O(1)$.
- In the NC-Audit scheme, the servers also do not need to do anything and only need to receive the coded blocks which are computed by the clients like the RDC-NC scheme. Thus, the computation cost on the server-side is also $O(1)$.
- In the MD-POR scheme, each of n servers must combine s coded blocks which are sent by the clients in order to compute d coded blocks for the server itself. Thus, the computation cost on the server-side is $O(sdn)$.

Computation on TPA-side. The computation costs on the TPA-side in the RDC-NC, NC-Audit and MD-POR schemes are given as follows:

- In the RDC-NC scheme, the TPA does not exist because the scheme does not support the public authentication. The client must check the servers periodically. Therefore, the storage cost on the TPA-side is N/A.
- In the NC-Audit scheme, the TPA does nothing during the encode phase. Thus, the computation cost on the TPA-side is $O(1)$.
- In the MD-POR scheme, the TPA also does nothing during the encode phase like the NC-Audit scheme. Thus, the computation cost on the TPA-side is also $O(1)$.

Communication. The communication costs in the RDC-NC, NC-Audit and MD-POR schemes are given as follows:

- In the RDC-NC scheme, the client computes dn coded blocks and sends d coded blocks to each of n servers. The size of a coded block is $(\frac{|F|}{g} + g)$ as mentioned in the analysis about the storage cost. The number of clients is s . Therefore, the communication cost is $O(sdn(\frac{|F|}{g} + g))$.
- In the NC-Audit scheme, besides the communication cost like the RDC-NC scheme ($O(sdn(\frac{|F|}{g} + g))$), the client also must send all $(sgdn)$ coefficients which are used to create the coded blocks to the servers ($O(sgdn)$). Thus, the communication cost is $O(sdn(\frac{|F|}{g} + g) + sgdn)$.
- In the MD-POR scheme, each of s clients sends the aggregated coded block to each of n servers. The size of a coded block in the MD-POR scheme is $(\frac{|F|}{g} + sg)$ as in the analysis about the storage cost on the server-side. Thus, the communication cost is $O(ns(\frac{|F|}{g} + sg))$.

4.6.3 Check Cost

Computation on Client-side. The computation costs on the client-side in the RDC-NC, NC-Audit and MD-POR schemes are given as follows:

- In the RDC-NC scheme, the client receives the aggregated coded block and the aggregated tag from each of n servers, and verifies each of them using the secret key of the client. Thus, the computation cost on the client-side is $O(n)$.
- In the NC-Audit scheme, the clients do nothing because the TPA will check the servers instead of the clients. The computation cost on the client-side is $O(1)$.
- In the MD-POR scheme, the clients also do nothing because the TPA will check the servers instead of the clients. The computation cost on the client-side is also $O(1)$.

Computation on Server-side. The computation costs on the server-side in the RDC-NC, NC-Audit and MD-POR schemes are given as follows:

- In the RDC-NC scheme, each of n servers combines its *spotcheck* coded blocks and its *spotcheck* tags where $spotcheck \in \{1, \dots, d\}$, and sends the aggregated coded block and the aggregated tag to the client. Thus, the computation cost on the server-side is $O(nd)$.
- In the NC-Audit scheme, each of n servers combines its *spotcheck* coded blocks and its *spotcheck* tags where $spotcheck \in \{1, \dots, d\}$, and sends the aggregated coded block and the aggregated tag to the TPA. Thus, the computation cost on the server-side is also $O(nd)$.
- In the MD-POR scheme, similar to the NC-Audit scheme, each of n servers combines its *spotcheck* coded blocks and its *spotcheck* tags where $spotcheck \in \{1, \dots, d\}$, and sends the aggregated coded block and the aggregated tag to the TPA. Thus, the computation cost on the server-side is also $O(nd)$.

Computation on TPA-side. The computation costs on the TPA-side in the RDC-NC, NC-Audit and MD-POR schemes are given as follows:

- In the RDC-NC scheme, the TPA does not exist. Thus, the computation cost on the TPA-side is N/A.
- In the NC-Audit scheme, the TPA verifies the aggregated coded block which is sent by each of n servers. The computation cost on the TPA-side is $O(n)$.
- In the MD-POR scheme, similar to the NC-Audit, the TPA also verifies the aggregated coded block which is sent from each of n servers. The computation cost on the TPA-side is also $O(n)$.

Communication. The communication costs in the RDC-NC, NC-Audit and MD-POR schemes are given as follows:

- In the RDC-NC scheme, during the check phase, each of n servers sends its aggregated coded block and its aggregated tag to the client. The size of a coded block is $(\frac{|F|}{g} + g)$. Thus, the communication cost is $O(n(\frac{|F|}{g} + g))$.
- In the NC-Audit scheme, similar to the RDC-NC scheme, during the check phase, each of n servers sends its aggregated coded block and its aggregated tag to the client. The size of a coded block is $(\frac{|F|}{g} + g)$. Thus, the communication cost is $O(n(\frac{|F|}{g} + g))$.
- In the MD-POR scheme, the mechanism is the same as the RDC-NC and NC-Audit scheme, but the different thing is that the size of a coded block in the MD-POR scheme is $(\frac{|F|}{g} + sg)$. Thus, the communication cost is $O(n(\frac{|F|}{g} + sg))$.

4.6.4 Repair Cost

Computation on Client-side. The computation costs on the client-side in the RDC-NC, NC-Audit and MD-POR schemes are given as follows:

- In the RDC-NC scheme, in the repair phase, the client firstly must check the l coded blocks which are provided from l healthy servers ($O(l)$). Thereafter, the client computes d new coded blocks for the new server by linearly combining l provided coded blocks ($O(ld)$). Thus, the computation cost on the client-side is $O((l + 1)d)$.
- In the NC-Audit scheme, the clients do nothing. Thus, the computation cost on the client-side is $O(1)$.
- In the MD-POR scheme, similar to the NC-Audit scheme, the clients do nothing. Thus, the computation cost on the client-side is $O(1)$.

Computation on Server-side. The computation costs on the server-side in the RDC-NC, NC-Audit and MD-POR schemes are given as follows:

- In the RDC-NC scheme, each of l healthy servers is required to combine its *spotcheck* coded blocks and *spotcheck* tags where *spotcheck* $\in \{1, \dots, d\}$. Thus, the computation cost on the server-side is $O(dl)$. The computation cost on the new server is N/A because the direct repair feature is not supported in the RDC-NC scheme.
- In the NC-Audit scheme, not only l healthy servers combine their coded blocks ($O(dl)$) but also the new server must compute its d new coded blocks by linearly combining l provided coded blocks ($O(dl)$).
- In the MD-POR scheme, similar to the NC-Audit scheme, only l healthy servers combine their coded blocks ($O(dl)$) but also the new server must compute its d new coded blocks by linearly combining l provided coded blocks ($O(dl)$).

Computation on TPA-side. The computation cost on the TPA-side in the RDC-NC, NC-Audit and MD-POR schemes are given as follows:

- In the RDC-NC scheme, the TPA does not exist. The computation cost on the TPA-side is N/A.
- In the NC-Audit scheme, the TPA must check l coded blocks which are provided by l servers during the repair phase. Thus, the computation cost on the TPA-side is $O(l)$.
- In the MD-POR scheme, the TPA does nothing because the new server will check pollution attack, not the TPA as the NC-Audit scheme. Therefore, the computation cost on the TPA-side is $O(1)$.

Communication. The communication costs in the RDC-NC, NC-Audit and MD-POR schemes are given as follows:

- In the RDC-NC scheme, each of l healthy servers sends the aggregated coded block whose size is $(\frac{|F|}{g} + g)$ to a client ($O(l(\frac{|F|}{g} + g))$). After computing d new coded blocks, the client sends them to the new server ($O(d(\frac{|F|}{g} + g))$). Therefore, the communication cost is $O((l + d)(\frac{|F|}{g} + g))$.
- In the NC-Audit scheme, each of l healthy servers also sends the aggregated coded block to the new server ($O(l(\frac{|F|}{g} + g))$). Then, the new server sends its linear coefficients which are used to compute d new coded blocks from l provided coded blocks to the TPA ($O(ld)$). Therefore, the communication cost is $O(l(\frac{|F|}{g} + g) + ld)$.
- In the MD-POR scheme, only each of l healthy servers sends the aggregated coded block to the new server. The size of each coded block is $(\frac{|F|}{g} + sg)$. Therefore, the communication cost is $O(l(\frac{|F|}{g} + sg))$.

4.6.5 Total cost

Although the MD-POR scheme supports many heavy features, the total cost in the MD-POR scheme is still better than the previous schemes. Let $O_p(A)$, $O_p(B)$, $O_p(C)$ denote the total computation costs of the RDC-NC, NC-Audit and MD-POR schemes, respectively. Let $O_m(A)$, $O_m(B)$, $O_m(C)$ denote the total communication costs of the RDC-NC, NC-Audit and MD-POR schemes, respectively. Let $O_s(A)$, $O_s(B)$, $O_s(C)$ denote the total storage costs of the RDC-NC, NC-Audit and MD-POR schemes, respectively. In reality, d and g are far larger than s and n ($d, g \gg s, n$), $l \in \{1, \dots, n\}$, and $d > g$. From Table 4.2, the following results are obtained.

- The difference between the computations costs of the RDC-NC scheme and our scheme is:

$$O_p(A) - O_p(C) = (gdn + d) - (sdn + g) \quad (4.36)$$

Because $g \gg s$, $O_p(A) - O_p(C) > 0$. Thus, $O_p(A) > O_p(C)$. This indicates that the RDC-NC incurs higher computation cost than the MD-POR scheme.

- The difference between the computations costs of the NC-Audit scheme and the MD-POR scheme is:

$$O_p(B) - O_p(C) = (gdn + l) - (sdn + g) \quad (4.37)$$

Because $g \gg s$, $O_p(B) - O_p(C) > 0$. Thus, $O_p(B) > O_p(C)$. This indicates that the NC-Audit incurs higher computation cost than the MD-POR scheme.

- The difference between the communication costs of the RDC-NC scheme and the MD-POR scheme is:

$$O_m(A) - O_m(C) = (dns + d - ns) \frac{|F|}{g} + g(sdn + n + l + d - ns^2 - ns - ls) \quad (4.38)$$

Because $d \gg s$ and $1 \leq l \leq n$, $O_m(A) - O_m(C) > 0$. Thus, $O_m(A) > O_m(C)$. This indicates that the RDC-NC scheme incurs higher communication cost than the MD-POR scheme.

- The difference between the communication costs of the NC-Audit scheme and the MD-POE scheme is:

$$O_m(B) - O_m(C) = (dns - ns) \frac{|F|}{g} + g(2sdn + n + l - ns^2 - ns - ls) + ld \quad (4.39)$$

Because $d \gg s$ and $1 \leq l \leq n$, $O_m(B) - O_m(C) > 0$. Thus, $O_m(B) > O_m(C)$. This indicates that the NC-Audit scheme incurs higher communication cost than the MD-POR scheme.

- The difference between the storage costs of the RDC-NC scheme and our scheme is:

$$O_s(A) - O_s(C) = (3\xi + 5g) \log_2 q + (s - 1)dn \frac{|F|}{g} - 2sg \log_2 q \quad (4.40)$$

Because $\frac{|F|}{g} = \xi \log_2 q$ and $d > g$, $O_s(A) - O_s(C) > 0$. Thus, $O_s(A) > O_s(C)$. This indicates that the RDC-NC scheme incurs higher storage cost than the MD-POR scheme.

- The difference between the storage costs of the NC-Audit scheme and the MD-POR scheme is:

$$O_s(B) - O_s(C) = (s - 1)dn \frac{|F|}{g} + g \log_2 q (dn - 2s + 2) \quad (4.41)$$

Because $d \gg s$, $O_s(B) - O_s(C) > 0$. Thus, $O_s(B) > O_s(C)$. This indicates that the NC-Audit scheme incurs higher storage cost than the MD-POR scheme.

4.7 Performance Evaluation

In this section, we evaluate the computation and communication performances of the MD-POR scheme to show that it is applicable for a real system. A program written by Python 2.7.3 is executed using a computer with Intel Core i5 processor, 2.4 GHz, 4 GB of RAM, Windows 7 64-bit OS. The parameters are set as follows:

- The length of the prime q is set to be 256 bits.
- The number of clients is set to be 2 ($s = 2$).
- The number of servers is set to be 3 ($n = 3$).
- The number of coded blocks stored in each server is set to be 10 ($d = 10$).
- The number of healthy servers which are used for repairing is set to be 3 ($l = 2$).
- The number of spot checks during the check and repair phases is set to be $d/2$.
- Each result is the average of 100 runs.

4.7.1 Computation Performance

Because $|F_i| = g \times |v_{ij}|$ (where $|F_i|$ is file size, g is number of file blocks, and $|v_{ij}|$ is block size), for the computation performance, we implemented the scheme with two cases:

- Case 1: Fix g to be 80 \rightarrow change $|v_{ij}|$ according to $|F_i|$.
- Case 2: Fix block size to be 1 MB ($|v_{ij}| = 2^{23}$ bits) \rightarrow change g according to $|F_i|$.

CASE 1: fix number of blocks, change block size

In this case, we fix g to be 80; then we change $|v_{ij}|$ according to $|F_i|$. Namely:

- $|F_i| = 10MB \Rightarrow |v_{ij}| = 0.125MB = 128KB$.
- $|F_i| = 20MB \Rightarrow |v_{ij}| = 0.25MB = 256KB$.
- $|F_i| = 30MB \Rightarrow |v_{ij}| = 0.375MB = 384KB$.
- $|F_i| = 40MB \Rightarrow |v_{ij}| = 0.5MB = 512KB$.
- $|F_i| = 50MB \Rightarrow |v_{ij}| = 0.625MB = 640KB$.

The summary of computation results in this case are described in Table 4.3. These results can also be observed in Figure 4.3 (init and keygen), Figure 4.4 (encode), Figure 4.5 (check), and Figure 4.6 (repair) by varying the file size of each client.

Table 4.3: Summary of computation results in case 1 (time unit: second)

	$ F = 10MB$ ($ v_{ij} = 0.125MB = 128KB$)	$ F = 20MB$ ($ v_{ij} = 0.25MB = 256KB$)	$ F = 30MB$ ($ v_{ij} = 0.375MB = 384KB$)	$ F = 40MB$ ($ v_{ij} = 0.5MB = 512KB$)	$ F = 50MB$ ($ v_{ij} = 0.625MB = 640KB$)
Init (client)	18.097	33.447	51.668	68.406	86.362
Keygen (client)	165.282	419.890	682.930	1124.515	2488.914
Encode					
(client)	1.374	2.683	4.025	5.397	6.739
(server)	0.151	0.307	0.452	0.609	0.842
Check					
challenge (TPA)	0.015	0.016	0.016	0.016	0.015
respond (server)	0.141	0.286	0.421	0.561	0.702
verify (TPA)	0.026	0.031	0.047	0.062	0.078
Repair					
(new server)	0.187	0.359	0.546	0.718	0.936
(server)	0.141	0.286	0.421	0.561	0.702

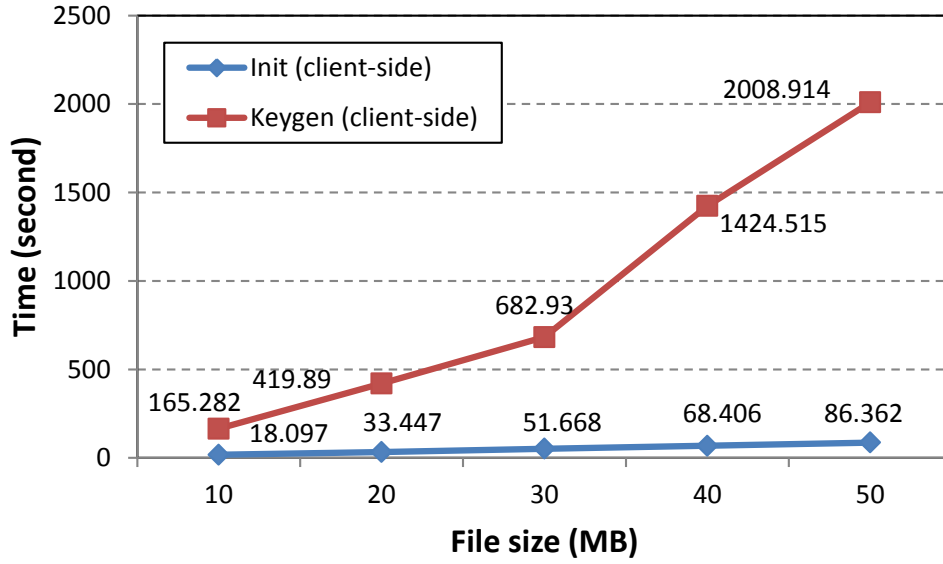


Figure 4.3: Computation time performance of init and keygen phases

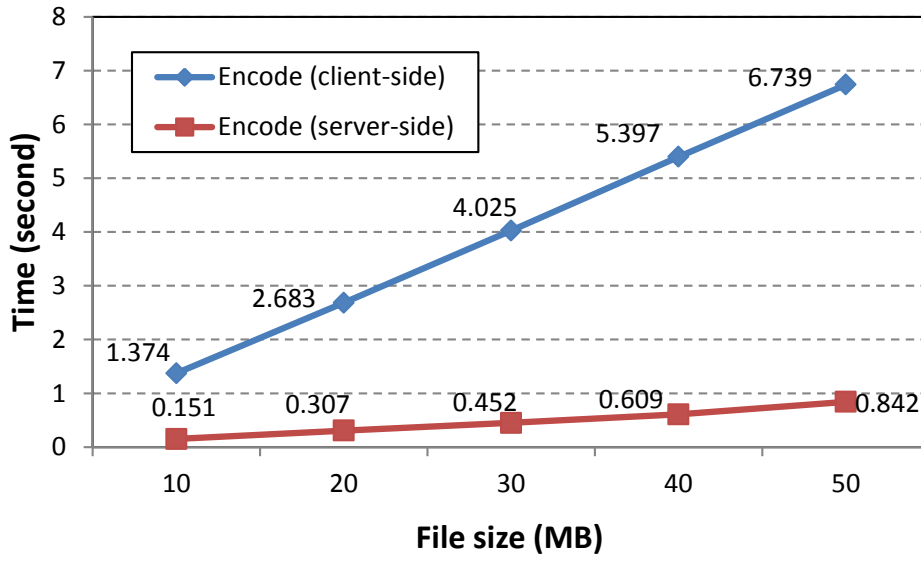


Figure 4.4: Computation time performance of encode phase

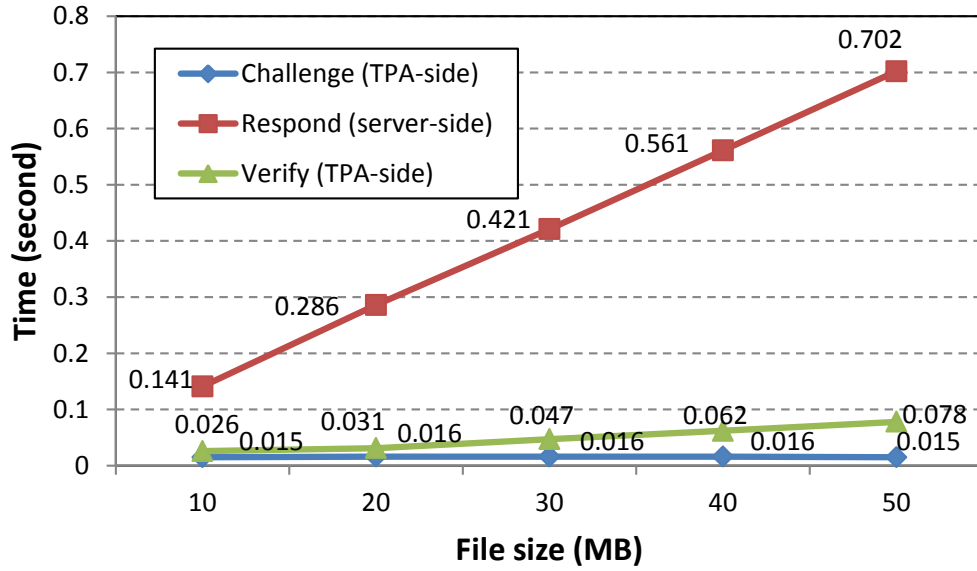


Figure 4.5: Computation time performance of check phase

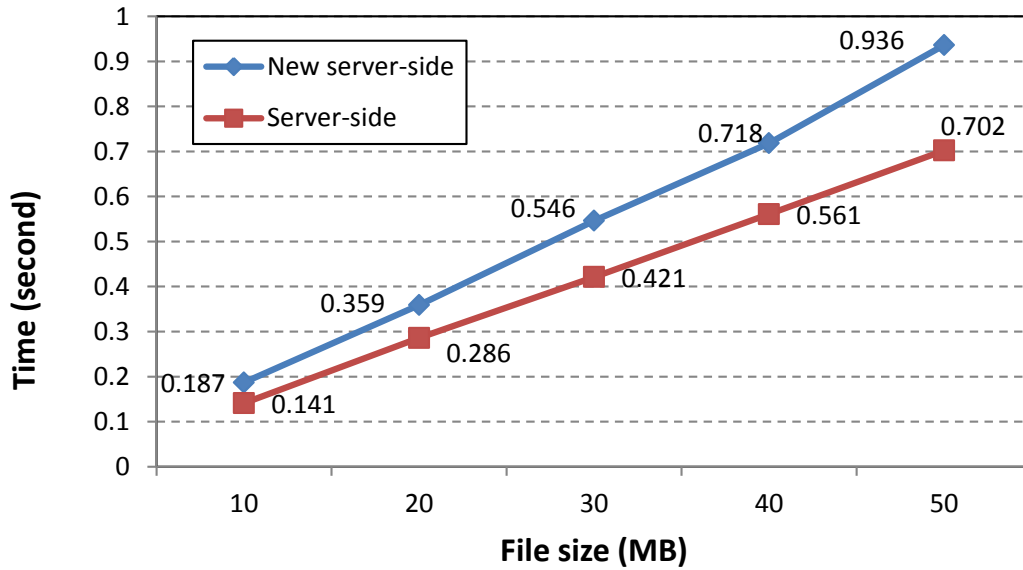


Figure 4.6: Computation time performance of repair phase

The experiment results reveal that the computation time increases almost linearly as the file size increases, and each graph has a different slope. Only the computation time of challenge step on the TPA-side in the check phase is almost constant. **Note that the init, keygen and encode phases are only executed one time in the beginning.** Meanwhile, the check and repair phases are executed very often during the

system lifetime. Therefore, the computation time of the check and repair phases are more important.

- In Figure 4.3, the graph slopes of the init and keygen phases are approximately 1.707 and 46.091, respectively. If the file size is 1 GB, the computation time of the init and keygen phases is estimated as 1748.615 seconds and 46901.35 seconds, respectively (*performed only one time*).
- In Figure 4.4, the graph slopes of the encode phase on client-side and server-side are approximately 0.134 and 0.017, respectively. If the file size is 1 GB, the computation time of the encode phase on client-side and server-side is estimated as 137.377 seconds and 17.67 seconds, respectively (*performed only one time*).
- In Figure 4.5, the graph slopes of the check phase (challenge, respond and verify) are approximately 0, 0.014 and 0.001, respectively. If the file size is 1 GB, the computation time of the challenge, respond and verify algorithms is estimated as 0.015 seconds, 14.362 seconds and 1.344 seconds, respectively (*performed often*).
- In Figure 4.6, the graph slopes of the repair phase on new server-side and healthy server-side are approximately 0.019 and 0.014, respectively. If the file size is 1 GB, the computation time of the repair phase on new server-side and healthy server-side is estimated as 19.174 seconds and 14.362 seconds, respectively (*performed often*).

CASE 2: fix block size, change number of blocks

In this case, we fix block size to be 1 MB ($|v_{ij}| = 2^{23}$ bits), and then change g according to $|F_i|$. Namely:

- $|F_i| = 10MB \Rightarrow g = 10$.
- $|F_i| = 20MB \Rightarrow g = 20$.
- $|F_i| = 30MB \Rightarrow g = 30$.
- $|F_i| = 40MB \Rightarrow g = 40$.
- $|F_i| = 50MB \Rightarrow g = 50$.

The summary of computation results in this case are described in Table 4.4. These results can also be observed in Figure 4.7 (init and keygen), Figure 4.8 (encode), Figure 4.9 (check), and Figure 4.10 (repair) by varying the file size of each client.

Table 4.4: Summary of computation results in case 2 (time unit: second)

	$ F = 10MB$ ($g = 10$)	$ F = 20MB$ ($g = 20$)	$ F = 30MB$ ($g = 30$)	$ F = 40MB$ ($g = 40$)	$ F = 50MB$ ($g = 50$)
Init (client)	18.721	40.841	58.557	76.206	95.231
Keygen (client)	7346.419	7631.388	7777.486	8849.160	9102.480
Encode					
(client)	1.435	2.800	4.111	5.507	6.795
(server)	1.258	1.259	1.310	1.326	1.336
Check					
challenge (TPA)	0.015	0.016	0.016	0.015	0.016
respond (server)	1.134	1.138	1.144	1.144	1.166
verify (TPA)	1.125	1.125	1.128	1.132	1.137
Repair					
(new server)	1.400	1.414	1.435	1.451	1.56
(server)	1.134	1.138	1.144	1.144	1.166

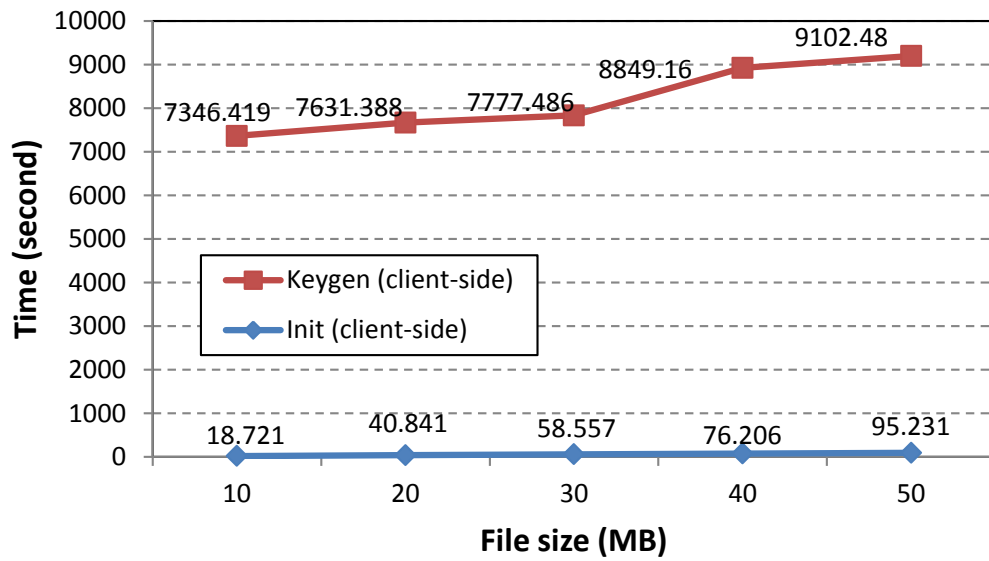


Figure 4.7: Computation time performance of init and keygen phases

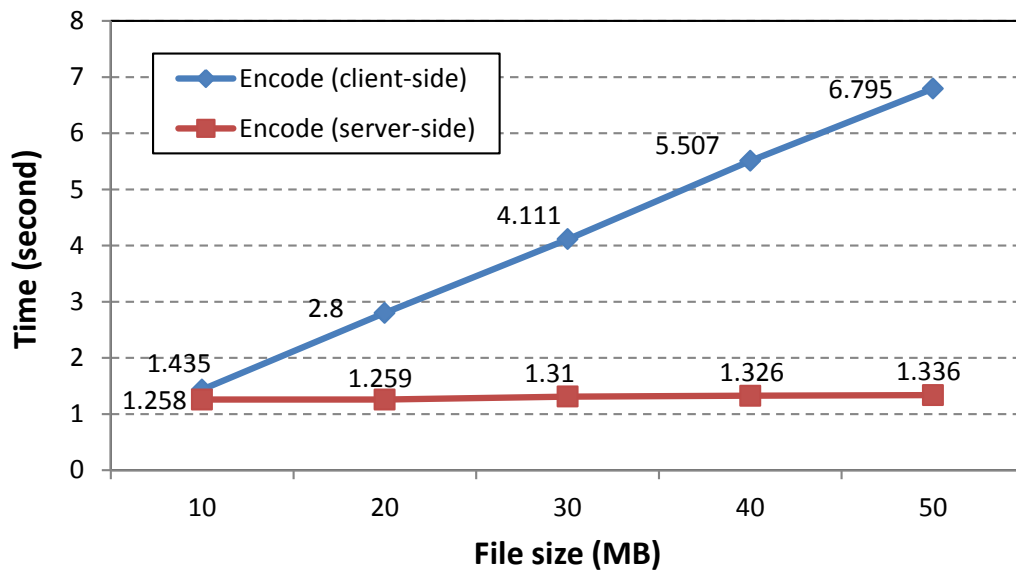


Figure 4.8: Computation time performance of encode phase

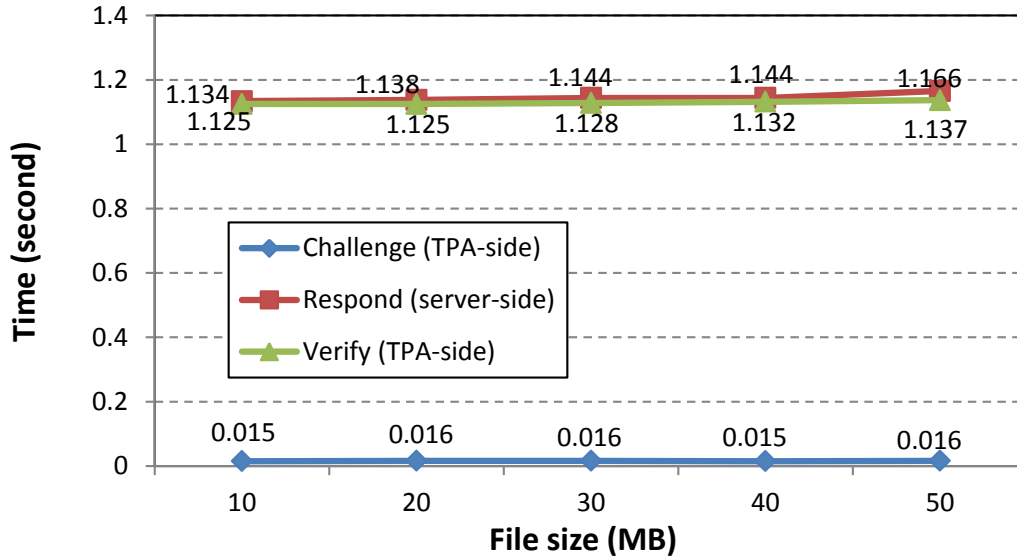


Figure 4.9: Computation time performance of check phase

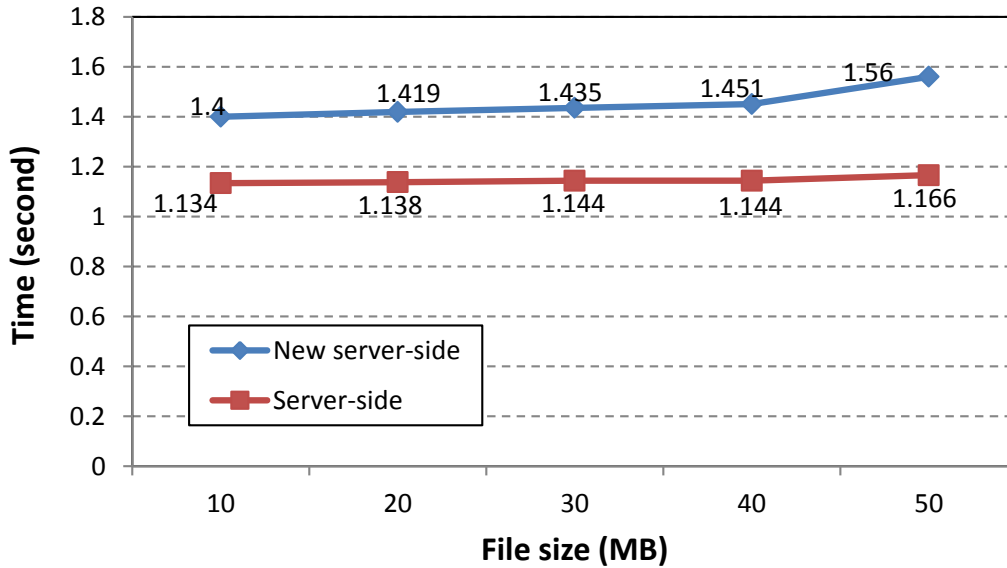


Figure 4.10: Computation time performance of repair phase

The experiment results reveal that the computation time increases almost linearly as the file size increases, and each graph has a different slope. Only the computation time of challenge step on the TPA-side in the check phase is almost constant. **Note that the init, keygen and encode phases are only executed one time in the beginning.** Meanwhile, the check and repair phases are executed very often during the

system lifetime. Therefore, the computation time of the check and repair phases are more important.

- In Figure 4.7, the graph slopes of the init and keygen phases are approximately 1.913 and 43.902, respectively. If the file size is 1 GB, the computation time of the init and keygen phases is estimated as 1958.25 seconds and 51862.57 seconds, respectively (*performed only one time*).
- In Figure 4.4, the graph slopes of the encode phase on client-side and server-side are approximately 0.134 and 0.002, respectively. If the file size is 1 GB, the computation time of the encode phase on client-side and server-side is estimated as 137.311 seconds and 3.235 seconds, respectively (*performed only one time*).
- In Figure 4.5, the graph slopes of the check phase (challenge, respond and verify) are approximately 0, 0.0008 and 0.0003, respectively. If the file size is 1 GB, the computation time of the challenge, respond and verify algorithms is estimated as 0.016 seconds, 1.945 seconds and 1.429 seconds, respectively (*performed often*).
- In Figure 4.6, the graph slopes of the repair phase on new server-side and healthy server-side are approximately 0.004 and 0.0008, respectively. If the file size is 1 GB, the computation time of the repair phase on new server-side and healthy server-side is estimated as 5.456 seconds and 1.945 seconds, respectively (*performed often*).

CASE 1 vs CASE 2

In this section, we compare case 1 and case 2 for 1 GB file in Figure 4.11 (keygen), Figure 4.12 (encode), Figure 4.13 (check) and Figure 4.14 (repair). The results show that:

- In keygen phase, the computation time of case 1 is better than that of case 2. Thus, block size is the dominant parameter.
- In the other three phases (encode, check and repair phases), the computation time of case 2 is better than that of case 1. Thus, number of file blocks is the dominant parameter.

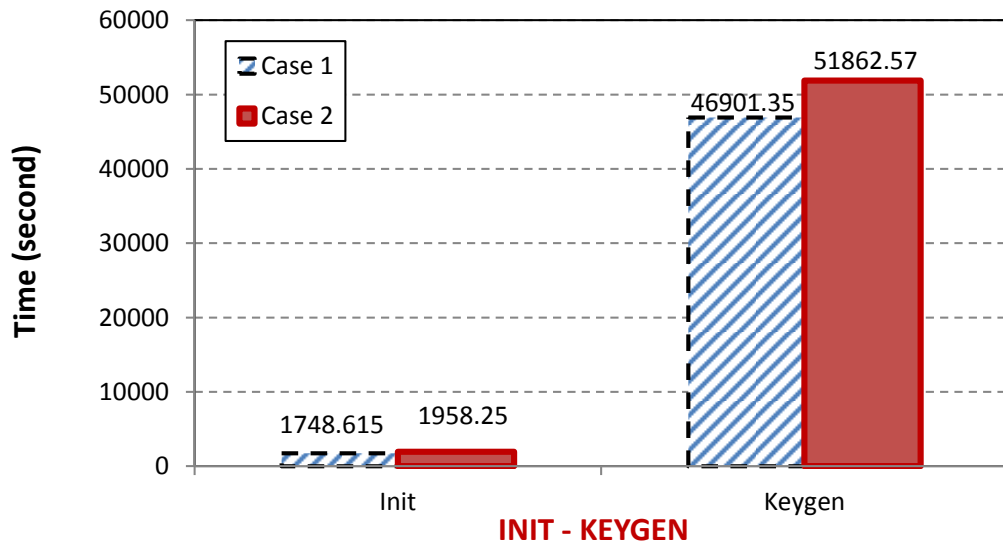


Figure 4.11: Computation time performance of init and keygen phases

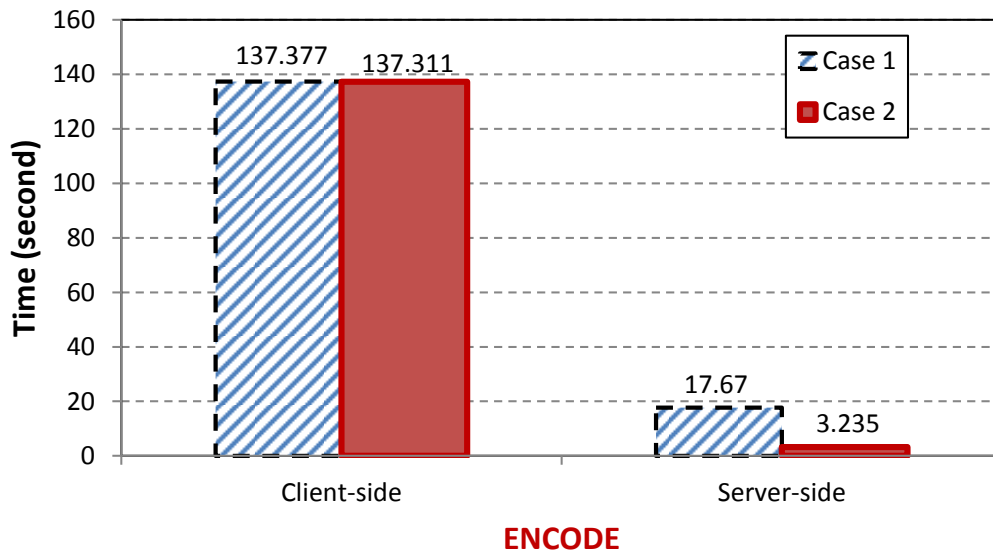


Figure 4.12: Computation time performance of encode phase

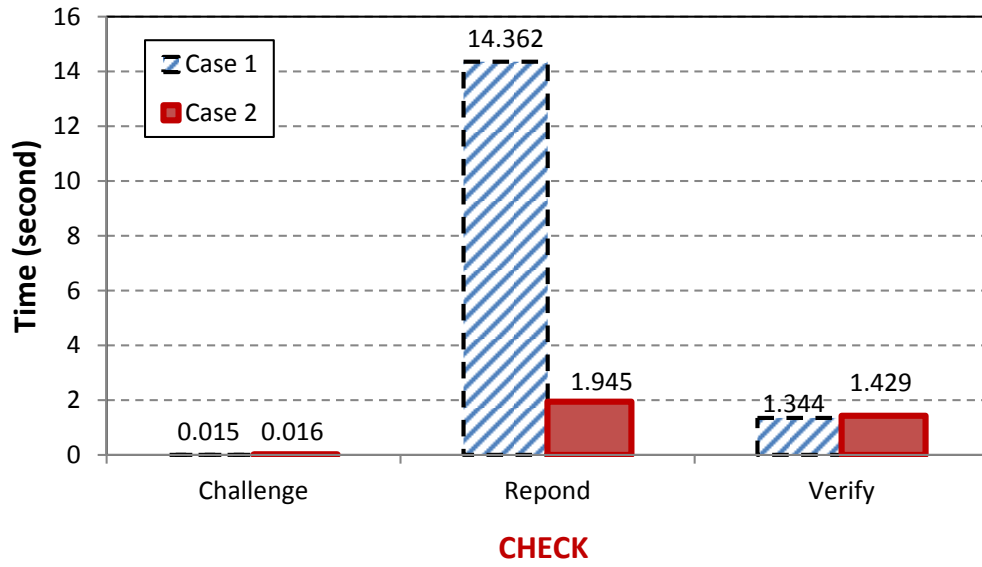


Figure 4.13: Computation time performance of check phase

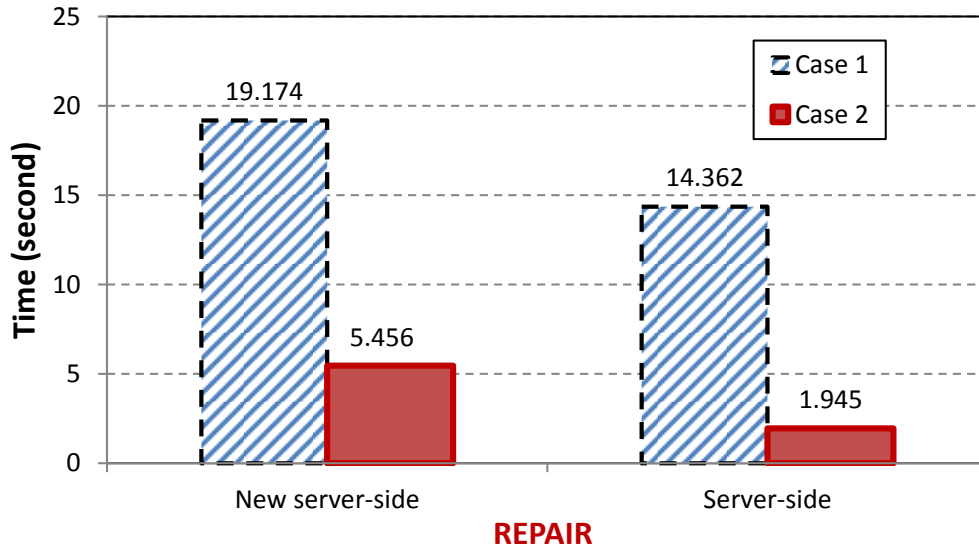


Figure 4.14: Computation time performance of repair phase

4.7.2 Communication Performance

The communication results can be observed with a set of communication performance by varying the file size of each client. These results are depicted in Figure 4.15 (encode, check, and repair).

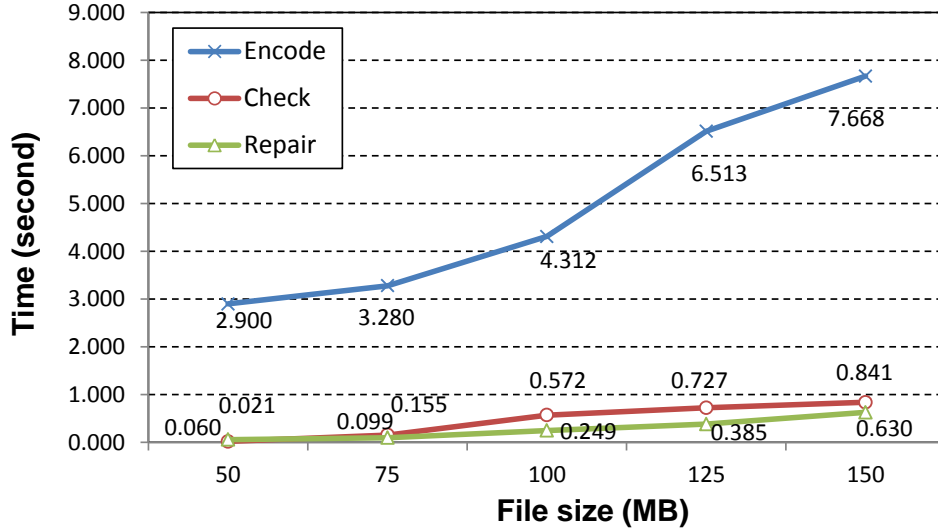


Figure 4.15: Communication time performance

The MD-POR scheme is performed with the bandwidth of 300Mbps. The experiment results reveal that the communication time increases almost linearly as the file size increases, and each graph in Figure 4.15 has a different slope. The slopes of increment in the graphs of the encode phase, the check phase and the repair phase are approximately 0.048, 0.008 and 0.006, respectively. Therefore, if the file size is 1 GB, the communication time of the encode phase, check phase, and repair phase is estimated as 49.27 seconds, 7.86 seconds and 5.83 seconds, respectively. In addition, the size of the response from each server is given as follows. The response sizes of the 50MB, 75MB, 100MB, 125MB and 150MB file size are 13KB, 19KB, 26KB, 32KB and 38KB, respectively. Therefore, if the file size is 1 GB, the response size is estimated as 264.87 KB.

The above results indicate that the computation and communication performances are very fast even when the file size is 1 GB.

4.8 Numeric Example of Keygen Phase

In this section, an example is given to explain how the keygen phase in our scheme works. Suppose that there are two clients: \mathcal{C}_1 and \mathcal{C}_2 . The augmented blocks are:

- $w_{11} = (2, 1, 0, 0, 0)$
- $w_{12} = (3, 0, 1, 0, 0)$
- $w_{21} = (1, 0, 0, 1, 0)$
- $w_{22} = (5, 0, 0, 0, 1)$

Suppose that all operations work in \mathbb{F}_7 .

4.8.1 The key of the client \mathcal{C}_1

A matrix M_1 is constructed in a way that it consists of all augmented blocks which do not belong to the client \mathcal{C}_1 :

$$M_1 = \begin{pmatrix} w_{21} \\ w_{22} \end{pmatrix} = \begin{pmatrix} 1, 0, 0, 1, 0 \\ 5, 0, 0, 0, 1 \end{pmatrix} \quad (4.42)$$

M_1 is then reduced by the Gauss-Jordan elimination to a row echelon form as follows:

$$M'_1 = \begin{pmatrix} \boxed{1}, 0, 0, 1, 0 \\ 0, 0, 0, \boxed{1}, 4 \end{pmatrix} \quad (4.43)$$

Let $\delta_1, \dots, \delta_5$ denote the unknown variables which correspond to the columns of M'_1 . Let $\delta = [\delta_1, \dots, \delta_5]^T$. The pivots which are the values in the squares belong to δ_1 and δ_4 . The free variables are δ_2, δ_3 and δ_5 . Solving $M'_1 \cdot \delta = 0$, we have $\delta_1 + \delta_4 = 0$ and $\delta_4 + 4\delta_5 = 0$. Let $\delta_2 = a, \delta_3 = b$ and $\delta_5 = c$.

$$\begin{pmatrix} \delta_1 \\ \delta_2 \\ \delta_3 \\ \delta_4 \\ \delta_5 \end{pmatrix} = a \begin{pmatrix} 0 \\ 1 \\ 0 \\ 0 \\ 0 \end{pmatrix} + b \begin{pmatrix} 0 \\ 0 \\ 1 \\ 0 \\ 0 \end{pmatrix} + c \begin{pmatrix} 4 \\ 0 \\ 0 \\ -4 \\ 1 \end{pmatrix} \quad (4.44)$$

Because the number of free variables is 3, the number of elements in the basis is also 3. Namely, the basis is as follows:

$$\left\{ \begin{pmatrix} 0 \\ 1 \\ 0 \\ 0 \\ 0 \end{pmatrix}, \begin{pmatrix} 0 \\ 0 \\ 1 \\ 0 \\ 0 \end{pmatrix}, \begin{pmatrix} 4 \\ 0 \\ 0 \\ -4 \\ 1 \end{pmatrix} \right\} \quad (4.45)$$

Suppose that the random values are 2, 3 and 2. The key of \mathcal{C}_1 is computed as follows:

$$k_1 = 2 \begin{pmatrix} 0 \\ 1 \\ 0 \\ 0 \\ 0 \end{pmatrix} + 3 \begin{pmatrix} 0 \\ 0 \\ 1 \\ 0 \\ 0 \end{pmatrix} + 2 \begin{pmatrix} 4 \\ 0 \\ 0 \\ -4 \\ 1 \end{pmatrix} \pmod{7} = \begin{pmatrix} 1 \\ 2 \\ 3 \\ 6 \\ 2 \end{pmatrix} \quad (4.46)$$

We can observe that k_1 is orthogonal to w_{21} and w_{22} . In other words, $k_1 \cdot w_{21} = k_1 \cdot w_{22} = 0$ because:

$$k_1 \cdot w_{21} = \begin{pmatrix} 1 \\ 2 \\ 3 \\ 6 \\ 2 \end{pmatrix} \cdot (1, 0, 0, 1, 0) \pmod{7} = 0 \quad (4.47)$$

$$k_1 \cdot w_{22} = \begin{pmatrix} 1 \\ 2 \\ 3 \\ 6 \\ 2 \end{pmatrix} (5, 0, 0, 0, 1) \pmod{7} = 0 \quad (4.48)$$

4.8.2 The key of the client \mathcal{C}_2

Similarly, k_2 is also constructed in the same way as k_1 . Concretely, a matrix M_2 is constructed in a way that it consists of all augmented blocks which do not belong to the client \mathcal{C}_2 :

$$M_2 = \begin{pmatrix} w_{11} \\ w_{12} \end{pmatrix} = \begin{pmatrix} 2, 1, 0, 0, 0 \\ 3, 0, 1, 0, 0 \end{pmatrix} \quad (4.49)$$

M_1 is then reduced by the Gauss-Jordan elimination to a row echelon form as follows:

$$M'_2 = \begin{pmatrix} \boxed{1}, 4, 0, 0, 0 \\ 0, \boxed{1}, 4, 0, 0 \end{pmatrix} \quad (4.50)$$

The free variables are δ_3, δ_4 and δ_5 . The basis is as follows:

$$\left\{ \begin{pmatrix} 16 \\ -4 \\ 1 \\ 0 \\ 0 \end{pmatrix}, \begin{pmatrix} 0 \\ 0 \\ 0 \\ 1 \\ 0 \end{pmatrix}, \begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 1 \end{pmatrix} \right\} \quad (4.51)$$

Suppose that the random values are 2, 2 and 1. The key k_2 is constructed as follows:

$$k_2 = 2 \begin{pmatrix} 16 \\ -4 \\ 1 \\ 0 \\ 0 \end{pmatrix} + 2 \begin{pmatrix} 0 \\ 0 \\ 0 \\ 1 \\ 0 \end{pmatrix} + 1 \begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 1 \end{pmatrix} \pmod{7} = \begin{pmatrix} 4 \\ 6 \\ 2 \\ 2 \\ 1 \end{pmatrix} \quad (4.52)$$

We can observe that k_2 is orthogonal to w_{11} and w_{12} . In other words, $k_2 \cdot w_{11} = k_2 \cdot w_{12} = 0$ because:

$$k_2 \cdot w_{11} = \begin{pmatrix} 4 \\ 6 \\ 2 \\ 2 \\ 1 \end{pmatrix} (2, 1, 0, 0, 0) \pmod{7} = 0 \quad (4.53)$$

$$k_2 \cdot w_{12} = \begin{pmatrix} 4 \\ 6 \\ 2 \\ 2 \\ 1 \end{pmatrix} (3, 0, 1, 0, 0) \pmod{7} = 0 \quad (4.54)$$

4.8.3 The key of the TPA

The TPA is given the key κ as follows:

$$\kappa = k_1 + k_2 = \begin{pmatrix} 1 \\ 2 \\ 3 \\ 6 \\ 2 \end{pmatrix} + \begin{pmatrix} 4 \\ 6 \\ 2 \\ 2 \\ 1 \end{pmatrix} \pmod{7} = \begin{pmatrix} 5 \\ 1 \\ 5 \\ 1 \\ 3 \end{pmatrix} \quad (4.55)$$

We can observe that κ is orthogonal to all the augmented blocks of all the clients. In other words, $\kappa \cdot w_{11} = \kappa \cdot w_{12} = \kappa \cdot w_{21} = \kappa \cdot w_{22} = 0$ because:

$$\kappa \cdot w_{11} = \begin{pmatrix} 5 \\ 1 \\ 5 \\ 1 \\ 3 \end{pmatrix} (2, 1, 0, 0, 0) \pmod{7} = 0 \quad (4.56)$$

$$\kappa \cdot w_{12} = \begin{pmatrix} 5 \\ 1 \\ 5 \\ 1 \\ 3 \end{pmatrix} (3, 0, 1, 0, 0) \pmod{7} = 0 \quad (4.57)$$

$$\kappa \cdot w_{21} = \begin{pmatrix} 5 \\ 1 \\ 5 \\ 1 \\ 3 \end{pmatrix} (1, 0, 0, 1, 0) \pmod{7} = 0 \quad (4.58)$$

$$\kappa \cdot w_{22} = \begin{pmatrix} 5 \\ 1 \\ 5 \\ 1 \\ 3 \end{pmatrix} (5, 0, 0, 0, 1) \pmod{7} = 0 \quad (4.59)$$

4.8.4 The key of the new server

The new server is given the key κ' as: $\kappa' = \kappa + k_{repair}$ where k_{repair} is constructed as follows. Firstly, a matrix M_r is constructed in a way that it consists of all augmented blocks:

$$M_r = \begin{pmatrix} w_{11} \\ w_{12} \\ w_{21} \\ w_{22} \end{pmatrix} = \begin{pmatrix} 2, 1, 0, 0, 0 \\ 3, 0, 1, 0, 0 \\ 1, 0, 0, 1, 0 \\ 5, 0, 0, 0, 1 \end{pmatrix} \quad (4.60)$$

M_r is then reduced by the Gauss-Jordan elimination to a row echelon form as follows:

$$M'_r = \begin{pmatrix} \boxed{1}, 0, 0, 0, 3 \\ 0, \boxed{1}, 0, 0, 1 \\ 0, 0, \boxed{1}, 4, 0 \\ 0, 0, 0, \boxed{1}, 4 \end{pmatrix} \quad (4.61)$$

The free variable is δ_5 . The basis is as follows:

$$\left\{ \begin{pmatrix} -3 \\ -1 \\ 16 \\ -4 \\ 1 \end{pmatrix} \right\} \quad (4.62)$$

Suppose the random value is 3. k_{repair} is computed as follows:

$$k_{repair} = 3 \begin{pmatrix} -3 \\ -1 \\ 16 \\ -4 \\ 1 \end{pmatrix} \pmod{7} = \begin{pmatrix} 5 \\ 4 \\ 6 \\ 2 \\ 3 \end{pmatrix} \quad (4.63)$$

We can see that k_{repair} is orthogonal to all augmented blocks: $k_{repair} \cdot w_{11} = k_{repair} \cdot w_{12} = k_{repair} \cdot w_{21} = k_{repair} \cdot w_{22} = 0$ because:

$$k_{repair} \cdot w_{11} = \begin{pmatrix} 5 \\ 4 \\ 6 \\ 2 \\ 3 \end{pmatrix} (2, 1, 0, 0, 0) \pmod{7} = 0 \quad (4.64)$$

$$k_{repair} \cdot w_{12} = \begin{pmatrix} 5 \\ 4 \\ 6 \\ 2 \\ 3 \end{pmatrix} (3, 0, 1, 0, 0) \pmod{7} = 0 \quad (4.65)$$

$$k_{repair} \cdot w_{21} = \begin{pmatrix} 5 \\ 4 \\ 6 \\ 2 \\ 3 \end{pmatrix} (1, 0, 0, 1, 0) \pmod{7} = 0 \quad (4.66)$$

$$k_{repair} \cdot w_{22} = \begin{pmatrix} 5 \\ 4 \\ 6 \\ 2 \\ 3 \end{pmatrix} (5, 0, 0, 0, 1) \pmod{7} = 0 \quad (4.67)$$

Finally, κ' is computed as follows:

$$\kappa' = \kappa + k_{repair} = \begin{pmatrix} 5 \\ 1 \\ 5 \\ 1 \\ 3 \end{pmatrix} + \begin{pmatrix} 5 \\ 4 \\ 6 \\ 2 \\ 3 \end{pmatrix} \pmod{7} = \begin{pmatrix} 3 \\ 5 \\ 4 \\ 3 \\ 6 \end{pmatrix} \quad (4.68)$$

4.9 Summary

In this chapter, a new network coding-based POR scheme named the MD-POR has been proposed. The MD-POR scheme supports multi-client, symmetric key-based direct repair and public authentication features. Moreover, the MD-POR scheme can protect against a strong adversary who can perform mobile attack, curious attack, response forgery and pollution attack. In addition, the efficiency analysis based on the complexity theory shows that although the MD-POR scheme supports many features, its costs are not bad compared with the previous schemes. The experiment results reveal that the computation time increases as the file size increases. However, the graphs show that the slope of increment for the MD-POR scheme increases merely. Future work is investigated to implement two previous RDC-NC and NC-Audit schemes in order to compare with the MD-POR scheme. The implementation of the MD-POR scheme shows that its computation cost is applicable for a real system.

Chapter 5

DD-POR: Dynamic Operations and Direct Repair for POR

5.1 System Model

The system model of the DD-POR scheme consists of three types of entities:

- Key manager: This entity is fully trusted, and has the responsibility to generate the keys for the other entities.
- Client: There is only a single client who can be either an enterprise or an individual customer. The client owns his/her data and wants to store the data in the cloud servers. The client relies on the cloud for data storage, computation, and maintenance.
- Servers: The servers are managed and monitored by a cloud service provider to accommodate a service of data storage and have significant and unlimited storage space and computation resources. In the cloud storage service, the client can store his/her data into a set of servers in a simultaneous and distributed manner.

5.2 Adversarial Model

In this scheme, the key manager and the client are trusted; meanwhile, the servers are untrusted. We assume that the servers do not collude with each other. The servers can perform the following attacks:

1. In the check phase: the servers may disrupt or modify the data. These attacks can be commonly prevented by the MAC tags, we thus do not focus on this attack.
2. In the repair phase: the servers may perform: (i) the pollution attack which is a common attack of the network coding, and (ii) the curious attack which is a special attack of the direct repair. We focus on these attacks in the security analysis.

- *Pollution Attack.* This attack is also considered in several papers [79, 82, 86–92, 137]. The malicious server firstly uses a valid coded block to pass the check phase, but then injects an invalid coded block in the repair phase to prevent data repair. An example is given as follows:
 - Encode: the client encodes the augmented blocks (w_1, w_2, w_3) to six coded blocks: c_{11}, c_{12} (stored in the server \mathcal{S}_1), c_{21}, c_{22} (stored in the server \mathcal{S}_2), and c_{31}, c_{32} (stored in the server \mathcal{S}_3). Suppose that \mathcal{S}_1 will perform a pollution attack.
 - Check: suppose that \mathcal{S}_3 is corrupted.
 - Repair: \mathcal{S}_3 should be repaired by two coded blocks: c'_{31} (which is a linear combination of c_{11} and c_{12}) and c'_{32} (which is a linear combination of c_{21} and c_{22}). However, \mathcal{S}_1 is not detected because this time is the repair phase, not the check phase. The client still thinks \mathcal{S}_1 is healthy, and thus the client requests the coded blocks from \mathcal{S}_1 and \mathcal{S}_2 . \mathcal{S}_1 will provide an invalid coded block c''_{31} to the client instead of c'_{31} .
- *Curious Attack.* This attack is the a specific attack in our scheme. This attack is performed by the new server in the repair phase. Every repair time, the new server is given a key k_r constructed from the secret key k_c of the client and a variant k_p . Having k_r , the new server may try to obtain k_c in order to pass the check phases in the later epochs.

5.3 Proposed DD-POR Scheme

The notations which are used throughout the DD-POR scheme are given in Table 5.1.

Table 5.1: List of notations in the DD-POR scheme.

Notation	Description
\mathcal{C}	client
F	original file
m	number of file blocks
n	number of servers
d	number of coded blocks in each server
k	file block index ($k \in \{1, \dots, m\}$)
i	server index ($i \in \{1, \dots, n\}$)
j	coded block index in each server ($j \in \{1, \dots, d\}$)
v_k	file block ($k \in \{1, \dots, m\}$)
w_k	augmented block of v_k
t_{w_k}	tag of w_k
\mathcal{S}_i	server
c_{ij}	j -th coded block stored in \mathcal{S}_i
t_{ij}	tag of c_{ij}

l	number of healthy servers used for data repair
\mathcal{S}_r	corrupted server
\mathcal{S}'_r	new server which is used to replace \mathcal{S}_r
\mathbb{F}_q^ξ	ξ -dimensional finite field \mathbb{F} of a prime order q
$ $	concatenate operation
$spotcheck$	number of coded blocks in a server which are checked during the check and repair phases ($spotcheck \in \{1, \dots, d\}$)

5.3.1 Keygen

1. *Create augmented blocks:*

- \mathcal{C} divides F into m blocks: $F = v_1 || \dots || v_m$. Each $v_k \in \mathbb{F}_q^\xi$ where $k \in \{1, \dots, m\}$.
- \mathcal{C} creates m augmented blocks as follows:

$$w_k = (v_k, \underbrace{0, \dots, 0, 1, 0, \dots, 0}_m) \in \mathbb{F}_q^{\xi+m} \quad (5.1)$$

2. *Keygen:* The key manager generates two types of keys:

- *The key of the client ($k_{\mathcal{C}}$):* $k_{\mathcal{C}} \xleftarrow{rand} \mathbb{F}_q^{\xi+m}$.
- *The one-time key of the new server every repair time (k_r):*
 - Compute $k_p \leftarrow \text{OrthogonalGen-SS}(w_1, \dots, w_m)$. The property of k_p is that $k_p \cdot w_k = 0$ where $k \in \{1, \dots, m\}$.
 - Compute the key for the new server:

$$k_r = k_{\mathcal{C}} + k_p \in \mathbb{F}_q^{\xi+m} \quad (5.2)$$

$k_{\mathcal{C}}$ is static, only k_p is re-computed every repair time.

- k_r is sent to the new server if and only if the repair phase is executed.

5.3.2 Encode

1. \mathcal{C} computes a tag for each augmented block as follows:

For $\forall k \in \{1, \dots, m\}$:

$$t_{w_k} = w_k \cdot k_{\mathcal{C}} \in \mathbb{F}_q \quad (5.3)$$

2. \mathcal{C} computes nd coded blocks and nd corresponding tags as follows:

For $\forall i \in \{1, \dots, n\}, \forall j \in \{1, \dots, d\}$:

- \mathcal{C} generates m coefficients: $\alpha_{ijk} \xleftarrow{rand} \mathbb{F}_q$ where $k \in \{1, \dots, m\}$.

- \mathcal{C} computes code block:

$$c_{ij} = \sum_{k=1}^m \alpha_{ijk} \cdot w_k \in \mathbb{F}_q^{\xi+m} \quad (5.4)$$

- \mathcal{C} computes tag:

$$t_{ij} = \sum_{k=1}^m \alpha_{ijk} \cdot t_{w_k} \in \mathbb{F}_q \quad (5.5)$$

3. \mathcal{C} sends d pairs of $\{c_{ij}, t_{ij}\}$ where $j \in \{1, \dots, d\}$ to the server \mathcal{S}_i .

5.3.3 Check

1. \mathcal{C} challenges each \mathcal{S}_i where $i \in \{1, \dots, n\}$:

- \mathcal{C} generates a challenge *chall* which consists of *spotcheck* pairs of index and coefficient: $chall = \{(j_1, \beta_1), \dots, (j_{spotcheck}, \beta_{spotcheck})\}$ where $j_{sp} \xleftarrow{rand} \{1, \dots, d\}$ and $\beta_{sp} \xleftarrow{rand} \mathbb{F}_q$ for $sp \in \{1, \dots, spotcheck\}$.
- \mathcal{C} sends *chall* to all the servers.

2. \mathcal{S}_i provides its aggregated coded block and aggregated tag as follows:

- \mathcal{S}_i combines coded blocks:

$$c_{\mathcal{S}_i} = \sum_{sp=1}^{spotcheck} \beta_{sp} \cdot c_{ij_{sp}} \in \mathbb{F}_q^{\xi+m} \quad (5.6)$$

- \mathcal{S}_i combines tags:

$$t_{\mathcal{S}_i} = \sum_{sp=1}^{spotcheck} \beta_{sp} \cdot t_{ij_{sp}} \in \mathbb{F}_q \quad (5.7)$$

- \mathcal{S}_i sends $\{c_{\mathcal{S}_i}, t_{\mathcal{S}_i}\}$ to \mathcal{C} .

3. \mathcal{C} verifies \mathcal{S}_i as follows:

For $\forall i \in \{1, \dots, n\}$:

- \mathcal{C} computes:

$$t'_{\mathcal{S}_i} = c_{\mathcal{S}_i} \cdot k_{\mathcal{C}} \in \mathbb{F}_q \quad (5.8)$$

- \mathcal{C} checks iff:

$$t_{\mathcal{S}_i} = t'_{\mathcal{S}_i} \quad (5.9)$$

If the equation 5.9 holds, \mathcal{S}_i is healthy. Otherwise, \mathcal{S}_i is corrupted.

5.3.4 Repair

Suppose \mathcal{S}_r is corrupted. A set of l healthy servers $\{\mathcal{S}_{i_1}, \dots, \mathcal{S}_{i_l}\}$ are required for repairing \mathcal{S}_r . Each of the servers $\{\mathcal{S}_{i_1}, \dots, \mathcal{S}_{i_l}\}$ provides its aggregated coded block along with its aggregated tag to the new server \mathcal{S}'_r , which is used to replace \mathcal{S}_r .

1. The new server \mathcal{S}'_r challenges the l healthy servers:

- \mathcal{S}'_r generates a challenge *chall* which consists of *spotcheck* pairs of index and coefficient: $chall = \{(j_1, \beta_1), \dots, (j_{spotcheck}, \beta_{spotcheck})\}$ where $j_{sp} \xleftarrow{rand} \{1, \dots, d\}$ and $\beta_{sp} \xleftarrow{rand} \mathbb{F}_q$ for $sp \in \{1, \dots, spotcheck\}$.
- \mathcal{S}'_r sends *chall* to all the l healthy servers.

2. \mathcal{S}_i where $i \in \{i_1, \dots, i_l\}$ provides its aggregated coded block along with its aggregated tag to \mathcal{S}'_r as follows:

- \mathcal{S}_i combines coded blocks:

$$c_{\mathcal{S}_i} = \sum_{sp=1}^{spotcheck} \beta_{sp} \cdot c_{ij_{sp}} \in \mathbb{F}_q^{\xi+m} \quad (5.10)$$

- \mathcal{S}_i combines tags:

$$t_{\mathcal{S}_i} = \sum_{sp=1}^{spotcheck} \beta_{sp} \cdot t_{ij_{sp}} \in \mathbb{F}_q \quad (5.11)$$

- \mathcal{S}_i sends $\{c_{\mathcal{S}_i}, t_{\mathcal{S}_i}\}$ to \mathcal{S}'_r .

3. \mathcal{S}'_r checks each \mathcal{S}_i where $i \in \{i_1, \dots, i_l\}$ as follows:

- \mathcal{S}'_r computes:

$$t'_{\mathcal{S}_i} = c_{\mathcal{S}_i} \cdot k_r \in \mathbb{F}_q \quad (5.12)$$

- \mathcal{S}'_r checks iff:

$$t'_{\mathcal{S}_i} = t_{\mathcal{S}_i} \quad (5.13)$$

If the equation 5.13 holds, \mathcal{S}_i is healthy. Otherwise, \mathcal{S}_i is corrupted.

4. \mathcal{S}'_r computes d new coded blocks and d corresponding tags for itself as follows:

For $\forall j \in \{1, \dots, d\}$:

- \mathcal{S}'_r generates l coefficients: $\gamma_{ri} \xleftarrow{rand} \mathbb{F}_q$ where $i \in \{i_1, \dots, i_l\}$.
- \mathcal{S}'_r computes new coded block:

$$c_{rj} = \sum_{i=i_1}^{i_l} \gamma_{ri} \cdot c_{\mathcal{S}_i} \in \mathbb{F}_q^{\xi+m} \quad (5.14)$$

- \mathcal{S}'_r computes new tag:

$$t_{rj} = \sum_{i=i_1}^{i_l} \gamma_{ri} \cdot t_{\mathcal{S}_i} \in \mathbb{F}_q \quad (5.15)$$

5. \mathcal{S}'_r stores d pairs of $\{c_{rj}, t_{rj}\}$ where $j \in \{1, \dots, d\}$:

5.4 Correctness

1. The correctness of Equation 5.9 is proven as follows:

$$\begin{aligned} t_{\mathcal{S}_i} &= \sum_{sp=1}^{spotcheck} \beta_{sp} t_{ij_{sp}} // \text{ because of Equation 5.7} \\ &= \sum_{sp=1}^{spotcheck} \beta_{sp} \left(\sum_{k=1}^m \alpha_{ij_{sp}k} t_{w_k} \right) // \text{ because of Equation 5.5} \\ &= \sum_{sp=1}^{spotcheck} \sum_{k=1}^m \beta_{sp} \alpha_{ij_{sp}k} t_{w_k} \\ &= \sum_{sp=1}^{spotcheck} \sum_{k=1}^m \beta_{sp} \alpha_{ij_{sp}k} (w_k k_{\mathcal{C}}) // \text{ because of Equation 5.3} \\ &= \sum_{sp=1}^{spotcheck} \sum_{k=1}^m \beta_{sp} \alpha_{ij_{sp}k} w_k k_{\mathcal{C}} \\ t'_{\mathcal{S}_i} &= c_{\mathcal{S}_i} \cdot k_{\mathcal{C}} // \text{ because of Equation 5.8} \\ &= \left(\sum_{sp=1}^{spotcheck} \beta_{sp} c_{ij_{sp}} \right) k_{\mathcal{C}} // \text{ because of Equation 5.6} \\ &= \sum_{sp=1}^{spotcheck} \beta_{sp} c_{ij_{sp}} k_{\mathcal{C}} \\ &= \sum_{sp=1}^{spotcheck} \beta_{sp} \left(\sum_{k=1}^m \alpha_{ij_{sp}k} w_k \right) k_{\mathcal{C}} // \text{ because of Equation 5.4} \\ &= \sum_{sp=1}^{spotcheck} \sum_{k=1}^m \beta_{sp} \alpha_{ij_{sp}k} w_k k_{\mathcal{C}} \\ &= t_{\mathcal{S}_i} \end{aligned}$$

2. The correctness of Equation 5.13 is proven as follows:

$$\begin{aligned} t_{\mathcal{S}_i} &= \sum_{sp=1}^{spotcheck} \beta_{sp} t_{ij_{sp}} // \text{ because of Equation 5.11} \\ &= \sum_{sp=1}^{spotcheck} \beta_{sp} \left(\sum_{k=1}^m \alpha_{ij_{sp}k} t_{w_k} \right) // \text{ because of Equation 5.5} \\ &= \sum_{sp=1}^{spotcheck} \sum_{k=1}^m \beta_{sp} \alpha_{ij_{sp}k} t_{w_k} \\ &= \sum_{sp=1}^{spotcheck} \sum_{k=1}^m \beta_{sp} \alpha_{ij_{sp}k} (w_k k_{\mathcal{C}}) // \text{ because of Equation 5.3} \\ &= \sum_{sp=1}^{spotcheck} \sum_{k=1}^m \beta_{sp} \alpha_{ij_{sp}k} w_k k_{\mathcal{C}} \\ t'_{\mathcal{S}_i} &= c_{\mathcal{S}_i} k_r // \text{ because of Equation 5.12} \\ &= c_{\mathcal{S}_i} (k_{\mathcal{C}} + k_p) // \text{ because of Equation 5.2} \\ &= \left(\sum_{sp=1}^{spotcheck} \beta_{sp} c_{ij_{sp}} \right) (k_{\mathcal{C}} + k_p) // \text{ because of Equation 5.10} \\ &= \sum_{sp=1}^{spotcheck} \beta_{sp} c_{ij_{sp}} (k_{\mathcal{C}} + k_p) \\ &= \sum_{sp=1}^{spotcheck} \beta_{sp} \left(\sum_{k=1}^m \alpha_{ij_{sp}k} w_k \right) (k_{\mathcal{C}} + k_p) // \text{ because of Equation 5.4} \\ &= \sum_{sp=1}^{spotcheck} \sum_{k=1}^m \beta_{sp} \alpha_{ij_{sp}k} w_k (k_{\mathcal{C}} + k_p) \end{aligned}$$

Because k_p is constructed such that $k_p \cdot w_k = 0$ for all $k \in \{1, \dots, m\}$ using the OrthogonalGen-SS, then we have:

$$\begin{aligned} t'_{\mathcal{S}_i} &= \sum_{sp=1}^{spotcheck} \sum_{k=1}^m \beta_{sp} \alpha_{ij_{sp}k} w_k k_{\mathcal{C}} \\ &= t_{\mathcal{S}_i} \end{aligned}$$

5.5 Dynamic Operations

When the client \mathcal{C} performs a dynamic operation on a file block (modification/ insertion/ deletion), herein introduces a challenge that how the servers deal with the coded blocks which are related to the modified/ inserted/ deleted file block. The trivial solution is to perform the encode phase again with the new data. This solution incurs very high computation costs. In our solution, the old coded blocks and tags stored in the servers can be re-used, and only a small additional computation is needed for the dynamic operations.

Before describing each type of dynamic operations, we give the following theorem, which will form the basis of the dynamic operations.

Theorem 12. *The basis vector of the matrix which consists of all m augmented blocks (each augmented block belongs to $\mathbb{F}_q^{\xi+m}$) is ξ .*

Proof. Let M be the matrix in which each of m augmented blocks is a row in M . Namely, M has the following form:

$$M = \begin{pmatrix} w_1 \\ w_2 \\ \vdots \\ w_m \end{pmatrix} = \underbrace{\begin{pmatrix} \overbrace{v_1 \ 1 \ 0 \ 0 \ \dots \ 0}^{\xi} \\ \overbrace{v_2 \ 0 \ 1 \ 0 \ \dots \ 0}^m \\ \vdots \\ \overbrace{v_m \ 0 \ \dots \ \dots \ 0 \ 1}^{m \times (\xi+m)} \end{pmatrix}}_{m \times (\xi+m)} \quad (5.16)$$

Because each augmented block $w_k \in \mathbb{F}_q^{\xi+m}$ (where $k \in \{1, \dots, m\}$) consists of $v_k \in \mathbb{F}_q^\xi$ and m elements in \mathbb{F}_q , the dimension of M is $m \times (\xi + m)$. Thus, we have:

- The number of pivot variables is m .
- The number free variables is $(\xi + m) - m = \xi$.

Therefore, the number of basis vectors of M is ξ . □

5.5.1 Modification

Suppose that the client \mathcal{C} modifies a file block v_X to a new file block v'_X where $X \in \{1, \dots, m\}$. Let w_X and w'_X denote the augmented block of v_X and v'_X , respectively.

1. \mathcal{C} re-computes k_r for the next repair time:

Let M be the matrix which consists of m augmented blocks. After the modification, only v_X is changed and other elements in M are unaffected. Namely, M is changed to M' as follows:

$$M = \underbrace{\begin{pmatrix} v_1 & 1 & 0 & 0 & \cdots & \cdots & \cdots & 0 \\ v_2 & 0 & 1 & 0 & \cdots & \cdots & \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ \boxed{v_X} & 0 & \underbrace{\cdots & 0}_{X} & 1 & 0 & \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ v_m & 0 & \cdots & \cdots & \cdots & \cdots & 0 & 1 \end{pmatrix}}_{m \times (\xi + m)} \quad (5.17)$$

$$M' = \underbrace{\begin{pmatrix} v_1 & 1 & 0 & 0 & \cdots & \cdots & \cdots & 0 \\ v_2 & 0 & 1 & 0 & \cdots & \cdots & \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ \boxed{v'_X} & 0 & \underbrace{\cdots & 0}_{X} & 1 & 0 & \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ v_m & 0 & \cdots & \cdots & \cdots & \cdots & 0 & 1 \end{pmatrix}}_{m \times (\xi + m)} \quad (5.18)$$

Because M is changed and k_C is not constructed from M , the modification does not affect k_C . However, because M is changed and k_p is constructed from M , the modification affects $k_r (= k_C + k_p)$. This is the reason we need to update k_r . We now describe how to update k_r as follows.

- The number of columns in M is $(\xi + m)$. The number of basis vectors of M is ξ (Theorem 12). Thus, each of these ξ basis vectors of M consists of $(\xi + m)$ elements in \mathbb{F}_q , denoted by $B_\psi = (b_1, \dots, b_{\xi+m})^T$ where $\psi \in \{1, \dots, \xi\}$. Similarly, each of the ξ basis vectors of M' also consists of $(\xi + m)$ elements in \mathbb{F}_q , denoted by $B'_\psi = (b'_1, \dots, b'_{\xi+m})^T$ where $\psi \in \{1, \dots, \xi\}$. We need to find B'_ψ from B_ψ for each $\psi \in \{1, \dots, \xi\}$.
- Because $v_X \in \mathbb{F}_q^\xi$, v_X is viewed as a set of ξ elements in \mathbb{F}_q as: $v_X = \{v_{X1}, \dots, v_{X\xi}\}$. Only $v_X \in \mathbb{F}_q^\xi$ in M is changed and other elements are not changed. For each $\psi \in \{1, \dots, \xi\}$, \mathcal{C} only needs to update the $(\xi + X)$ -th element of B_ψ by

computing $(-\sum_{\mu=1}^{\xi} v'_{X\mu} b_{\mu} \bmod q)$. Namely,

$$B'_{\psi} = \begin{pmatrix} b_1 \\ \vdots \\ b_{\xi+X-1} \\ \boxed{-\sum_{\mu=1}^{\xi} v'_{X\mu} b_{\mu} \bmod q} \\ b_{\xi+X+1} \\ \vdots \\ b_{\xi+m} \end{pmatrix} \quad (5.19)$$

- After having B'_{ψ} for all $\psi \in \{1, \dots, \xi\}$, \mathcal{C} then computes $k'_p \leftarrow \text{Kg-SS}(B'_1, \dots, B'_{\xi})$.
- \mathcal{C} finally sends $k'_r = k_{\mathcal{C}} + k'_p$ to the new server when the next repair phase is executed.

2. \mathcal{C} computes the tag of w'_k . :

- \mathcal{C} computes the tag: $t'_X = w'_X \cdot k_{\mathcal{C}} \in \mathbb{F}_q$.
- \mathcal{C} sends $\{w'_X, t'_X\}$ to each \mathcal{S}_i .

3. Each server updates its coded blocks and tags:

Because a file block $v_k \in \mathbb{F}_q^{\xi}$, v_k can be viewed as a set of ξ elements in \mathbb{F}_q as: $v_k = \{v_{k1}, \dots, v_{k\xi}\}$. An augmented block $w_k \in \mathbb{F}_q^{\xi+m}$ has the form:

$$w_k = (v_{k1}, \dots, v_{k\xi}, \underbrace{0, \dots, 0, 1, 0, \dots, 0}_k^m) \in \mathbb{F}_q^{\xi+m} \quad (5.20)$$

Because a coded block $c_{ij} = \sum_{k=1}^m \alpha_{ijk} w_k \in \mathbb{F}_q^{\xi+m}$, c_{ij} can be also viewed as a set of $(\xi + m)$ elements in \mathbb{F}_q . Let $c_{ij}[x]$ denote the x -th element of c_{ij} where $x \in \{1, \dots, \xi + m\}$:

$$c_{ij} = \begin{pmatrix} \sum_{k=1}^m \alpha_{ijk} v_{k1} \\ \vdots \\ \sum_{k=1}^m \alpha_{ijk} v_{k\xi} \\ \alpha_{ij1} \\ \vdots \\ \alpha_{ijm} \end{pmatrix}^T = \begin{pmatrix} c_{ij}[1] \\ \vdots \\ c_{ij}[\xi] \\ c_{ij}[\xi + 1] \\ \vdots \\ c_{ij}[\xi + m] \end{pmatrix}^T \quad (5.21)$$

- \mathcal{S}_i updates coded blocks:

For $\forall j \in \{1, \dots, d\}$, the new coded block is computed as:

$$c'_{ij} = \begin{pmatrix} c_{ij}[1] + \alpha_{ijX}(v'_{X1} - v_{X1}) \\ \vdots \\ c_{ij}[\xi] + \alpha_{ijX}(v'_{X\xi} - v_{X\xi}) \\ c_{ij}[\xi + 1] \\ \vdots \\ c_{ij}[\xi + m] \end{pmatrix}^T \quad (5.22)$$

- \mathcal{S}_i updates tags:

For $\forall j \in \{1, \dots, d\}$, the new tag is computed as:

$$t'_{ij} = t_{ij} + \alpha_{ijX}(t'_X - t_X) \quad (5.23)$$

where c_{ij} and t_{ij} are the old coded block and tag. The coefficient α_{ijX} can be found at the $(\xi + X)$ -th element of c_{ij} .

The modification only needs $O(\xi)$ for recomputing k_r , $O(1)$ for computing tag for w'_k and $O(\xi)$ for updating a coded block and a tag.

5.5.2 Insertion

Suppose that the client \mathcal{C} inserts a file block v_I after the existing file block v_X where $X \in \{1, \dots, m\}$. Let w_I denote the augmented block of v_I .

1. \mathcal{C} modifies $k_{\mathcal{C}}$:

Before the insertion, an augmented block has $(\xi + m)$ elements in \mathbb{F}_q :

$$w_k = (v_{k1}, \dots, v_{k\xi}, \underbrace{0, \dots, 0, 1, 0, \dots, 0}_k^m) \in \mathbb{F}_q^{\xi+m} \quad (5.24)$$

Thus, $k_{\mathcal{C}}$ has $(\xi + m)$ elements in \mathbb{F}_q (says, $k_{\mathcal{C}} = (k_1, \dots, k_{\xi+m})^T$). After the insertion, an augmented block has $(\xi + m + 1)$ elements in \mathbb{F}_q :

$$w'_k = (v_{k1}, \dots, v_{k\xi}, \underbrace{0, \dots, 0, 1, 0, \dots, 0}_k^{m+1}) \in \mathbb{F}_q^{\xi+m+1} \quad (5.25)$$

Thus, the new $k'_{\mathcal{C}}$ also has $(\xi + m + 1)$ elements in \mathbb{F}_q (says, $k'_{\mathcal{C}} = (k'_1, \dots, k'_{\xi+m+1})^T$). Given $k_{\mathcal{C}}$, we find $k'_{\mathcal{C}}$ as follows:

- The first $(\xi + X)$ elements of $k'_{\mathcal{C}}$ are the same as the first $(\xi + X)$ elements of $k_{\mathcal{C}}$.

- The $(\xi + X + 1)$ -th element of k'_C (denoted by k_I) is computed as: $k_I \xleftarrow{rand} \mathbb{F}_q$.
- The last $(m - \xi - X)$ elements of k'_C are the same as the last $(m - \xi - X)$ elements of k_C .

Namely:

$$k'_C = (k_1, \dots, k_{\xi+X}, \boxed{k_I}, k_{\xi+X+1}, \dots, k_{\xi+m})^T \quad (5.26)$$

The reason that we construct such k'_C will be explained in Step 3 (tag update).

2. \mathcal{C} re-computes k_r for the next repair time:

After the insertion, the matrix M is changed as follows:

- In each of the first X rows: a '0' bit is padded in the final position.
- In the inserted row (w_I): v_I is placed in the first ξ elements, a '1' bit is placed at the $(\xi + X + 1)$ -th element counted from the left, and '0' bits are placed elsewhere.
- In each of the last $(m - X)$ rows: a '0' bit is padded in the final position and then, the '1' bit is shipped to the next right position.

$$M = \underbrace{\begin{pmatrix} v_1 & 1 & 0 & \dots & \dots & \dots & \dots & \dots & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ v_X & 0 & \dots & 0 & 1 & 0 & \dots & \dots & 0 \\ v_{X+1} & 0 & \dots & \underbrace{0 \quad 0}_{X} & 1 & 0 & \dots & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ v_m & 0 & \dots & \dots & \dots & \dots & \dots & 0 & 1 \end{pmatrix}}_{m \times (\xi+m)} \quad (5.27)$$

$$M' = \underbrace{\begin{pmatrix} v_1 & 1 & 0 & \dots & \dots & \dots & \dots & \dots & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ v_X & 0 & \dots & 0 & 1 & 0 & \dots & \dots & 0 \\ \boxed{v_I} & 0 & \dots & \underbrace{0 \quad 0}_{X} & 1 & 0 & \dots & \dots & 0 \\ v_{X+1} & 0 & \dots & \underbrace{0 \quad 0 \quad 0}_{X+1} & 1 & 0 & \dots & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ v_m & 0 & \dots & \dots & \dots & \dots & \dots & 0 & 1 \end{pmatrix}}_{(m+1) \times (\xi+m+1)} \quad (5.28)$$

We now update k'_r as follows:

- Let $B_\psi = (b_1, \dots, b_{\xi+m})^T$ and $B'_\psi = (b'_1, \dots, b'_{\xi+m+1})^T$ be the ξ basis vectors of M and M' , respectively. Given B_ψ , we firstly find B'_ψ as follows:
 - The first $(\xi + X)$ elements of B'_ψ are the same as the first $(\xi + X)$ elements of B_ψ :

$$\begin{cases} b'_1 = b_1 \\ \vdots \\ b'_{\xi+X} = b_{\xi+X} \end{cases} \quad (5.29)$$

- The $(\xi + X + 1)$ -th elements of B'_ψ is computed as:

$$b'_{\xi+X+1} = \left(- \sum_{\mu=1}^{\xi} v_{I\mu} b_\mu \right) \mod q \quad (5.30)$$

- The last $(m - X)$ elements of B'_ψ are simply computed as:

$$b'_t = b_{t-1} \quad \text{where } t \in \{\xi + X + 2, \dots, \xi + m + 1\} \quad (5.31)$$

In other words:

$$B'_\psi = (b_1, \dots, b_{\xi+X}, \boxed{- \sum_{\mu=1}^{\xi} v_{I\mu} b_\mu \mod q}, b_{\xi+X+2}, \dots, b_{\xi+m+1})^T \quad (5.32)$$

- After having B'_ψ for all $\psi \in \{1, \dots, \xi\}$, \mathcal{C} then computes $k'_p \leftarrow \text{Kg-SS}(B'_1, \dots, B'_\xi)$.
- \mathcal{C} finally sends $k'_r = k'_\mathcal{C} + k'_p$ to the new server when the next repair phase is executed.

3. \mathcal{C} computes a tag for w_I :

- \mathcal{C} computes a tag for w_I as: $t_I = w_I \cdot k'_\mathcal{C}$.
- \mathcal{C} sends $\{w_I, t_I\}$ to \mathcal{S}_i .

4. Each server \mathcal{S}_i updates its coded blocks and tags:

Because a file block $v_k \in \mathbb{F}_q^\xi$, v_k can be viewed as a set of ξ elements in \mathbb{F}_q as: $v_k = \{v_{k1}, \dots, v_{k\xi}\}$. An augmented block $w_k \in \mathbb{F}_q^{\xi+m}$ has the form:

$$w_k = (v_{k1}, \dots, v_{k\xi}, \underbrace{0, \dots, 0, 1, 0, \dots, 0}_k) \in \mathbb{F}_q^{\xi+m} \quad (5.33)$$

Because a coded block $c_{ij} = \sum_{k=1}^m \alpha_{ijk} w_k \in \mathbb{F}_q^{\xi+m}$, c_{ij} can be also viewed as a set of $(\xi + m)$ elements in \mathbb{F}_q . Let $c_{ij}[x]$ denote the x -th element of c_{ij} where

$x \in \{1, \dots, \xi + m\}$:

$$c_{ij} = \begin{pmatrix} \sum_{k=1}^m \alpha_{ijk} v_{k1} \\ \vdots \\ \sum_{k=1}^m \alpha_{ijk} v_{k\xi} \\ \alpha_{ij1} \\ \vdots \\ \alpha_{ijm} \end{pmatrix}^T = \begin{pmatrix} c_{ij}[1] \\ \vdots \\ c_{ij}[\xi] \\ c_{ij}[\xi + 1] \\ \vdots \\ c_{ij}[\xi + m] \end{pmatrix}^T \quad (5.34)$$

- \mathcal{S}_i updates its coded blocks as follows:

$$c'_{ij} = \begin{pmatrix} \sum_{k=1}^m \alpha_{ijk} v_{k1} + \alpha_{ijI} v_{I1} \\ \vdots \\ \sum_{k=1}^m \alpha_{ijk} v_{k\xi} + \alpha_{ijI} v_{I\xi} \\ \alpha_{ij1} \\ \vdots \\ \alpha_{ijX} \\ \boxed{\alpha_{ijI}} \\ \alpha_{ij(X+1)} \\ \vdots \\ \alpha_{ijm} \end{pmatrix}^T = \begin{pmatrix} c_{ij}[1] + \alpha_{ijI} v_{I1} \\ \vdots \\ c_{ij}[\xi] + \alpha_{ijI} v_{I\xi} \\ c_{ij}[\xi + 1] \\ \vdots \\ c_{ij}[\xi + X] \\ \boxed{\alpha_{ijI}} \\ c_{ij}[\xi + X + 1] \\ \vdots \\ c_{ij}[\xi + m] \end{pmatrix}^T \quad (5.35)$$

where $\alpha_{ijI} \xleftarrow{rand} \mathbb{F}_q$.

- \mathcal{S}_i updates its tags:

The tags of the augmented blocks before the insertion are:

$$\begin{aligned}
 \begin{pmatrix} t_{w_1} \\ \vdots \\ t_{w_X} \\ t_{w_{(X+1)}} \\ \vdots \\ t_{w_m} \end{pmatrix} &= M \cdot k_{\mathcal{C}} \\
 &= \begin{pmatrix} v_1 & 1 & 0 & \cdots & \cdots & \cdots & \cdots & \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ v_X & 0 & \underbrace{\cdots 0 1}_X & 0 & \cdots & \cdots & \cdots & \cdots & 0 \\ v_{X+1} & 0 & \underbrace{\cdots 0 0}_{X+1} & 1 & 0 & \cdots & \cdots & \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ v_m & 0 & \cdots & \cdots & \cdots & \cdots & \cdots & 0 & 1 \end{pmatrix} \begin{pmatrix} k_1 \\ \vdots \\ k_{\xi+1} \\ \vdots \\ k_{(\xi+X)} \\ k_{(\xi+X+1)} \\ \vdots \\ k_{\xi+m} \end{pmatrix} \quad (5.36) \\
 &= \begin{pmatrix} v_{11}k_1 + \cdots + v_{1\xi}k_{\xi} + k_{\xi+1} \\ \vdots \\ v_{X1}k_1 + \cdots + v_{X\xi}k_{\xi} + k_{\xi+X} \\ v_{(X+1)1}k_1 + \cdots + v_{(X+1)\xi}k_{\xi} + k_{\xi+X+1} \\ \vdots \\ v_{m1}k_1 + \cdots + v_{m\xi}k_{\xi} + k_{\xi+m} \end{pmatrix}
 \end{aligned}$$

By constructing $k'_{\mathcal{C}}$ in Step 1, the tags of the augmented blocks after the insertion are:

$$\begin{aligned}
 \begin{pmatrix} t'_{w_1} \\ \vdots \\ t'_{w_X} \\ \boxed{t_I} \\ t'_{w_{(X+1)}} \\ \vdots \\ t'_{w_{m+1}} \end{pmatrix} &= M' \cdot k'_c \\
 &= \begin{pmatrix} v_1 & 1 & 0 & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ v_X & 0 & \cdots & 0 & 1 & 0 & \cdots & \cdots & \cdots & 0 \\ \boxed{v_I} & 0 & \cdots & 0 & 0 & 1 & 0 & \cdots & \cdots & 0 \\ v_{X+1} & 0 & \cdots & 0 & 0 & 0 & 1 & 0 & \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ v_m & 0 & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & 0 & 1 \end{pmatrix} \begin{pmatrix} k_1 \\ \vdots \\ k_\xi \\ k_{\xi+1} \\ \vdots \\ k_{(\xi+X)} \\ \boxed{k_I} \\ k_{(\xi+X+1)} \\ \vdots \\ k_{\xi+m} \end{pmatrix} \\
 &= \begin{pmatrix} v_{11}k_1 + \cdots + v_{1\xi}k_\xi + k_{\xi+1} \\ \vdots \\ v_{X1}k_1 + \cdots + v_{X\xi}k_\xi + k_{\xi+X} \\ v_{I1}k_1 + \cdots + v_{I\xi}k_\xi + k_I \\ v_{(X+1)1}k_1 + \cdots + v_{(X+1)\xi}k_\xi + k_{\xi+X+1} \\ \vdots \\ v_{m1}k_1 + \cdots + v_{m\xi}k_\xi + k_{\xi+m} \end{pmatrix}
 \end{aligned} \tag{5.37}$$

We can observe that before and after the insertion, the first (X) tags and the last $(m - X)$ tags are not changed; only a new tag t_I , which is the tag of w_I , is inserted. Furthermore, the old tag of c_{ij} is computed as $t_{ij} = \sum_{k=1}^m \alpha_{ijk} \cdot t_{w_k}$. We are now ready to compute the tag for c'_{ij} as follows:

$$t_{c'_{ij}} = \sum_{k=1}^X \alpha_{ijk} t_{w_k} + \alpha_{ijI} t_I + \sum_{k=X+1}^m \alpha_{ijk} t_{w_k} = t_{ij} + \boxed{\alpha_{ijI} t_I} \tag{5.38}$$

where α_{ijI} is the same as in Equation 5.35.

The insertion only needs $O(1)$ for recomputing k_c , $O(\xi)$ for recomputing k_r , $O(1)$ for computing tag for w_I and $O(\xi)$ for updating a coded block and a tag.

5.5.3 Deletion

Suppose that the client \mathcal{C} deletes the X -th file block (v_X). Let w_X denote the augmented block of v_X .

1. \mathcal{C} modifies $k_{\mathcal{C}}$:

Similar to the insertion operation, before the deletion, the key of \mathcal{C} has the form: $k_{\mathcal{C}} = (k_1, \dots, k_{\xi+m})^T$. After the deletion, \mathcal{C} simply removes the $(\xi + X)$ -th element in $k_{\mathcal{C}}$. Namely,

$$k'_{\mathcal{C}} = (k_1, \dots, k_{\xi+X-1}, k_{\xi+X+1}, \dots, k_{\xi+m})^T \quad (5.39)$$

The reason to construct such $k'_{\mathcal{C}}$ will be explained in Step 3 (tag update).

2. \mathcal{C} recomputes k_r for the next repair time:

After the deletion, the matrix M is changed as follows:

- In each of the first $(X - 1)$ rows, the ‘0’ bit at the final position is removed.
- The X -th row is removed.
- In each of the last $(m - X)$ rows, the ‘1’ bit is shipped to the previous left position and then, the ‘0’ bit at the final position is removed.

$$M = \underbrace{\begin{pmatrix} v_1 & 1 & 0 & \dots & \dots & \dots & \dots & \dots & \dots & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ v_{X-1} & 0 & \dots & 0 & 1 & 0 & \dots & \dots & \dots & 0 \\ \boxed{v_X} & 0 & \dots & 0 & 0 & 1 & 0 & \dots & \dots & 0 \\ v_{X+1} & 0 & \dots & 0 & 0 & 0 & 1 & 0 & \dots & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ v_m & 0 & \dots & \dots & \dots & \dots & \dots & \dots & 0 & 1 \end{pmatrix}}_{m \times (\xi+m)} \quad (5.40)$$

$$M' = \underbrace{\begin{pmatrix} v_1 & 1 & 0 & \cdots & \cdots & \cdots & \cdots & \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ v_{X-1} & 0 & \cdots & 0 & 1 & 0 & \cdots & \cdots & 0 \\ v_{X+1} & 0 & \cdots & \underbrace{0 \quad 0}_{X-1} & 1 & 0 & \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ v_m & 0 & \cdots & \cdots & \cdots & \cdots & \cdots & 0 & 1 \end{pmatrix}}_{(m-1) \times (\xi+m-1)} \quad (5.41)$$

We now update k'_r as follows:

- Let $B_\psi = (b_1, \dots, b_{\xi+m})^T$ and $B'_\psi = (b_1, \dots, b_{\xi+m-1})^T$ where $\psi \in \{1, \dots, \xi\}$ be the ξ basis vectors of M and M' , respectively.
- To compute B'_ψ from B_ψ , \mathcal{C} simply removes the $(\xi + X + 1)$ -th element of B_ψ . Namely,

$$B'_\psi = (b_1, \dots, b_X, b_{X+2}, \dots, b_{\xi+m})^T \quad (5.42)$$

- After having B'_ψ for all $\psi \in \{1, \dots, \xi\}$, \mathcal{C} computes k'_p as:

$$k'_p \leftarrow \text{Kg-SS}(B'_1, \dots, B'_\xi) \quad (5.43)$$

- \mathcal{C} finally sends $k'_r = k'_c + k'_p$ to the new server when the next repair phase is executed.

3. \mathcal{S}_i updates its coded blocks and tags:

Because a file block $v_k \in \mathbb{F}_q^\xi$, v_k can be viewed as a set of ξ elements in \mathbb{F}_q as: $v_k = \{v_{k1}, \dots, v_{k\xi}\}$. An augmented block $w_k \in \mathbb{F}_q^{\xi+m}$ has the form:

$$w_k = (v_{k1}, \dots, v_{k\xi}, \underbrace{0, \dots, 0, 1, 0, \dots, 0}_m) \in \mathbb{F}_q^{\xi+m} \quad (5.44)$$

Because a coded block $c_{ij} = \sum_{k=1}^m \alpha_{ijk} w_k \in \mathbb{F}_q^{\xi+m}$, c_{ij} can be also viewed as a set of $(\xi + m)$ elements in \mathbb{F}_q . Let $c_{ij}[x]$ denote the x -th element of c_{ij} where $x \in \{1, \dots, \xi + m\}$:

$$c_{ij} = \begin{pmatrix} \sum_{k=1}^m \alpha_{ijk} v_{k1} \\ \vdots \\ \sum_{k=1}^m \alpha_{ijk} v_{k\xi} \\ \alpha_{ij1} \\ \vdots \\ \alpha_{ijm} \end{pmatrix}^T = \begin{pmatrix} c_{ij}[1] \\ \vdots \\ c_{ij}[\xi] \\ c_{ij}[\xi + 1] \\ \vdots \\ c_{ij}[\xi + m] \end{pmatrix}^T \quad (5.45)$$

- \mathcal{S}_i updates its coded blocks:

$$c'_{ij} = \begin{pmatrix} \sum_{k=1}^m \alpha_{ijk} v_{k1} - \alpha_{ijX} v_{X1} \\ \vdots \\ \sum_{k=1}^m \alpha_{ijk} v_{k\xi} - \alpha_{ijX} v_{X\xi} \\ \alpha_{ij1} \\ \vdots \\ \alpha_{ij(X-1)} \\ \alpha_{ij(X+1)} \\ \vdots \\ \alpha_{ijm} \end{pmatrix}^T = \begin{pmatrix} c_{ij}[1] - \alpha_{ijX} v_{X1} \\ \vdots \\ c_{ij}[\xi] - \alpha_{ijX} v_{X\xi} \\ c_{ij}[\xi + 1] \\ \vdots \\ \alpha_{ij}[\xi + X - 1] \\ \alpha_{ij}[\xi + X + 1] \\ \vdots \\ \alpha_{ij}[\xi + m] \end{pmatrix}^T \quad (5.46)$$

where $\alpha_{ijX} = c_{ij}[\xi + X]$.

- \mathcal{S}_i updates its tags as follows:
The tags of the augmented blocks before the deletion are:

$$\begin{aligned}
 \begin{pmatrix} t_{w_1} \\ \vdots \\ t_{w_{X-1}} \\ t_{w_X} \\ t_{w_{(X+1)}} \\ \vdots \\ t_{w_m} \end{pmatrix} &= M \cdot k_{\mathcal{C}} \\
 &= \begin{pmatrix} v_1 & 1 & 0 & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ v_{X-1} & 0 & \cdots & \underbrace{0 \quad 1}_{X-1} & 0 & \cdots & \cdots & \cdots & 0 \\ \boxed{v_X} & 0 & \cdots & \underbrace{0 \quad 0}_{X} & 1 & 0 & \cdots & \cdots & 0 \\ v_{X+1} & 0 & \cdots & \underbrace{0 \quad 0 \quad 0}_{X+1} & 1 & 0 & \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ v_m & 0 & \cdots & \cdots & \cdots & \cdots & \cdots & 0 & 1 \end{pmatrix} \begin{pmatrix} k_1 \\ \vdots \\ k_{\xi} \\ k_{\xi+1} \\ \vdots \\ k_{\xi+m} \end{pmatrix} \\
 &= \begin{pmatrix} v_{11}k_1 + \cdots + v_{1\xi}k_{\xi} + k_{\xi+1} \\ \vdots \\ v_{(X-1)1}k_1 + \cdots + v_{(X-1)\xi}k_{\xi} + k_{\xi+X-1} \\ v_{X1}k_1 + \cdots + v_{X\xi}k_{\xi} + k_{\xi+X} \\ v_{(X+1)1}k_1 + \cdots + v_{(X+1)\xi}k_{\xi} + k_{\xi+X+1} \\ \vdots \\ v_{m1}k_1 + \cdots + v_{m\xi}k_{\xi} + k_{\xi+m} \end{pmatrix}
 \end{aligned} \tag{5.47}$$

By constructing $k'_{\mathcal{C}}$ as Step 1, the tags of all augmented blocks after the deletion are:

$$\begin{aligned}
 \begin{pmatrix} t'_{w_1} \\ \vdots \\ t'_{w_{X-1}} \\ t'_{w_{(X+1)}} \\ \vdots \\ t'_{w_{m+1}} \end{pmatrix} &= M' \cdot k'_C \\
 &= \begin{pmatrix} v_1 & 1 & 0 & \cdots & \cdots & \cdots & \cdots & \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ v_{X-1} & 0 & \cdots & 0 & 1 & 0 & \cdots & \cdots & 0 \\ v_{X+1} & 0 & \cdots & \underbrace{0 \quad \cdots \quad 0}_{X-1} & 0 & 1 & 0 & \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ v_m & 0 & \cdots & \underbrace{\cdots \quad \cdots \quad \cdots}_X & \cdots & \cdots & \cdots & 0 & 1 \end{pmatrix} \begin{pmatrix} k_1 \\ \vdots \\ k_\xi \\ k_{\xi+1} \\ \vdots \\ k_{\xi+X-1} \\ k_{\xi+X+1} \\ \vdots \\ k_{\xi+m} \end{pmatrix} \quad (5.48) \\
 &= \begin{pmatrix} v_{11}k_1 + \cdots + v_{1\xi}k_\xi + k_{\xi+1} \\ \vdots \\ v_{(X-1)1}k_1 + \cdots + v_{(X-1)\xi}k_\xi + k_{\xi+X-1} \\ v_{(X+1)1}k_1 + \cdots + v_{(X+1)\xi}k_\xi + k_{\xi+X+1} \\ \vdots \\ v_{m1}k_1 + \cdots + v_{m\xi}k_\xi + k_{\xi+m} \end{pmatrix}
 \end{aligned}$$

We observe that before and after the deletion, the X -th tag is removed and the other tags are still the same. The old tag of c_{ij} is computed as: $t_{ij} = \sum_{k=1}^m \alpha_{ijk} t_{w_k}$. We compute the tag for c'_{ij} as follows:

$$t_{c'_{ij}} = \sum_{k=1}^{X-1} \alpha_{ijk} t_{w_k} + \sum_{k=X+1}^m \alpha_{ijk} t_{w_k} = t_{ij} - \boxed{\alpha_{ijX} t_X} \quad (5.49)$$

where α_{ijX} is the same as in Equation 5.46.

The deletion only needs $O(1)$ for recomputing k_C , $O(\xi)$ for recomputing k_r , $O(\xi)$ for updating a coded block and tag.

5.6 Security Analysis

5.6.1 Pollution Attack

We show that our scheme is secure from the pollution attack via the following theorem.

Theorem 13. *The DD-POR is secured from the pollution attack.*

Proof. In the check phase, the server \mathcal{S}_r is detected as a corrupted server. Then, a set of l servers $\mathcal{S}_{i_1}, \dots, \mathcal{S}_{i_l}$ are required to provide their responses which consist of aggregated coded blocks (as in Equation 5.10) and aggregated tags (as in Equation 5.11) to the new server \mathcal{S}'_r for repairing \mathcal{S}_r . Suppose that \mathcal{S}_x , which is a server in the set of the l servers $\mathcal{S}_{i_1}, \dots, \mathcal{S}_{i_l}$, is the malicious server which performs the pollution attack. Instead of sending the valid pair of aggregated coded block and aggregated tag $\{c_x, t_x\}$ to the new server \mathcal{S}'_r , \mathcal{S}_x sends a pair of forged coded block and forged tag (c''_x, t''_x) to the new server \mathcal{S}'_r .

The key idea here is that \mathcal{S}'_r always checks each aggregated coded block which is provided from each of the servers $\mathcal{S}_{i_1}, \dots, \mathcal{S}_{i_l}$. Although \mathcal{S}_x already passed the check phase, \mathcal{S}_x must be checked again by \mathcal{S}'_r in the repair phase before \mathcal{S}'_r uses the aggregated coded block of \mathcal{S}_x for repairing \mathcal{S}_r . Namely, we analyse the probability of \mathcal{S}_x as follows. (c''_x, t''_x) holds the verification $t''_x = c''_x \cdot \kappa'$ if \mathcal{S}_x can obtain k_r because the new server \mathcal{S}'_r is assumed to not collude with the other servers and k_r is sent to \mathcal{S}'_r via a secure channel.

- Using the brute force search: the probability to find k_r is $\frac{1}{q^{\xi+m}}$ because $k_r \in \mathbb{F}_q^{\xi+m}$. Formally:

$$\Pr_{\text{BruteForce}}[\mathcal{S}_x \rightarrow k_r] = \frac{1}{q^{\xi+m}} \quad (5.50)$$

- Using the access to the KeyGen algorithm: because $k_r = k_c + k_p$, the problem to find k_r now becomes the problem to find k_c and k_p in the OrthogonalGen-SS algorithm. Because $k_c \xleftarrow{\text{rand}} \mathbb{F}_q^{\xi+m}$, the probability to find k_c is:

$$\Pr_{\text{KeyGen}}[\mathcal{S}_x \rightarrow k_c] = \frac{1}{q^{(\xi+m)}} \quad (5.51)$$

k_p is computed as:

$$k_p \leftarrow \text{OrthogonalGen-SS}(w_1, \dots, w_m)$$

In the OrthogonalGen-SS algorithm, after finding the basis vectors B_1, \dots, B_ξ , k_p is computed as:

$$\begin{aligned} - r_x &\leftarrow f(k_{PRF}, x) \in \mathbb{F}_q, \forall x \in \{1, \dots, \xi\}. \\ - k_p &\leftarrow \sum_{x=1}^{\xi} r_x \cdot B_x \in \mathbb{F}_q^{\xi+m}. \end{aligned}$$

where f is a pseudo-random function. The probability to find each r_x is $(\Pr[f] + \frac{1}{q})$. The probability to find all r_1, \dots, r_ξ is $(\Pr[f] + \frac{1}{q^\xi})$. It is not $(\xi \Pr[f] + \frac{1}{q^\xi})$ because $\Pr[f]$ can be re-used for finding other r_i . This is also the probability to find k_p . Formally:

$$\Pr_{\text{KeyGen}}[\mathcal{S}_x \rightarrow k_p] = \Pr[f] + \frac{1}{q^\xi} \quad (5.52)$$

From Equation 5.51 and Equation 5.52, the probability for \mathcal{S}_x to find k_r using the access to KeyGen algorithm is:

$$\begin{aligned}
\Pr_{\text{KeyGen}}[\mathcal{S}_x \rightarrow k_r] &= \Pr_{\text{KeyGen}}[\mathcal{S}_x \rightarrow k_c] + \Pr_{\text{KeyGen}}[\mathcal{S}_x \rightarrow k_p] \\
&= \frac{1}{q^{(\xi+m)}} + \Pr[f] + \frac{1}{q^\xi}
\end{aligned} \tag{5.53}$$

From Equation 5.50 and Equation 5.53, the probability for \mathcal{S}_x to pass the verification of the new server \mathcal{S}'_r in the repair phase is as follows:

$$\begin{aligned}
\Pr[\mathcal{S}_x \rightarrow \text{verify}(\text{RepairPhase}) = 1] &= \Pr_{\text{BruteForce}}[\mathcal{S}_x \rightarrow k_r] + \Pr_{\text{KeyGen}}[\mathcal{S}_x \rightarrow k_r] \\
&= 2 \frac{1}{q^{\xi+m}} + \Pr[f] + \frac{1}{q^\xi}
\end{aligned} \tag{5.54}$$

If the pseudo-random function f is unforgeable and q is chosen large enough (e.g., 160 bits), the probability for \mathcal{S}_x to pass the verification of the new server \mathcal{S}'_r in the repair phase is negligible.

We also consider that the \mathcal{S}'_r itself is a malicious server who will perform the pollution attack in the next epoch. Even though \mathcal{S}'_r holds k_r , \mathcal{S}'_r cannot pass the verification in the repair phase because k_r is a one-time repair key. Another new server will be given a key $k'_r \neq k_r$. \square

5.6.2 Curious Attack

We also show that our scheme is secure from the curious attack via the following theorem.

Theorem 14. *The DD-POR is secured from the curious attack.*

Proof. The new server is given the key $k_r = k_c + k_p \in \mathbb{F}_q^{\xi+m}$.

- Via the brute force search: the probability of the new server to find k_c is $1/q^{\xi+m}$. This probability is from searching k_c directly or searching k_p and then obtaining k_c by $k_c = k_r - k_p$. If q is chosen large enough (e.g, 160 bits), the probability is $1/(2^{160})^{\xi+m}$, which is negligible.
- Via the access to the **Keygen** algorithm: the probability of the new server to find k_c from learning k_p and then obtaining k_c by $k_c = k_r - k_p$ is: $\Pr[f] + \frac{1}{q^\xi}$. If q is chosen large enough and f is unforgeable, the probability is negligible.

\square

5.6.3 File reconstruction condition

We show the condition to reconstruct F via the following theorem.

Theorem 15. *The original file F can be reconstructed if in any epoch, at least l out of n servers collectively store m coded blocks which are linearly independent combinations of m augmented blocks, and the matrix consisting of the accumulated coefficients has full rank (i.e, the rank is m).*

Proof. \mathcal{S}_i contains d coded blocks: c_{ij} where $j \in \{1, \dots, d\}$. c_{ij} is computed from m augmented blocks w_1, \dots, w_m by $c_{ij} = \sum_{k=1}^m \alpha_{ijk} w_k \in \mathbb{F}_q^{\xi+m}$. Therefore, to reconstruct F , m augmented blocks are viewed as the unknowns that need to be solved. To solve these unknowns, at least m coded blocks are required such that the coefficient matrix has full rank.

$$\begin{cases} c_{(ij)_1} = \sum_{k=1}^m \alpha_{(ijk)_1} \cdot w_k \\ c_{(ij)_2} = \sum_{k=1}^m \alpha_{(ijk)_2} \cdot w_k \\ \dots \\ c_{(ij)_m} = \sum_{k=1}^m \alpha_{(ijk)_m} \cdot w_k \end{cases} \quad (5.55)$$

Let l be the number of servers ($l < n$) which collectively stores these m coded blocks. Because each server stores d coded blocks, $l = \lceil \frac{m}{d} \rceil$. \square

5.7 Efficiency Analysis

The feature and efficiency comparison between the DD-POR scheme and the previous scheme (RDC-NC, MD-POR, NC-Audit) is depicted in Table 5.2. Because the MD-POR and NC-Audit schemes focus on the public authentication, the system models have one more entity called TPA (Third Party Auditor) who is delegated the task of checking the servers by the client \mathcal{C} . For the fair comparison, we assume that the check task in these schemes is performed by the client \mathcal{C} .

5.7.1 Encode Computation

- In all the schemes, \mathcal{C} needs $O(m)$ to compute m tags for m augmented blocks, and $O(mnd)$ to compute nd coded blocks along with the tags. The complexity on the client-side is thus $O(mnd)$.
- Meanwhile, the servers only need to receive the coded blocks and tags from \mathcal{C} without any computation. The complexity on the server-side is thus $O(1)$.

5.7.2 Check Computation

- In all the schemes, \mathcal{C} needs $O(1)$ to verify the aggregated coded block and tag of each server. Therefore, the complexity on the client-side is $O(n)$ to verify n servers.
- Meanwhile, each server needs to combine *spotcheck* coded blocks and *spotcheck* tags to compute the aggregated coded block and aggregated tag, respectively where *spotcheck* $\in \{1, \dots, d\}$. Therefore, the complexity of n servers is $O(dn)$.

5.7.3 Repair Computation

- In the RDC-NC scheme, \mathcal{C} needs $O(l)$ to check l pairs of the aggregated coded block and aggregated tag from l healthy servers, and needs $O(dl)$ to compute d

Table 5.2: Efficiency comparison between the DD-POR and previous schemes

Feature		RDC-NC [61]	MD-POR	NC-Audit [62]	DD-POR
Encode Computation	Direct repair	No	Yes	Not completed (*)	Yes
	Dynamic operations	No	No	No	Yes
	Symmetric key	Yes	Yes	Yes	Yes
Check Computation	Client-side	$O(mnd)$	$O(mnd)$	$O(mnd)$	$O(mnd)$
	Server-side	$O(1)$	$O(1)$	$O(1)$	$O(1)$
Repair Computation	Client-side	$O(n)$	$O(n)$	$O(n)$	$O(n)$
	Server-side	$O(dn)$	$O(dn)$	$O(dn)$	$O(dn)$
Modification Computation	Client-side	$O(dl)$	$O(1)$	$O(1)$	$O(1)$
	Server-side	$O(dl)$	$O(dl)$	$O(dl)$	$O(dl)$
Insertion Computation	Client-side	N/A	N/A	N/A	$O(\xi)$
	Server-side	N/A	N/A	N/A	$O(dn\xi)$
Deletion Computation	Client-side	N/A	N/A	N/A	$O(\xi)$
	Server-side	N/A	N/A	N/A	$O(dn\xi)$

N/A means not applicable due to the lack of support. (*) In the NC-Audit, the direct repair can lead to the pollution attack because the new server cannot check the provided coded blocks.

pairs of new coded blocks and new tags using the linear combinations of l pairs of the provided coded blocks and tags. Therefore, the complexity on the client-side is $O(dl)$. In the MD-POR, NC-Audit and DD-POR schemes, the complexity on the client-side is $O(1)$ because \mathcal{C} does not need to do anything due to the direct repair feature.

- In the RDC-NC scheme, each of l servers combines its *spotcheck* coded blocks and *spotcheck* tags to compute the aggregated coded block and aggregated tag, respectively where *spotcheck* $\in \{1, \dots, d\}$. Therefore, the complexity on the server-side is $O(dl)$. In the MD-POR, NC-Audit and DD-POR schemes, l healthy servers perform as in the RDC-NC ($O(dl)$), and the new server performs the task of \mathcal{C} as in the RDC-NC ($O(dl)$). Therefore, the complexity on the server-side is $O(dl)$.

5.7.4 Modification Computation

- In the DD-POR, \mathcal{C} only needs $O(\xi)$ to recompute k_r (Step 1), and $O(1)$ to compute the new tag of the modified augmented block (Step 2). Therefore, the complexity on the client-side is $O(\xi)$.
- Meanwhile, each server needs $O(d\xi)$ to update the coded blocks and tags (Step 3). Therefore, the complexity of n servers is $O(dn\xi)$.

5.7.5 Insertion Computation

- In the DD-POR, \mathcal{C} only needs $O(1)$ to recompute $k_{\mathcal{C}}$ (Step 1), $O(\xi)$ to recompute k_r (Step 2), and $O(1)$ to compute the tag of the inserted augmented block (Step 3). Therefore, the complexity on the client-side is $O(\xi)$.
- Meanwhile each server needs $O(d\xi)$ to update the coded blocks and tags (Step 4). Therefore, the complexity of n servers is $O(dn\xi)$.

5.7.6 Deletion Computation

- In the DD-POR, \mathcal{C} only needs $O(1)$ to recompute $k_{\mathcal{C}}$ (Step 1), and $O(\xi)$ to recompute k_r (Step 2). Thus, the complexity on the client-side is $O(\xi)$.
- Meanwhile each server needs $O(d\xi)$ to update the coded blocks and tags (Step 3). Thus, the complexity of n servers is $O(dn\xi)$.

5.8 Numeric Example

Suppose $m = 3, \xi = 1, q = 7$. The file blocks are:

$$\begin{cases} v_1 = 3 \in \mathbb{F}_7^1 \\ v_2 = 4 \in \mathbb{F}_7^1 \\ v_3 = 2 \in \mathbb{F}_7^1 \end{cases} \quad (5.56)$$

The augmented blocks are:

$$\begin{cases} w_1 = (v_1, 1, 0, 0) = (3, 1, 0, 0) \in \mathbb{F}_7^4 \\ w_2 = (v_2, 0, 1, 0) = (4, 0, 1, 0) \in \mathbb{F}_7^4 \\ w_3 = (v_3, 0, 0, 1) = (2, 0, 0, 1) \in \mathbb{F}_7^4 \end{cases} \quad (5.57)$$

5.8.1 Generating Keys

Key for client. $k_C \xleftarrow{\text{rand}} \mathbb{F}_7^4$. Suppose:

$$k_C = \begin{pmatrix} k_1 \\ k_2 \\ k_3 \\ k_4 \end{pmatrix} = \begin{pmatrix} 1 \\ 2 \\ 3 \\ 4 \end{pmatrix} \in \mathbb{F}_7^4 \quad (5.58)$$

Key for new server. $k_r = k_C + k_p \in \mathbb{F}_7^4$. k_p is generated such that $w_k k_p = 0$ for all $k \in \{1, \dots, m\}$ as follows:

- Construct a matrix M consisting of all augmented blocks:

$$M = \begin{pmatrix} w_1 \\ w_2 \\ w_3 \end{pmatrix} = \begin{pmatrix} 3 & 1 & 0 & 0 \\ 4 & 0 & 1 & 0 \\ 2 & 0 & 0 & 1 \end{pmatrix} \quad (5.59)$$

- Transform M by Gaussian row echelon in \mathbb{F}_7 to obtain M' as follows:

$$M' = \begin{pmatrix} \boxed{1} & 5 & 0 & 0 \\ 0 & \boxed{1} & 1 & 0 \\ 0 & 0 & \boxed{1} & 5 \end{pmatrix} \quad (5.60)$$

- Let $x = (x_1, x_2, x_3, x_4)^T$. Solve $M'x = 0$, we have:

$$\begin{cases} x_1 + 5x_2 = 0 \\ x_2 + x_3 = 0 \\ x_3 + 5x_4 = 0 \end{cases} \quad (5.61)$$

- From M' , determine free variables = $\{x_4\}$ and pivot variables = $\{x_1, x_2, x_3\}$. Equation 5.61 yields:

$$\begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{pmatrix} = x_4 \begin{pmatrix} 3 \\ 5 \\ 2 \\ 1 \end{pmatrix} \quad (5.62)$$

- Thus, the basis vector of M is:

$$B = \begin{pmatrix} b_1 \\ b_2 \\ b_3 \\ b_4 \end{pmatrix} = \begin{pmatrix} 3 \\ 5 \\ 2 \\ 1 \end{pmatrix} \quad (5.63)$$

- Generate randomly $r = 3$ in \mathbb{F}_7 .
- Compute k_p as:

$$k_p = rB = 3 \begin{pmatrix} 3 \\ 5 \\ 2 \\ 1 \end{pmatrix} \mod 7 = \begin{pmatrix} 2 \\ 1 \\ 6 \\ 3 \end{pmatrix} \quad (5.64)$$

It is clear that:

$$\begin{cases} w_1 k_p \mod 7 = 0 \\ w_2 k_p \mod 7 = 0 \\ w_3 k_p \mod 7 = 0 \end{cases} \quad (5.65)$$

- Finally, k_r is computed as:

$$k_r = k_C + k_p = \begin{pmatrix} 1 \\ 2 \\ 3 \\ 4 \end{pmatrix} + \begin{pmatrix} 2 \\ 1 \\ 6 \\ 3 \end{pmatrix} \mod 7 = \begin{pmatrix} 3 \\ 3 \\ 2 \\ 0 \end{pmatrix} \quad (5.66)$$

5.8.2 Dynamic Operations

Modification

Suppose $v_2 = 4$ is modified to $v'_2 = 5$. Matrix M is changed as:

$$M = \begin{pmatrix} 3 & 1 & 0 & 0 \\ 4 & 0 & 1 & 0 \\ 2 & 0 & 0 & 1 \end{pmatrix} \rightarrow M' = \begin{pmatrix} 3 & 1 & 0 & 0 \\ \boxed{5} & 0 & 1 & 0 \\ 2 & 0 & 0 & 1 \end{pmatrix} \quad (5.67)$$

1. \mathcal{C} recomputes k_r :

- Recompute basis vector:

$$B' = \begin{pmatrix} b_1 \\ b_2 \\ -b_1 v'_2 \mod 7 \\ b_4 \end{pmatrix} = \begin{pmatrix} 3 \\ 5 \\ 6 \\ 1 \end{pmatrix} \quad (5.68)$$

- Generate randomly $r' = 2 \in \mathbb{F}_7$.

- Recompute k_p :

$$k'_p = r'B' = 2 \begin{pmatrix} 3 \\ 5 \\ 6 \\ 1 \end{pmatrix} \mod 7 = \begin{pmatrix} 6 \\ 3 \\ 5 \\ 2 \end{pmatrix} \quad (5.69)$$

It is clear that:

$$\begin{cases} k_p w_1 \mod 7 = 0 \\ k_p w'_2 \mod 7 = 0 \\ k_p w_3 \mod 7 = 0 \end{cases} \quad (5.70)$$

- Recompute k_r :

$$k'_r = k_c + k'_p = \begin{pmatrix} 1 \\ 2 \\ 3 \\ 4 \end{pmatrix} + \begin{pmatrix} 6 \\ 3 \\ 5 \\ 2 \end{pmatrix} \mod 7 = \begin{pmatrix} 0 \\ 5 \\ 1 \\ 6 \end{pmatrix} \quad (5.71)$$

2. \mathcal{C} computes tag for w'_2 :

- \mathcal{C} computes:

$$t'_2 = w'_2 k_c = (5, 0, 1, 0) \begin{pmatrix} 1 \\ 2 \\ 3 \\ 4 \end{pmatrix} \mod 7 = 1 \quad (5.72)$$

- \mathcal{C} sends $\{w'_2, t'_2\}$ to \mathcal{S}_i

3. \mathcal{S}_i recomputes coded blocks and tags:

Let $\{c_{ij}[1], \dots, c_{ij}[4]\}$ denote the elements of c_{ij} :

$$c_{ij} = \begin{pmatrix} \sum_{k=1}^m \alpha_{ijk} w_k \\ \alpha_{ij1} \\ \alpha_{ij2} \\ \alpha_{ij3} \end{pmatrix}^T = \begin{pmatrix} c_{ij}[1] \\ c_{ij}[2] \\ c_{ij}[3] \\ c_{ij}[4] \end{pmatrix}^T \quad (5.73)$$

\mathcal{S}_i update coded blocks:

$$c_{ij} = \begin{pmatrix} c_{ij}[1] + \alpha_{ij2}(w'_2 - w_2) \\ c_{ij}[2] \\ c_{ij}[3] \\ c_{ij}[4] \end{pmatrix}^T \quad (5.74)$$

\mathcal{S}_i updates tags:

$$t'_{ij} = t_{ij} + \alpha_{ij2}(t'_2 - t_2) \quad (5.75)$$

where $\alpha_{ij2} = c_{ij}[3]$.

Insertion

Suppose $v_I = 1$ is inserted after v_2 . The matrix M is changed as follows:

$$M = \begin{pmatrix} 3 & 1 & 0 & 0 \\ 4 & 0 & 1 & 0 \\ 2 & 0 & 0 & 1 \end{pmatrix} \rightarrow M' = \begin{pmatrix} 3 & 1 & 0 & 0 & 0 \\ 4 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 \\ 2 & 0 & 0 & 0 & 1 \end{pmatrix} \quad (5.76)$$

1. \mathcal{C} recomputes $k_{\mathcal{C}}$:

$$k'_{\mathcal{C}} = \begin{pmatrix} k_1 \\ k_2 \\ k_3 \\ k_I \xleftarrow{rand} \mathbb{F}_q \\ k_4 \end{pmatrix} = \begin{pmatrix} 1 \\ 2 \\ 3 \\ 5 \\ 4 \end{pmatrix} \quad (5.77)$$

2. \mathcal{C} recomputes k_r :

- Recompute basis vector:

$$B' = \begin{pmatrix} b_1 \\ b_2 \\ b_3 \\ -v_I b_1 \bmod 7 \\ b_4 \end{pmatrix} = \begin{pmatrix} 3 \\ 5 \\ 2 \\ 4 \\ 1 \end{pmatrix} \quad (5.78)$$

- Generate randomly $r' = 4 \in \mathbb{F}_7$.
- Recompute k_p :

$$k'_p = r' B' = 4 \begin{pmatrix} 3 \\ 5 \\ 2 \\ 4 \\ 1 \end{pmatrix} \bmod 7 = \begin{pmatrix} 5 \\ 6 \\ 1 \\ 2 \\ 4 \end{pmatrix} \quad (5.79)$$

It is clear that:

$$\begin{cases} w_1 k'_p \bmod 7 = 0 \\ w_2 k'_p \bmod 7 = 0 \\ w_I k'_p \bmod 7 = 0 \\ w_3 k'_p \bmod 7 = 0 \end{cases} \quad (5.80)$$

- Recompute k_r :

$$k'_r = k'_{\mathcal{C}} + k'_p = \begin{pmatrix} 1 \\ 2 \\ 3 \\ 5 \\ 4 \end{pmatrix} + \begin{pmatrix} 5 \\ 6 \\ 1 \\ 2 \\ 4 \end{pmatrix} \bmod 7 = \begin{pmatrix} 6 \\ 1 \\ 4 \\ 0 \\ 1 \end{pmatrix} \quad (5.81)$$

3. \mathcal{C} computes tag for w'_2 :

- \mathcal{C} computes:

$$t_I = w_I k'_\mathcal{C} = (1, 0, 0, 1, 0) \begin{pmatrix} 1 \\ 2 \\ 3 \\ 5 \\ 4 \end{pmatrix} \mod 7 = 6 \quad (5.82)$$

- \mathcal{C} sends $\{w_I, t_I\}$ to \mathcal{S}_i .

4. \mathcal{S}_i recomputes coded blocks and tags:

Let $\{c_{ij}[1], \dots, c_{ij}[4]\}$ denote the elements of c_{ij} :

$$c_{ij} = \begin{pmatrix} \sum_{k=1}^m \alpha_{ijk} w_k \\ \alpha_{ij1} \\ \alpha_{ij2} \\ \alpha_{ij3} \end{pmatrix}^T = \begin{pmatrix} c_{ij}[1] \\ c_{ij}[2] \\ c_{ij}[3] \\ c_{ij}[4] \end{pmatrix}^T \quad (5.83)$$

\mathcal{S}_i updates coded blocks:

$$c'_{ij} = \begin{pmatrix} c_{ij}[1] + \alpha_{ijI} w_I \\ c_{ij}[2] \\ c_{ij}[3] \\ \alpha_{ijI} \xleftarrow{\text{rand}} \mathbb{F}_q \\ c_{ij}[4] \end{pmatrix}^T \quad (5.84)$$

\mathcal{S}_i updates tags:

$$t'_{ij} = t_{ij} + \alpha_{ijI} t_I \quad (5.85)$$

Deletion

Suppose $v_2 = 4$ is deleted. The matrix M is changed as follows:

$$M = \begin{pmatrix} 3 & 1 & 0 & 0 \\ 4 & 0 & 1 & 0 \\ 2 & 0 & 0 & 1 \end{pmatrix} \rightarrow M' = \begin{pmatrix} 3 & 1 & 0 \\ 2 & 0 & 1 \end{pmatrix} \quad (5.86)$$

1. \mathcal{C} recomputes $k_\mathcal{C}$:

$$k'_\mathcal{C} = \begin{pmatrix} k_1 \\ k_2 \\ k_4 \end{pmatrix} = \begin{pmatrix} 1 \\ 2 \\ 4 \end{pmatrix} \quad (5.87)$$

2. \mathcal{C} recomputes k_r :

- Recompute basis vector:

$$B' = \begin{pmatrix} b_1 \\ b_2 \\ b_4 \end{pmatrix} = \begin{pmatrix} 3 \\ 5 \\ 1 \end{pmatrix} \quad (5.88)$$

- Generate randomly $r' = 3 \in \mathbb{F}_7$.
- Recompute k_p :

$$k'_p = r'B' = 3 \begin{pmatrix} 3 \\ 5 \\ 1 \end{pmatrix} \mod 7 = \begin{pmatrix} 2 \\ 1 \\ 3 \end{pmatrix} \quad (5.89)$$

It is clear that:

$$\begin{cases} w_1 k'_p \mod 7 = 0 \\ w_3 k'_p \mod 7 = 0 \end{cases} \quad (5.90)$$

- Recompute k_r :

$$k'_r = k'_c + k'_p = \begin{pmatrix} 1 \\ 2 \\ 4 \end{pmatrix} + \begin{pmatrix} 2 \\ 1 \\ 3 \end{pmatrix} \mod 7 = \begin{pmatrix} 3 \\ 3 \\ 0 \end{pmatrix} \quad (5.91)$$

3. \mathcal{S}_i recomputes coded blocks and tags:

Let $\{c_{ij}[1], \dots, c_{ij}[4]\}$ denote the elements of c_{ij} :

$$c_{ij} = \begin{pmatrix} \sum_{k=1}^m \alpha_{ijk} w_k \\ \alpha_{ij1} \\ \alpha_{ij2} \\ \alpha_{ij3} \end{pmatrix}^T = \begin{pmatrix} c_{ij}[1] \\ c_{ij}[2] \\ c_{ij}[3] \\ c_{ij}[4] \end{pmatrix}^T \quad (5.92)$$

\mathcal{S}_i updates coded blocks:

$$c'_{ij} = \begin{pmatrix} c_{ij}[1] - \alpha_{ij2} w_2 \\ c_{ij}[2] \\ c_{ij}[3] \\ c_{ij}[4] \end{pmatrix}^T \quad (5.93)$$

\mathcal{S}_i updates tags:

$$t'_{ij} = t_{ij} - \alpha_{ij2} t_2 \quad (5.94)$$

where $\alpha_{ij2} = c_{ij}[3]$.

5.9 Summary

In this chapter, we have proposed a network coding-based POR scheme, name the DD-POR scheme, to support the direct repair and the dynamic operations in a symmetric key setting. The main idea is based on the inter MAC technique which can generate a key such such that the key is orthogonal to the augmented blocks. The security analysis shows that the scheme can prevent the pollution attack and curious attack. The efficiency analysis is given based on complexity theory to compare with the previous schemes.

Chapter 6

ND-POR: Network Coding and Dispersal Coding for POR

6.1 System Model

The ND-POR scheme has the following two entities:

- The first entity is the client who can be individuals or organizations. The client outsources his/her data to a cloud storage and relies on the cloud storage for data storage and maintenance.
- The second entity is the cloud servers which are managed by a cloud provider. The cloud servers store the data of the clients and have responsibility to prove to the client that the stored data are always available and intact.

6.2 Adversarial Model

The ND-POR scheme considers an adversary \mathcal{A} as follows. \mathcal{A} may control the servers by corrupting the servers and robbing all the privileges of the servers. If \mathcal{A} has not corrupted a server, \mathcal{A} cannot do anything because that all the data and the keys between the client and the servers are assumed to be transmitted via a secure channel. After \mathcal{A} corrupts a server, \mathcal{A} can modify/replace/forged data stored on that server and pretend to be a healthy server by providing a fake valid MAC tag to the client, can prevent the client from recovering the original file, and can perform the below four attacks (small corruption attack, large corruption attack, replay attack and pollution attack). A restriction of \mathcal{A} is that \mathcal{A} can control at most $(n - h)$ out of the n servers within any time step (called *epoch*). More concretely, after corrupting a server, \mathcal{A} can perform as follows:

1. Accessing to the encode and check phases to output a codeword c such that \mathcal{A} can pass the verification without being detected with an advantage defined as:

$$\text{Adv}_{\mathcal{A}}^{\text{ND-POR}} = \Pr[\kappa \leftarrow \text{KGenECC}_{\kappa}(1^{\lambda}); c \leftarrow \mathcal{A}^{\text{MTagECC}_{\kappa}(\cdot), \text{MVerECC}_{\kappa}(\cdot)}; \text{MVerECC}_{\kappa}(c) = (m, 1) \wedge m \text{ is not queried to } \text{MTagECC}_{\kappa}(\cdot)]$$

2. Preventing F to be recovered in the repair phase.

3. Performing the following four attacks:

- **Small corruption attack.** \mathcal{A} corrupts at most a t -fraction of the file F with a small data unit, where $t = \frac{n-l+1}{2}$, in order to hide the data loss incidents. This applies to the servers that want to preserve their reputation. To prevent the small corruption attack, the ECC is used to detect and correct errors [10, 26].
- **Large corruption attack.** \mathcal{A} corrupts more than a t -fraction of the file F with a large data unit, where t is the same parameter as in the small corruption attack, to discard a significant fraction of the data. This applies to the servers who want to sell the storage resource to multiple clients. To prevent the large data corruption, the spot check method is proposed [9, 102] in which the client randomly samples small portions of the data. Then, the server returns a computation over these portions of the data to the client. The results are checked by MACs. The spot check can only prevent the large corruption attack but cannot prevent the small corruption attack [8, 102].
- **Replay attack.** \mathcal{A} tries to prevent the client from repairing the corruption by re-using the old coded blocks instead of the current coded blocks and providing these old coded blocks to the client in the repair phase. For example:
 - The client encodes the augmented blocks $\{b_1, b_2, b_3\}$ into six coded blocks: $c_{11} = b_1$ and $c_{12} = b_2 + b_3$ (stored on the server S_1), $c_{21} = b_3$ and $c_{22} = b_1 + b_2$ (stored on the server S_2), $c_{31} = b_1 + b_3$ and $c_{32} = b_2 + b_3$ (stored on the server S_3).
 - In epoch 1, suppose that S_3 is corrupted.
 - In epoch 2, the client repairs S_3 by two new coded blocks: $c'_{31} = b_1 + b_2 + 2b_3$ and $c'_{32} = 2b_1 + b_2$.
 - In the end of epoch 2, suppose that S_1 is corrupted.
 - In epoch 3, S_1 is repaired by two new coded blocks: $c'_{11} = 3b_1 + 3b_2$ and $c'_{12} = 3b_2 + 3b_3$. At this time, \mathcal{A} re-uses the old coded blocks c_{31} and c_{32} of S_3 instead of c'_{31} and c'_{32} .
 - Thus, if S_2 is corrupted in epoch 4, the linear combination between the coded blocks of S_1 and S_2 is unable to repair S_2 .
- **Pollution attack.** \mathcal{A} uses a valid data to avoid detection in the check phase, but provides an invalid data in the repair phase. For example:
 - Encode: the client encodes the augmented blocks $\{b_1, b_2, b_3\}$ into six coded blocks: $c_{11} = b_1$ and $c_{12} = b_2 + b_3$ (stored on the server S_1), $c_{21} = b_3$ and $c_{22} = b_1 + b_2$ (stored on the server S_2), $c_{31} = b_1 + b_3$ and $c_{32} = b_2 + b_3$ (stored on the server S_3).
 - Check: suppose that the corrupted server S_3 is detected.
 - Repair: S_3 is repaired by two new coded blocks: $c'_{31} = b_1 + b_2 + 2b_3$ and $c'_{32} = 2b_1 + b_2$. At this time, \mathcal{A} corrupts S_1 without detection because this

time is the repair phase, not the check phase. To repair S_3 , suppose that the client requests coded blocks from S_1 and S_2 . S_1 then provides invalid coded blocks to the client.

One of the contributions is to prevent the small corruption attack. The other three attacks are still prevented in this ND-POR scheme by using the same solution as the RDC-NC scheme [61]. These are discussed in the security analysis of the ND-POR scheme.

6.3 Proposed ND-POR Scheme

Throughout this ND-POR scheme, the notations described in Table 6.1 are used.

Table 6.1: List of notations in the ND-POR scheme

Notation	Description
\mathcal{C}	client
F	original file
m	number of file blocks
n	number of servers
l	number of NC-servers
$n - l$	number of DC-servers
β	number of blocks stored on a server.
k	file block index ($k \in \{1, \dots, m\}$)
i	server index ($i \in \{1, \dots, n\}$)
j	coded block index in a server ($j \in \{1, \dots, \beta\}$)
v_k	file block ($k \in \{1, \dots, m\}$)
b_k	augmented block of v_k ($k \in \{1, \dots, m\}$)
S_i	server ($i \in \{1, \dots, n\}$)
c_{ij}	coded block ($i \in \{1, \dots, l\}$ and $j \in \{1, \dots, \beta\}$)
d_{ij}	dispersal coding parity block ($i \in \{l + 1, \dots, n\}$, $j \in \{1, \dots, \beta\}$)
\mathbb{F}_p	finite field of a prime order p
z	length of v_k over \mathbb{F}_p
\mathbb{F}_p^z	a vector of length z over \mathbb{F}_p
w	$w = z + m$ (length of a coded block over \mathbb{F}_p)
\mathbb{F}_p^w	a vector of length w over \mathbb{F}_p
f	Pseudo-Random Function (PRF) $f : \{0, 1\}^* \times \{0, 1\}^\kappa \rightarrow \mathbb{F}_p$ where κ is the key length of f and κ should be large enough (e.g., 160)
$ $	concatenate operator
S_y	corrupted server
S'_y	new server which is used to replace S_y
h	number of healthy servers used for data repair
v	number of rows used for spot checks

\mathcal{A}	adversary
s	number of segments in a coded block of the RDC-NC scheme
λ	security parameter
t	boundary of small corruption attack or ECC threshold ($t = (n - l + 1)/2$)

In the ND-POR scheme, n servers are employed:

- The first l servers $\{S_1, \dots, S_l\}$, called **NC-servers**, store the coded blocks c_{ij} where $i \in \{1, \dots, l\}, j \in \{1, \dots, \beta\}$.
- The last $(n-l)$ servers $\{S_{l+1}, \dots, S_n\}$, called **DC-servers**, store the dispersal coding parity blocks d_{ij} where $i \in \{l+1, \dots, n\}, j \in \{1, \dots, \beta\}$.

The structure of the ND-POR scheme is depicted in Figure 6.1.

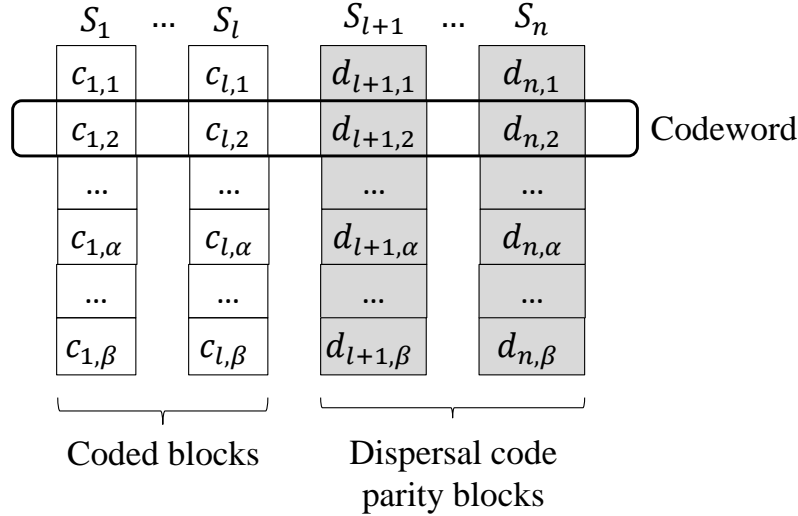


Figure 6.1: The structure of the ND-POR scheme

The ND-POR scheme is now described via each phase of the POR as follows.

6.3.1 Keygen

\mathcal{C} generates the secret key: $\mathcal{K} = \{\mathcal{K}_{\text{rtag}}, \mathcal{K}'_{\text{rtag}}, \{\mathcal{K}_i, \mathcal{K}'_i\}_{i \in \{l+1, \dots, n\}}, \mathcal{K}_{\text{enc}}\}$ which are randomly chosen in $\{0, 1\}^\kappa$.

6.3.2 Encode

\mathcal{C} divides the original file into m file blocks: $F = v_1 || \cdots || v_m$. $v_k \in \mathbb{F}_p^z$ where $k \in \{1, \dots, m\}$. \mathcal{C} creates m augmented blocks $\{b_1, \dots, b_m\}$ in which $b_k \in \mathbb{F}_p^w$ ($k \in \{1, \dots, m\}$) has the following form:

$$b_k = (v_k, \underbrace{0, \dots, 0, 1, 0, \dots, 0}_k) \in \mathbb{F}_p^{z+m} \quad (6.1)$$

where $w = z + m$. Given a set of m augmented blocks, \mathcal{C} computes $l\beta$ coded blocks c_{ij} using the linear combinations and stores them on the NC servers $\{S_1, \dots, S_l\}$. \mathcal{C} then encodes c_{ij} using the dispersal coding into a dispersal coding parity block d_{ij} for each row and stores them on the DC-servers $\{S_{l+1}, \dots, S_n\}$. Namely, the encode phase is described as follows.

1. \mathcal{C} computes coded blocks from m augmented blocks:

For $\forall i \in \{1, \dots, l\}, \forall j \in \{1, \dots, \beta\}$:

- Generate m coefficients $\alpha_{ijk} \xleftarrow{\text{rand}} \mathbb{F}_p$ where $k \in \{1, \dots, m\}$.
- Compute coded blocks:

$$c_{ij} = \sum_{k=1}^m \alpha_{ijk} b_k \quad (6.2)$$

Therefore, a matrix $\{c_{ij}\}$ where $i \in \{1, \dots, l\}, j \in \{1, \dots, \beta\}$ is constructed.

2. \mathcal{C} computes dispersal coding parity blocks in each row:

For $\forall i \in \{l+1, \dots, n\}, \forall j \in \{1, \dots, \beta\}$, \mathcal{C} computes dispersal coding parity blocks:

$$d_{ij} = \text{MTagECC}_{\mathcal{K}_i, \mathcal{K}'_i}(c_{i1}, \dots, c_{i\beta}) \quad (6.3)$$

3. \mathcal{C} computes metadata for coded blocks:

For $\forall i \in \{1, \dots, l\}$:

- Generate w values $\{\xi_1, \dots, \xi_w\}$:

$$\xi_u = f_{\mathcal{K}_{\text{tag}}}(i || u), \forall u \in \{1, \dots, w\} \quad (6.4)$$

- For $\forall j \in \{1, \dots, \beta\}$, $c_{ij} \in \mathbb{F}_p^w$ is viewed as a column vector of w symbols: $c_{ij} = (c_{ij1}, \dots, c_{ijw})$ with $c_{iju} \in \mathbb{F}_p$ where $u \in \{1, \dots, w\}$. \mathcal{C} computes a repair tag for c_{ij} :

$$T_{ij} = f_{\mathcal{K}'_{\text{tag}}}(i || j || \alpha_{ij1} || \cdots || \alpha_{ijm}) + \sum_{u=1}^w \xi_u c_{iju} \pmod{p} \quad (6.5)$$

- \mathcal{C} encrypts the coefficients:

$$\epsilon_{ijk} = \text{Enc}_{\mathcal{K}_{\text{enc}}}(\alpha_{ijk}), \forall k \in \{1, \dots, m\}. \quad (6.6)$$

This encryption is used to prevent the replay attack.

4. \mathcal{C} distributes data to the servers:

- \mathcal{C} sends to S_i where $i \in \{1, \dots, l\}$ the following information:
 - The coded blocks c_{ij} where $i \in \{1, \dots, l\}, j \in \{1, \dots, \beta\}$.
 - The encrypted coefficients ϵ_{ijk} where $i \in \{1, \dots, l\}, j \in \{1, \dots, \beta\}, k \in \{1, \dots, m\}$.
 - The repair tags T_{ij} where $i \in \{1, \dots, l\}, j \in \{1, \dots, \beta\}$.
- \mathcal{C} sends to S_i where $i \in \{l+1, \dots, n\}$ the following information:
 - The dispersal coding parity blocks d_{ij} where $i \in \{l+1, \dots, n\}, j \in \{1, \dots, \beta\}$.

6.3.3 Check

\mathcal{C} chooses a number of row indices to challenge the servers using the spot check method. The servers respond \mathcal{C} . \mathcal{C} checks the responses using the MVerECC algorithm. All the servers operate over the same subset of rows. Because the responses of all the servers lie on a codeword, all the servers can be checked for each challenge.

1. \mathcal{C} challenges the servers:

- \mathcal{C} firstly chooses an integer $v \xleftarrow{\text{rand}} [1, \beta]$.
- \mathcal{C} then sends to each server a set of row indices $D = \{j_1, \dots, j_v\}$ where $j_1, \dots, j_v \xleftarrow{\text{rand}} [1, \beta]$ and a key $k \in \mathcal{I}$ where \mathcal{I} is a field with operation $(+, \times)$.

2. The servers respond \mathcal{C} :

- S_i computes:

$$R_i = \text{RS-UHF}_k(c_{ij_1}, \dots, c_{ij_v}) \quad (6.7)$$

3. \mathcal{C} verifies the servers: Because all servers operate over the same subset of rows D , the combined response $R = (R_1, \dots, R_n)$ is a codeword of the dispersal coding.

- \mathcal{C} firstly checks R by calling $\text{MVerECC}(R_1, \dots, R_n)$ of the dispersal coding to verify. It returns **false** if the responses are invalid, and return **true** otherwise.
- After checking R , \mathcal{C} checks the validity of each individual response R_i to detect which server is corrupted. For the l NC-servers $\{S_1, \dots, S_l\}$, R_i is a valid response if it matches the i -th symbol in \vec{m} . For the $(n-l)$ DC-servers $\{S_{l+1}, \dots, S_n\}$, R_i is a valid response if it is a valid MAC on \vec{m} .

6.3.4 Repair

If a failure is detected in the check phase, \mathcal{C} executes the repair phase with the following two sub-phases:

Sub-phase 1. The corruptions are firstly repaired by the RS decoder with the boundary number of corruptions $t = \frac{n-l+1}{2}$. If the number of corruptions is more than t , \mathcal{C} uses the sub-phase 2.

Sub-phase 2. The corruptions are repaired by the network coding. \mathcal{C} firstly requires the healthy servers to compute the aggregated coded blocks. Then, \mathcal{C} combines these coded blocks to generate β coded blocks for the new server. Suppose that S_y is the corrupted server and S'_y is the new server which is used to replace S_y .

1. \mathcal{C} requests h healthy servers $\{S_{i_1}, \dots, S_{i_h}\}$ to compute the aggregated coded blocks and the proofs of correct encoding:

For $\forall i \in \{i_1, \dots, i_h\}$:

- \mathcal{C} generates the coefficients $\{x_{i1}, \dots, x_{i\beta}\}$ where $x_{ij} \xleftarrow{rand} \mathbb{F}_p$ with $j \in \{1, \dots, \beta\}$.
- \mathcal{C} requests S_i to compute an aggregated coded block and a proof of correct encoding.
- S_i computes:

$$\overline{a_i} = \sum_{j=1}^{\beta} x_{ij} c_{ij} \in \mathbb{F}_p^w \quad (6.8)$$

then computes a proof of correct encoding:

$$\theta = \sum_{j=1}^{\beta} x_{ij} T_{ij} \pmod{p} \quad (6.9)$$

and sends $\overline{a_i}, \theta, \{\epsilon_{ij1}, \dots, \epsilon_{ijm}\}$ where $j \in \{1, \dots, \beta\}$ to \mathcal{C} .

- \mathcal{C} decrypts the encrypted coefficients from S_i to get the raw coefficients: $\{\alpha_{ij1}, \dots, \alpha_{ijm}\}$ where $j \in \{1, \dots, \beta\}$.
- \mathcal{C} regenerates w values $\{\xi_1, \dots, \xi_w\}$:

$$\xi_u = f_{\mathcal{K}_{\text{rtag}}}(i||u), \forall u \in \{1, \dots, w\} \quad (6.10)$$

- \mathcal{C} checks if:

$$\theta \neq \sum_{j=1}^{\beta} x_{ij} f_{\mathcal{K}'_{\text{rtag}}}(i||j||\alpha_{ij1}||\dots||\alpha_{ijm}) + \sum_{u=1}^w \xi_u a_{iu} \quad (6.11)$$

where $\{a_{i1}, \dots, a_{iw}\}$ are the symbols of the block $\overline{a_i}$. This verification is to ensure that S_i does not have the pollution attack.

2. \mathcal{C} repairs S_y :

- \mathcal{C} generates w values $\{\xi_1, \dots, \xi_w\}$:

$$\xi_u = f_{\mathcal{K}_{\text{rtag}}}(y||u), \forall u \in \{1, \dots, w\} \quad (6.12)$$

- For $\forall j \in \{1, \dots, \beta\}, \forall \gamma \in \{1, \dots, h\}$, \mathcal{C} generates the coefficients $\alpha_{yj\gamma} \xleftarrow{\text{rand}} \mathbb{F}_p$, and computes the coded block:

$$c_{yj} = \sum_{\gamma=1}^h z_{yj\gamma} \overline{a_\gamma} \in \mathbb{F}_p^w \quad (6.13)$$

By viewing c_{yj} as a column vector of w symbols: $c_{yj} = \{c_{yj1}, \dots, c_{yjh}\}$, \mathcal{C} computes a repair tag for the block c_{yj} :

$$T_{yj} = f_{\mathcal{K}'_{\text{rtag}}}(i||j||\alpha_{yj1}||\dots||\alpha_{yjh}) + \sum_{u=1}^w \xi_u c_{yju} \pmod{p} \quad (6.14)$$

and encrypts the coefficient:

$$\forall \gamma \in \{1, \dots, h\}, \epsilon_{ij\gamma} = \text{Enc}_{\mathcal{K}_{\text{enc}}}(\alpha_{ij\gamma}) \quad (6.15)$$

3. \mathcal{C} sends to the new server S'_y :

- c_{yj} where $j \in \{1, \dots, \beta\}$.
- $\epsilon_{yj\gamma}$ where $j \in \{1, \dots, \beta\}, \gamma \in \{1, \dots, h\}$.
- T_{yj} where $j \in \{1, \dots, \beta\}$.

6.4 Security Analysis

This section describes the advantage of the defined adversary and explains how the small corruption attack, large corruption attack, replay attack and pollution attack are prevented.

6.4.1 Adversarial Check and Repair

A MAC consists of three algorithms: $\{\text{MGen}, \text{MTag}, \text{MVer}\}$. Let q_1 denote the number of queries to MTag , and q_2 denote the number of queries to MVer . Let t denote the running time. The boundary of the advantage of \mathcal{A} on UMAC [99] is given in the following fact:

Fact 1. Let $\text{Adv}_{\text{UMAC}}(q_1, q_2, t)$ denote the advantage of the adversary \mathcal{A} on UMAC making q_1 queries to MTag , q_2 queries to MVer , and running in the time t . Let $\text{Adv}_{\text{prf}}(q_1, q_2, t)$ denote the advantage of the adversary \mathcal{A} making $(q_1 + q_2)$ queries to the oracle PRF and running in the time t . Suppose that the UHF is an ϵ^{UHF} -AXU family of hash function. Then, the following inequality is obtained:

$$\text{Adv}_{\text{UMAC}}(q_1, q_2, t) \leq \text{Adv}_{\text{prf}}(q_1 + q_2, t) + \epsilon^{\text{UHF}} q_2 \quad (6.16)$$

Furthermore, the boundary of the advantage of \mathcal{A} on the dispersal coding codeword is given as follows:

Theorem 16. *Let $\text{Adv}_{\text{codeword}}(q_1, q_2, t)$ denote the advantage of \mathcal{A} on the dispersal coding making q_1 queries to MTagECC , q_2 queries to MVerECC , and running in the time t . If RS-UHF is constructed from an (n, l) -RS code, then the following inequality is obtained:*

$$\text{Adv}_{\text{codeword}}(q_1, q_2, t) \leq 2[\text{Adv}_{\text{UMAC}}(q_1, q_2, t)] \quad (6.17)$$

Proof. Suppose that \mathcal{A} is successful when \mathcal{A} makes q_1 queries to the tagging oracle MTagECC , q_2 queries to the verification oracle MVerECC and runs in time t . \mathcal{A} outputs a codeword (c_1, \dots, c_n) which can be decoded to the message $\vec{m} = (m_1, \dots, m_l)$ such that at least one of the last s symbols in the codeword is a valid MAC on \vec{m} computed with UMAC. Another adversary \mathcal{A}' is considered for the UMAC construction. \mathcal{A}' is given an access to a tagging oracle $\text{UTag}_{\kappa, \kappa'}(\cdot)$ and a verification oracle $\text{UVer}_{\kappa, \kappa'}(\cdot, \cdot)$ and needs to output a new message and a tag pair. \mathcal{A}' chooses a position $j \in \{n - s + 1, \dots, n\}$ randomly, and generates keys $\{\kappa_i\}_{i=1}^n$ and $\{\kappa'_i\}_{i=n-s+1}^n$ for $i \neq j$. \mathcal{A}' runs \mathcal{A} . When \mathcal{A} makes a query to tag $\vec{m} = (m_1, \dots, m_l)$, \mathcal{A}' computes $c_i \leftarrow \text{RS-UHF}_{\kappa_i}(\vec{m})$ for $i \in \{1, \dots, n - s\}$, and $c_i = \text{UTag}_{\kappa_i, \kappa'_i}(\vec{m})$ for $i \in \{n - s + 1, \dots, n\}, i \neq j$. \mathcal{A}' calls the UTag oracle to compute $c_j = \text{UTag}_{\kappa, \kappa'}(\vec{m})$. \mathcal{A}' then responds to \mathcal{A} with $\vec{c} \leftarrow (c_1, \dots, c_n)$. When \mathcal{A} makes a query $\vec{c} = (c_1, \dots, c_n)$ to the verification oracle, \mathcal{A}' tries to decode $(c_1, \dots, c_{j-1}, c_{j+1}, \dots, c_n)$ into message \vec{m} . If the decoding fails (the number of errors in the codeword is more than $t = \frac{n-l+1}{2}$), then \mathcal{A}' responds to \mathcal{A} with $(\perp, 0)$. Otherwise, let \vec{m} be the decoded message. \mathcal{A}' makes a query to the verification oracle $\alpha \leftarrow \text{UVer}_{\kappa, \kappa'}(\vec{m}, c_j)$ and returns (\vec{m}, α) to \mathcal{A} . Assume that \mathcal{A} outputs $\vec{c} = (c_1, \dots, c_n)$ under the codeword that can be decoded to \vec{m} , such that \vec{m} was not an input to the tagging oracle and at least one of the last s symbols in \vec{c} is a valid MAC for \vec{m} . Then \mathcal{A}' outputs (\vec{m}, c_j) . Because $t = \frac{n-l+1}{2}$, the number of remaining correct blocks is at least $n - \frac{n-l+1}{2} = \frac{n+l-1}{2}$. The number of correct parity blocks is thus at least $\frac{n+l-1}{2} - l = \frac{n-l-1}{2}$. Furthermore, the number of the original parity blocks before errors is $(n - l)$. Therefore, the number of correct parity blocks is at least $\frac{(n-l-1)/2}{n-l} \approx \frac{1}{2}$ of the number of the original parity blocks. In other words, the codeword \vec{c} can be decoded if at least a majority of its parity blocks are correct. Then, with probability at least $\frac{1}{2}$, c_j is a correct MAC on \vec{m} . It follows that \mathcal{A}' succeeds in outputting a correct message and MAC pair (\vec{m}, c_j) with probability at least half the success probability of \mathcal{A} . \square

From Equation 6.16 and Equation 6.17, the following inequality is obtained:

$$\text{Adv}_{\text{codeword}}(q_1, q_2, t) \leq \text{Adv}_{\text{prf}}(q_1 + q_2, t) + \epsilon^{\text{UHF}}_{q_2} \quad (6.18)$$

Assume that the PRF is secure and the MAC is unforgeable, then:

$$\text{Adv}_{\text{prf}}(q_1 + q_2, t) + \epsilon^{\text{UHF}}_{q_2} \leq \varepsilon(\text{negligible}). \quad (6.19)$$

Therefore, $\text{Adv}_{\text{codeword}}(q_1, q_2, t) \leq \varepsilon$. In other words, the probability for \mathcal{A} to output codeword c such that \mathcal{A} can pass the verification is negligible:

$\Pr[\kappa \leftarrow \text{KGenECC}_\kappa(1^\lambda); c \leftarrow \mathcal{A}^{\text{MTagECC}_\kappa(\cdot), \text{MVerECC}_\kappa(\cdot)} : \text{MVerECC}_\kappa(c) = (m, 1) \wedge m \text{ is not queried to } \text{MTagECC}_\kappa(\cdot)] \leq \varepsilon.$

Now the probability of \mathcal{A} to prevent data recovery is given as the following theorem.

Theorem 17. *F can be recovered as long as in any epoch, at least h out of n servers are healthy and the matrix which consists of all the coefficients of the coded blocks has full rank, i.e., rank equals to m .*

Proof. m augmented blocks are $\{b_1, \dots, b_m\}$ which are created from m file blocks $\{v_1, \dots, v_m\}$. The number of coded blocks is nx (n servers, x coded blocks per server). To compute a coded block c_{ij} for the server S_i , \mathcal{C} chooses m coefficients $\{\alpha_{ij1}, \dots, \alpha_{ijm}\}$, and uses the linearly independent combination: $c_{ij} = \sum_{k=1}^m \alpha_{ijk} b_k$. $\{b_1, \dots, b_m\}$ are viewed as the unknowns that need to be solved. After solving $\{b_1, \dots, b_m\}$, the file blocks v_1, \dots, v_m can be obtained by picking the first coordinate of each b_k where $k \in \{1, \dots, m\}$. F is finally recovered as $F = v_1 || \dots || v_m$. To solve m unknowns $\{b_1, \dots, b_m\}$, at least m coded blocks are required which make the matrix have full rank because the number of unknowns in an equation system has to be less than the number of equations. Let $\{c_{r_1}, \dots, c_{r_m}\}$ denote such m coded blocks which are required for file recovery. Let $\{\alpha_{r_k1}, \dots, \alpha_{r_km}\}$ denote m coefficients which are used to construct c_{r_k} .

$$\begin{cases} c_{r_1} = \sum_{k=1}^m \alpha_{r_1k} b_k \\ c_{r_2} = \sum_{k=1}^m \alpha_{r_2k} b_k \\ \dots \\ c_{r_m} = \sum_{k=1}^m \alpha_{r_mk} b_k \end{cases} \quad (6.20)$$

Let h be the number of healthy servers that collectively store m coded blocks. In any epoch, $h = \frac{m}{x}$. In the RDC-NC scheme, there are n servers and α coded blocks per server. Thus, the number of healthy servers in an epoch in the RDC-NC scheme is at least $h = \frac{m}{\alpha}$. In the ND-POR scheme, there are also n servers but such n servers are divided into two types: l NC-servers and $(n - l)$ DC-servers. Because $l < n$, each NC-server has $\beta = \frac{n\alpha}{l}$ coded blocks. Therefore, the number of healthy servers in each epoch is at least $h = \frac{m}{\beta} = \frac{ml}{\alpha n}$. If the theorem is satisfied, the probability for \mathcal{A} to prevent recovering F is negligible:

$$\Pr[F = \{v_1, \dots, v_m\} \text{ cannot be recovered}] \leq \varepsilon \quad (6.21)$$

□

6.4.2 Small Corruption Attack

Theorem 18. *The RS code in the dispersal coding is sufficient to prevent the small corruption attack.*

Proof. Let t_{error} denote the number of corruptions caused by \mathcal{A} in an epoch. Firstly, because the RS is constructed with the parameter (n, l) , the message is interpreted as the description of a polynomial p of the degree less than l which is evaluated at n distinct

points $\{a_1, \dots, a_n\}$. The sequence of the values is the corresponding codeword \mathcal{C} : $\mathcal{C} = \{p(a_1), \dots, p(a_n)\}$ (Section 3.4). Because any two different polynomials of the degree less than l agree in at most $(l - 1)$ points, any two codewords of the RS code disagree in at least $n - (l - 1) = n - l + 1$ positions. Moreover, there are two polynomials that do agree in $(l - 1)$ points but are not equal. Hence, the distance of the RS code is:

$$d = n - l + 1 \quad (6.22)$$

Secondly, because any two strings in \mathcal{C} differ in at least d places, we have:

$$2t_{\text{error}} \leq d \quad (6.23)$$

From Equation 6.22 and Equation 6.23, we have:

$$t_{\text{error}} \leq \frac{n - l + 1}{2} \quad (6.24)$$

The inequality reflects the fact that, given any string s , there is at most one string $c \in \mathcal{C}$ which is within the distance d of t_{error} from s . This means that the advantage of \mathcal{A} is always bounded by the error resilience of the RS code. \square

6.4.3 Large Corruption Attack, Replay Attack and Pollution Attack

The attacks are addressed in the RDC-NC scheme. This section briefly describes the key ideas as follows:

Large corruption attack. In the check phase, using the spot check method, \mathcal{C} periodically and randomly samples a set of indices of the coded blocks stored on each server. \mathcal{C} then checks whether these sampled blocks match with the embedded MAC. If the adversary corrupts a large fraction of the data stored on the server, \mathcal{C} easily detects the corruptions with an optimal computation and I/O at the server and communication between the server and the client.

Replay attack. To avoid the adversary \mathcal{A} replaying a coded block, the common solution is to use a counter which is incremented each time the coded block on a server is recreated due to server failure. However, in this solution, the client must store locally the latest value of the counters. Therefore, the RDC-NC scheme uses a different solution to mitigate the replay attack and to reduce the storage cost for the client. That is, the coefficients are encrypted and stored together with the coded blocks to prevent \mathcal{A} from knowing how the original blocks were combined to obtain the coded block. The ability of \mathcal{A} is negligible because \mathcal{A} does not know which old coded blocks to replay.

Pollution attack. For each coded block c_{ij} of the server S_i , a repair tag which is constructed from a MAC is embedded into the coded block. In the repair phase, \mathcal{C} requires a number of servers which are used for data repair to provide their aggregated coded blocks. Before computing the new coded blocks for the new server, \mathcal{C} uses the tag to check whether these servers combine the coded blocks correctly. Therefore, the servers cannot inject polluted blocks to \mathcal{C} .

6.5 Efficiency Analysis

Table 6.2 shows that the encode cost in the ND-POR scheme is more than that in the RDC-NC scheme. However, the encode phase is performed only one time in the beginning, but the check and repair phases are performed very often during the system lifetime. Therefore, the check and repair costs are more important than the encode cost. Table 6.2 shows that the check and repair costs in the ND-POR scheme are less than these in the RDC-NC scheme.

Before analysing the costs in the RDC-NC and ND-POR schemes, recall that each coded block $c_{ij} \in \mathbb{F}_p^w$ where $w = m + z$. The coded block size is $w \log_2 p$. The unit of the below computational complexities is the number of operations over \mathbb{F}_p^w .

6.5.1 Encode Phase

The RDC-NC scheme computes $n\alpha$ coded blocks. Its encode computation cost is thus $O(n\alpha)$. The ND-POR scheme computes $l\beta$ coded blocks, then computes $(n-l)\beta$ dispersal coding parity blocks, where $\beta = \frac{n\alpha}{l}$. Its encode computation cost is thus $O(\frac{n^2\alpha}{l})$. It is clear that the cost in the ND-POR scheme is more than $\frac{n}{l}$ times that in the RDC-NC scheme. In an ECC, because the redundant blocks are chosen such that $n-l < l$, $\frac{n}{l} < 2$. Because $n > l$, $\frac{n}{l} > 1$. Therefore, $1 < \frac{n}{l} < 2$. This means that although the cost in the ND-POR scheme is more than $\frac{n}{l}$ times that of the RDC-NC scheme, it is less than double times.

The number of MACs. In the RDC-NC scheme, the number of MACs is $n\alpha s$ where n is the number of servers, α is the number of coded blocks stored on a server, s is the number of segments in a coded block. This is because the MACs are embedded in the segments of the coded blocks. In the ND-POR scheme, the number of MACs is $l\alpha$ where l is a number of servers out of n servers ($l < n$). This is because the MACs are only required for the network coding coded blocks which are located in only l servers.

6.5.2 Check Phase

The RDC-NC scheme challenges a subset of segment indices in each coded block of a server. \mathcal{C} thus needs $n\alpha$ challenges to check all blocks stored on n servers where α is the number of blocks per server. In the ND-POR scheme, \mathcal{C} challenges a subset of row indices (the codewords of the dispersal coding). Each codeword lies on all n servers. There are

Table 6.2: The comparison between the RDC-NC and ND-POR schemes

	RDC-NC	ND-POR
Security	Small corruption attack	No
	Large corruption attack	Yes
	Replay attack	Yes
	Pollution attack	Yes
Efficiency	Encode computation	$O(n\alpha \times \frac{n}{l})$
	Check computation	$\frac{n\alpha}{l}$
	Repair computation	$O(n \log_2^2 n \log_2 \log_2 n)$ (RS decode) $O(\frac{3lm F }{n\alpha+lm})$ (Network coding)
	The number of MACs	$l\alpha$
	Required healthy servers	$\lceil \frac{lm}{n\alpha} \rceil (l < n)$
	Storage server cost	$O(n\alpha \log_2 p(\frac{nw}{l} + 1))$

β codewords. \mathcal{C} thus needs $\beta = \frac{n\alpha}{l}$ challenges to check all blocks stored on n servers. The cost in the RDC-NC scheme is more than l times that in the ND-POR scheme. This is an advantage of the ND-POR scheme when multiple servers are checked per challenge instead of one sever as the RDC-NC scheme.

6.5.3 Repair Phase

In the RDC-NC scheme, the cost for repairing a corrupted server is $O(h \frac{2|F|}{h+1} + \frac{|F|}{1+\frac{1}{h}})$ in which the cost of h healthy servers is $h \frac{2|F|}{h+1}$ and the cost of client-side is $\frac{|F|}{1+\frac{1}{h}}$. Because $h = \frac{m}{\alpha}$, the cost is $O(|F| \frac{3m}{\alpha+m})$. In the ND-POR scheme, in the sub-phase 1, the corruptions are repaired by the RS decoder which has $O(n \log_2^2 n \log_2 \log_2 n)$ [107]. Because n is far less than the dominant parameter $|F|$, the cost of the RS decoder is less than the RDC-NC scheme. Let $n = 12$ as in the experimental evaluation of the RDC-NC scheme, the RS can decode in only 284 field operations. In the sub-phase 2, the corruptions are repaired by the network coding like the RDC-NC scheme. The cost is thus $O(h' \frac{2|F|}{h'+1} + \frac{|F|}{1+\frac{1}{h'}})$. Because $h' = \frac{lm}{n\alpha}$, the cost in the ND-POR is thus $O(\frac{3lm|F|}{n\alpha+lm})$. Because $n > l$, $\frac{3m|F|}{m+\alpha} > \frac{3lm|F|}{n\alpha+lm}$. In both cases, the costs in the ND-POR scheme are still better than the RDC-NC scheme.

Parameter Choice. The parameter choice is now discussed for maximizing resilience of both cases simultaneously. Because an (n, l) -ECC can recover up to $t = \frac{n-l+1}{2}$ errors in each row, \mathcal{A} can win the ECC if the number of corruptions is more than t . Furthermore, because the number of healthy servers is at least $\frac{ml}{n\alpha}$ (Theorem 2), \mathcal{A} can win if \mathcal{A} corrupts more than $\frac{ml}{n\alpha}$. Let $f_1(l) = \frac{n-l+1}{2}$, $f_2(l) = \frac{m}{\alpha} \times \frac{l}{n}$. l should be chosen such that the advantage of \mathcal{A} is reduced. In other words, $f_1(l)$ and $f_2(l)$ are increased. If $f_1(l)$ and $f_2(l)$ are considered separately, $f_1(l) = \frac{n-l+1}{2}$ increases if l increases, $f_2(l) = \frac{ml}{n\alpha}$ increases if l decreases. It is not synchronous. Hence, l should be balanced between f_1 and f_2 . Let $f_1(l) = f_2(l)$, we determine $l = \lceil \frac{n\alpha(n+1)}{2m+n\alpha} \rceil$.

Healthy Servers For Data Repair. As mentioned in Theorem 17, the RDC-NC scheme needs at least $\lceil \frac{m}{\alpha} \rceil$ healthy servers for data repair while the ND-POR scheme only needs at least $\lceil \frac{lm}{n\alpha} \rceil$ healthy servers for data repair ($l < n$). It is clear that the number of healthy servers in the RDC-NC scheme is more than $\frac{n}{l}$ times that in the ND-POR scheme.

6.5.4 Storage Cost

In the RDC-NC scheme, the size of $n\alpha$ coded blocks in \mathbb{F}_p^w is $n\alpha w \log_2 p$. The size of $n\alpha s$ challenge tags in \mathbb{F}_p is $n\alpha s \log_2 p$ where s denotes the number of segments in a coded block of the RDC-NC scheme. The size of $n\alpha$ repair tags in \mathbb{F}_p is $n\alpha \log_2 p$. Therefore, the storage cost in the RDC-NC scheme is $O(n\alpha \log_2 p(w + s + 1))$. In the ND-POR scheme, the size of $n\alpha$ coded blocks and $(n-l)\beta$ where $\beta = \frac{n\alpha}{l}$ dispersal coding parity blocks in \mathbb{F}_p^w is $w \log_2 p(n\alpha + (n-l)\frac{n\alpha}{l}) = \frac{n^2\alpha}{l} w \log_2 p$. The size of $n\alpha$ repair tags in \mathbb{F}_p is $n\alpha \log_2 p$.

Thus, the storage cost in the ND-POR scheme is $O(n\alpha \log_2 p(\frac{nw}{l} + 1))$. To make the cost in the ND-POR scheme better than the RDC-NC scheme, let $n\alpha \log_2 p(\frac{nw}{l} + 1) < n\alpha \log_2 p(w + s + 1)$. As a result, the parameters should be chosen such that $w < \frac{sl}{n-l}$.

6.5.5 Numerical Examples of The Parameters

In this section, we give two concrete numerical examples of the parameters as follows.

Example 6.5.1. $n = 12, \alpha = 3, s = 5, l = 10, m = 7, w = 20, z = 13, p = 4099, |F| = 1092$. Suppose that all elements in \mathbb{F}_p is less than or equal to 4095. This is to let the elements not exceed 12 bits length. These parameters satisfy the conditions which we stated in the manuscript:

- $1 < \frac{n}{l} < 2$ as stated in Section 6.5.1.
- $l = \lceil \frac{n\alpha(n+1)}{2m+n\alpha} \rceil$ as stated in Section 6.5.3. (In this example, $l = \lceil \frac{12 \cdot 3 \cdot 13}{2 \cdot 7 + 12 \cdot 3} \rceil = 10$).
- $w < \frac{sl}{n-l}$ as stated in Section 6.5.4.

We now show the costs of the RDC-NC and ND-POR schemes.

- The encode computation cost of the RDC-NC scheme is $n\alpha = 12 \cdot 3 = 36$. Meanwhile, the encode computation cost of the ND-POR scheme is $n\alpha \frac{n}{l} = 12 \cdot 3 \cdot \frac{12}{10} = 43.2$.
- The check computation cost of the RDC-NC scheme is $n\alpha = 12 \cdot 3 = 36$. Meanwhile, the check computation cost of the ND-POR scheme is $\frac{n\alpha}{l} = \frac{12 \cdot 3}{10} = 3.6$.
- The repair computation cost of the RDC-NC scheme is $\frac{3m|F|}{m+\alpha} = \frac{3 \cdot 7 \cdot 1092}{7+3} = 2293.2$. Meanwhile, the repair computation cost of the ND-POR scheme is $n \log_2^2 n \log_2 \log_2 n = 284$ (in the case of the RS code), or $\frac{3lm|F|}{n\alpha+lm} = \frac{3 \cdot 10 \cdot 7 \cdot 1092}{12 \cdot 3 + 10 \cdot 7} = 2163.4$ (in the case of the network coding).
- The number of MACs in the RDC-NC scheme is $ns\alpha = 12 \cdot 5 \cdot 3 = 180$. Meanwhile, the number of MACs in the ND-POR scheme is $l\alpha = 10 \cdot 3 = 30$.
- The number of the required healthy servers for data repair in the RDC-NC scheme is $\lceil \frac{m}{\alpha} \rceil = \lceil \frac{7}{3} \rceil = 3$. Meanwhile, that in the ND-POR scheme is $\lceil \frac{lm}{n\alpha} \rceil = \lceil \frac{10 \cdot 7}{12 \cdot 3} \rceil = 2$.
- The storage cost in the RDC-NC scheme is $n\alpha \log_2 p(w + s + 1) = 12 \cdot 3 \cdot \log_2 4099 \cdot (20 + 5 + 1) = 11232$. Meanwhile, the storage cost in the ND-POR scheme is $n\alpha \log_2 p(\frac{nw}{l} + 1) = 12 \cdot 3 \cdot \log_2 4099 \cdot (\frac{12 \cdot 20}{10} + 1) = 10800$.

Example 6.5.2. $n = 16, \alpha = 5, s = 4, l = 14, m = 10, w = 25, z = 15, p = 1031, |F| = 1500$. Suppose that all elements in \mathbb{F}_p is less than or equal to 1023. This is to let the elements not exceed 10 bits length. These parameters satisfy the conditions which we stated in the manuscript:

- $1 < \frac{n}{l} < 2$ as stated in Section 6.5.1.

- $l = \lceil \frac{n\alpha(n+1)}{2m+n\alpha} \rceil$ as stated in Section 6.5.3. (In this example, $l = \lceil \frac{16 \cdot 5 \cdot 17}{2 \cdot 10 + 16 \cdot 5} \rceil = 14$).
- $w < \frac{sl}{n-l}$ as stated in Section 6.5.4.

We now show the costs of the RDC-NC and ND-POR schemes.

- The encode computation cost of the RDC-NC scheme is $n\alpha = 16 \cdot 5 = 80$. Meanwhile, the encode computation cost of the ND-POR scheme is $n\alpha \frac{n}{l} = 16 \cdot 5 \cdot \frac{16}{14} = 91.43$.
- The check computation cost of the RDC-NC scheme is $n\alpha = 16 \cdot 5 = 80$. Meanwhile, the check computation cost of the ND-POR scheme is $\frac{n\alpha}{l} = \frac{16 \cdot 5}{14} = 5.71$.
- The repair computation cost of the RDC-NC scheme is $\frac{3m|F|}{m+\alpha} = \frac{3 \cdot 10 \cdot 1500}{10+5} = 3000$. Meanwhile, the repair computation cost of the ND-POR scheme is $n \log_2^2 n \log_2 \log_2 n = 512$ (in the case of the RS code), or $\frac{3lm|F|}{n\alpha+lm} = \frac{3 \cdot 14 \cdot 10 \cdot 1500}{16 \cdot 5 + 14 \cdot 10} = 2863.64$ (in the case of the network coding).
- The number of MACs in the RDC-NC scheme is $ns\alpha = 16 \cdot 4 \cdot 5 = 320$. Meanwhile, the number of MACs in the ND-POR scheme is $l\alpha = 14 \cdot 5 = 70$.
- The number of the required healthy servers for data repair in the RDC-NC scheme is $\frac{m}{\alpha} = \frac{10}{5} = 2$. Meanwhile, that in the ND-POR scheme is $\frac{lm}{n\alpha} = \frac{14 \cdot 10}{16 \cdot 5} = 1.75$.
- The storage cost in the RDC-NC scheme is $n\alpha \log_2 p(w + s + 1) = 16 \cdot 5 \cdot \log_2 1031 \cdot (25 + 4 + 1) = 24000$. Meanwhile, the storage cost in the ND-POR scheme is $n\alpha \log_2 p(\frac{nw}{l} + 1) = 16 \cdot 5 \cdot \log_2 1031 \cdot (\frac{16 \cdot 25}{14} + 1) = 23657$.

In summary, although the ND-POR scheme combines the network coding and the dispersal coding, the ND-POR scheme is not worse than the RDC-NC scheme in totals.

6.6 Summary

In this chapter, the ND-POR scheme has been proposed in which the network coding and the dispersal coding technique are combined to reduce the costs of two important phases, the check and the repair phases, and to prevent small corruption attack, replay attack, pollution attack and large corruption.

In future work, we focus on the following problem. The repair phase has not been optimized because the healthy servers need to provide the aggregated coded blocks to the client, then the client computes the new coded blocks and stores them on the new server. A new mechanism can be considered in which the healthy servers send their coded blocks directly to the new server without sending back to the client. This mechanism can reduce the burden for the client and also reduce the communication. To support this mechanism, a signature scheme can be employed such as [84, 103] that allows the new server to verify the coded blocks provided from the healthy servers instead of the client, and to construct the new coded blocks by itself.

Chapter 7

SW-SSS: Slepian-Wolf Coding-based SSS

7.1 System Model

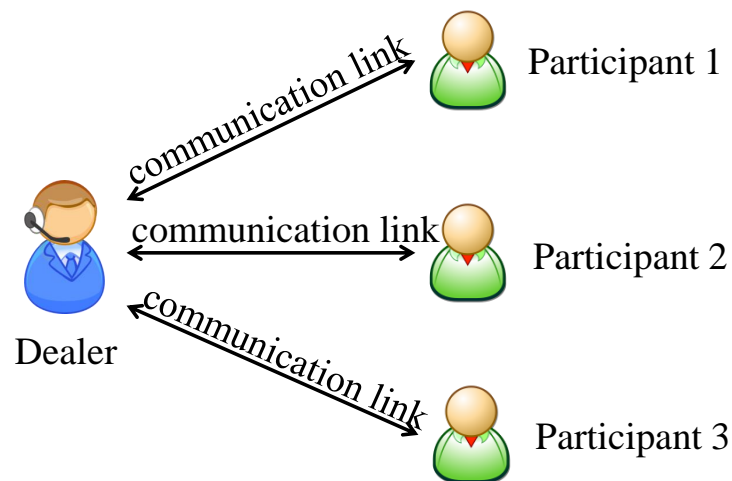


Figure 7.1: System Model of the SW-SSS

In the adversarial model of the SW-SSS scheme, there are two types of entities as depicted in Figure 7.1.

- Dealer: this entity is trusted, and has the following responsibilities:
 - Generate the shares from the secret, then distribute the shares to the participants.

- Collect the shares from the participants to reconstruct the secret when needed.
- Repair the share which is stored in the corrupted participant.
- Participants: these entities are untrusted, and have the following responsibilities:
 - Store the shares (each participant store a share).
 - Provide the shares to the dealer when the dealer needs to reconstruct the secret.
 - Provide the shares to the dealer when the dealer needs to repair a corrupted share.

The notations which are used in our SW-SSS scheme are given in Table 7.1.

Table 7.1: List of notations in the revisited SSS and the SW-SSS

Notation	Description
S	secret
m	number of secret blocks (the first threshold)
b_i	secret block ($i \in \{0, \dots, m-1\}$)
n	number of participants (the second threshold)
L	number of participants whose shares are collectively constructed from m secret blocks.
\mathcal{P}_i	participant ($i \in \{0, \dots, n-1\}$)
c_i	share stored in \mathcal{P}_i
s_i	XOR used to construct c_i ($s_i = b_j \oplus b_t \oplus b_z$)
d_i	metadata of c_i (the number of ‘1’ bits in s_i)
j	index of the first operand of s_i
t	index of the second operand of s_i
z	index of the third operand of s_i
$ b $	bit-size of a secret block ($ b = \frac{ S }{m}$)
\oplus	XOR operator
$ $	concatenation operator

7.2 Revisited XOR Network Coding-based SSS

In this section, we revisit the XOR network coding and apply it to a SSS. Although the XOR-based network coding has been proposed in many previous network coding schemes [135–139], the problem is that none of them applies the XOR network coding to a (m, n) -SSS. Therefore, we revisit it and apply it to a (m, n) SSS as follows.

The dealer firstly divides S into m blocks: $S = b_0 || \dots || b_{m-1}$ ($|b_i| = \frac{|S|}{m}$) and encodes S into n shares. Each participant \mathcal{P}_i holds a share c_i where $i \in \{0, \dots, n-1\}$. To compute c_i , the dealer chooses a number of secret blocks randomly and combines them using the

XOR. The dealer then pads that XOR with a vector of length m which contains a ‘1’ bit in the index of each chosen secret block and $(m - 1)$ ‘0’ bits elsewhere. The padded vector is called the coefficient of c_i . Suppose that c_i is constructed from t secret blocks $b_{i_0}, \dots, b_{i_{t-1}}$. Let $s_i = b_{i_0} \oplus \dots \oplus b_{i_{t-1}}$. c_i has the following form:

$$c_i = (\underbrace{a_0, a_2, \dots, a_{m-1}}_m, \underbrace{s_i}_{\frac{|S|}{m}}) \quad (7.1)$$

where $a_i = 1$ if $i \in \{i_0, \dots, i_{t-1}\}$ and $a_i = 0$ elsewhere. The share size is $|c_i| = m + \frac{|S|}{m}$. The ideal property of a SSS is $|c_i| = |S|$. The revisited XOR network coding-based SSS achieves a better share size if $|c_i| \leq |S|$. From this inequality, $(m - \frac{|S|}{2})^2 \leq \frac{|S|^2}{4} - |S|$. Because $|S|$ is large in a real system, $\frac{|S|^2}{4} - |S| \approx \frac{|S|^2}{4}$. Therefore, $m \leq |S|$. In other words, if the parameters are chosen such that $m \leq |S|$, the scheme can reduce the share size. Moreover, the coefficients are chosen such that the matrix consisting of the coefficients of any m shares has rank m . This condition is to ensure that m secret blocks can be reconstructed from any m shares. To reconstruct S , the dealer chooses any m shares to find m secret blocks, then, concatenates them together. To repair a corrupted share, the dealer requires m healthy shares to reconstitute using the XOR.

Example 7.2.1. Suppose that $S = b_0||b_1||b_2$ and $n = 4$. The dealer creates the following four shares:

- $c_0 = (1, 1, 1, b_0 \oplus b_1 \oplus b_2)$
- $c_1 = (1, 1, 0, b_0 \oplus b_1)$
- $c_2 = (1, 0, 1, b_0 \oplus b_2)$
- $c_3 = (1, 0, 0, b_0)$

The dealer sends $\{c_0, \dots, c_3\}$ to the participants $\{\mathcal{P}_0, \dots, \mathcal{P}_3\}$, respectively. To reconstruct S , the dealer chooses $m = 3$ shares (suppose c_0, c_2, c_3) and constructs the following equation system:

$$\begin{cases} s_0 = b_0 \oplus b_1 \oplus b_2 \\ s_2 = b_0 \oplus b_2 \\ s_3 = b_0 \end{cases} \quad (7.2)$$

Then, $\{b_0, b_1, b_2\}$ are solved using the Gaussian elimination. Finally, S is reconstructed as $S = b_0||b_1||b_2$. Suppose that \mathcal{P}_2 is corrupted, the dealer requires $\mathcal{P}_0, \mathcal{P}_1$ and \mathcal{P}_3 to provide s_0, s_1 and s_3 . The dealer repairs s_2 by $s_2 = s_0 \oplus s_1 \oplus s_3$.

7.3 Proposed SW-SSS

In this scheme, a share c_i does not have the same form as in the revisited XOR network coding-based SSS (Equation 7.1). Instead, c_i is the index of the bin that the XOR belongs

to. This scheme focuses on the share generation, secret reconstruction and share repair. Checking a corrupted participant is beyond the scope of this proposed scheme because several existing schemes using the MAC or signature techniques can be used.

7.3.1 Share Generation

Each XOR is constructed from three different secret blocks. From m secret blocks, there are $\binom{m}{3}$ XORs. However, only n out of $\binom{m}{3}$ XORs are required for n participants. The idea to choose these n XORs as the following remark:

Remark 1. *The dealer chooses each XOR such that the index sequence of the three secret blocks is a permutation of the proper set $\{0, \dots, m-1\}$ in an ascending order. Each XOR itself is also sorted in an ascending order. Namely, the dealer chooses n shares for n participants from the following XORs respectively, until the dealer has enough n XORs:*

$$\begin{aligned} & (b_0 \oplus b_1 \oplus b_2), (b_0 \oplus b_1 \oplus b_3), \dots, (b_0 \oplus b_1 \oplus b_{m-1}), \\ & (b_0 \oplus b_2 \oplus b_3), (b_0 \oplus b_2 \oplus b_4), \dots, (b_0 \oplus b_2 \oplus b_{m-1}), \\ & \dots \\ & (b_1 \oplus b_2 \oplus b_3), (b_1 \oplus b_2 \oplus b_4), \dots, (b_1 \oplus b_2 \oplus b_{m-1}), \\ & \dots \end{aligned}$$

Concretely, the dealer performs the **ShareGen** algorithm which takes m, n and S as the inputs, and outputs n pairs of share c_i and its metadata d_i as follows:

- **ShareGen**(m, n, S) $\rightarrow \{(c_0, d_0), \dots, (c_{n-1}, d_{n-1})\}$
 - Divide the secret into m blocks: $S = b_0 || \dots || b_{m-1}$.
 - Compute the size of a secret block: $|b| = |S|/m$.
 - Set a value: $count \leftarrow 0$.
 - For $\forall j \in \{0, \dots, m-3\}, \forall t \in \{j+1, \dots, m-2\}$ and $\forall z \in \{t+1, \dots, m-1\}$:
 - * Compute a XOR for each share: $s_i \leftarrow b_j \oplus b_t \oplus b_z$.
 - * Find the number of ‘1’ bits in s_i : $d_i \leftarrow s_i.count('1')$. This is also the metadata of the share c_i .
 - * Find the share $c_i \leftarrow \text{FindShare}(|b|, s_i, d_i)$.
 - * Increase $count$ by 1: $count = count + 1$
 - * Check if $(count == n - 1)$, then return $\{(c_0, d_0), \dots, (c_{n-1}, d_{n-1})\}$
- **FindShare**($|b|, s_i, d_i$) $\rightarrow c_i$: this is the sub-algorithm which is used in the **ShareGen** algorithm:
 - Construct a set M_i which consists of all permutations of each XOR, given $|b|$ and d_i . The elements in M_i are sorted in an ascending order.

- Find the corresponding index of s_i in M_i : $c_i \leftarrow \text{index}(M_i, s_i)$. This is the share.

We can observe that the share is not s_i but the index of s_i in the set M_i . The number of elements in M_i is $|M_i| = \binom{|b|}{d_i}$. The number of bits for representing a share is at most $\log_2 |M_i|$. The bandwidth and the storage cost can be reduced because the size of an index is less than the size of a XOR. Thank for the SWC. The **ShareGen** algorithm finally returns $(c_0, d_0), \dots, (c_{n-1}, d_{n-1})$. The dealer distributes $\{c_i, d_i\}$ to the participant \mathcal{P}_i .

Example 7.3.1. Suppose that $S = 10100111001110110001$. S is divided into $m = 5$ blocks: $b_0 = 1010, b_1 = 0111, b_2 = 0011, b_3 = 1011$ and $b_4 = 0001$ ($|S| = 20, |b_i| = 4$). Suppose that $n = 8$, the shares are $\{c_0, \dots, c_7\}$.

To construct c_0 , $s_0 = b_0 \oplus b_1 \oplus b_2 = 1110$ is used. Because the number of ‘1’ bits in s_0 is 3, $d_0 = 3$. Because $d_0 = 3$ and $|b_i| = 4$, $M_0 = \{0111, 1011, 1101, 1110\}$. The elements in M_0 are sorted in an ascending order and are indexed as $\{0, \dots, 3\}$. Because $|M_0| = \binom{4}{3} = 4$, at most $\log_2 4 = 2$ bits are required to represent c_0 instead of 4 bits of s_0 . Because the index of s_0 in M_0 is 3, $c_0 = 3_{\text{decimal}} = 11_{\text{binary}}$. $\{c_0, d_0\}$ are sent to the participant \mathcal{P}_0 . Similarly, $\{c_1, \dots, c_7\}$ are computed as follows.

i	s_i	d_i	M_i	c_i
0	$b_0 \oplus b_1 \oplus b_2 = 1110$	3	$\{0111, 1011, 1101, 1110\}$	$3_{\text{decimal}} = 11_{\text{binary}}$
1	$b_0 \oplus b_1 \oplus b_3 = 0110$	2	$\{0011, 0101, 0110, 1001, 1010, 1100\}$	$2_{\text{decimal}} = 10_{\text{binary}}$
2	$b_0 \oplus b_1 \oplus b_4 = 1100$	2	$\{0011, 0101, 0110, 1001, 1010, 1100\}$	$5_{\text{decimal}} = 101_{\text{binary}}$
3	$b_0 \oplus b_2 \oplus b_3 = 0010$	1	$\{0001, 0010, 0100, 1000\}$	$1_{\text{decimal}} = 1_{\text{binary}}$
4	$b_0 \oplus b_2 \oplus b_4 = 1000$	1	$\{0001, 0010, 0100, 1000\}$	$3_{\text{decimal}} = 11_{\text{binary}}$
5	$b_0 \oplus b_3 \oplus b_4 = 0000$	0	$\{\}$	$0_{\text{decimal}} = 0_{\text{binary}}$
6	$b_1 \oplus b_2 \oplus b_3 = 1111$	4	$\{1111\}$	$0_{\text{decimal}} = 0_{\text{binary}}$
7	$b_1 \oplus b_2 \oplus b_4 = 0101$	2	$\{0011, 0101, 0110, 1001, 1010, 1100\}$	$1_{\text{decimal}} = 1_{\text{binary}}$

$\{c_i, d_i\}$ are then sent to \mathcal{P}_i where $i \in \{0, \dots, 7\}$.

7.3.2 Secret Reconstruction

To reconstruct S , the dealer requires m participants to provide their shares (suppose $c_{k_0}, \dots, c_{k_{m-1}}$). These shares are chosen such that the binary matrix consisting of the coefficient vectors of the XORs has full rank. Concretely, the dealer performs the secret reconstruction **Reconst** algorithm which takes m pairs of (c_{k_i}, d_{k_i}) as the inputs, and outputs m secret blocks $\{b_0, \dots, b_{m-1}\}$ as follows:

- **Reconst** $((c_{k_0}, d_{k_0}), \dots, (c_{k_{m-1}}, d_{k_{m-1}})) \rightarrow \{b_0, \dots, b_{m-1}\}$:
 - For $\forall i \in \{0, \dots, m-1\}$:
 - * Find the XOR for c_{k_i} : $s_{k_i} \leftarrow \text{FindXOR}(|b|, d_{k_i}, c_{k_i})$.

- * Find the indices of three operands of s_{k_i} : $(j_{k_i}, t_{k_i}, z_{k_i}) \leftarrow \text{LocateIndices}(m, k_i)$.
 - * Construct a vector v_{k_i} which consists of $(m+1)$ elements: m first elements are the binary coefficients of m secret blocks and the finally element is s_{k_i} . Namely, $v_{k_i} = [e_0, e_1, \dots, e_{m-1}, s_{k_i}]$ where $e_x \in \{0, 1\}$ for $x = 0, \dots, m-1$. $e_x = 1$ when x is the index of each operand in s_{k_i} (j_{k_i} , t_{k_i} and z_{k_i}). $e_x = 0$ elsewhere.
 - * Construct a matrix Q in which each vector v_{k_i} is a row of the matrix Q : $Q \leftarrow [v_{k_0}, v_{k_1}, \dots, v_{k_{m-1}}]^T$.
 - * Execute the Gaussian elimination on Q to obtain a matrix Q' .
 - * Filter m unknowns $\{b_0, \dots, b_{m-1}\}$ from Q' .
 - * Reconstruct the secret by concatenating all the secret blocks: $S = b_0 || \dots || b_{m-1}$.
- **FindXOR**($|b|, d_{k_i}, c_{k_i}$) $\rightarrow s_{k_i}$: this is the sub-algorithm which is used in the **Reconst** algorithm:
 - Lists all permutations given $|b|$ and d_{k_i} .
 - Find the XOR s_{k_i} by picking the c_{k_i} -th element in the set M_{k_i} .
 - **LocateIndices**(m, k_i) $\rightarrow \{j_{k_i}, t_{k_i}, z_{k_i}\}$: this is sub-algorithm which is used in the **Reconst** algorithm:
 - Set a value: $count \leftarrow -1$.
 - For $\forall j \in \{0, \dots, m-3\}$, $\forall t \in \{j+1, \dots, m-2\}$, $\forall z \in \{t+1, \dots, m-1\}$:
 - * Increase $count$ by 1: $count = count + 1$.
 - * Check if $(count == k_i)$, then set:
 - $j_{k_i} \leftarrow j$
 - $t_{k_i} \leftarrow t$
 - $z_{k_i} \leftarrow z$

Example 7.3.2. This example follows Example 7.3.1. Suppose that $\{c_1, c_3, c_5, c_6, c_7\}$ are chosen to reconstruct S because the matrix consisting of the coefficient vectors of $\{s_1, s_3, s_5, s_6, s_7\}$ has full rank. Because $|b| = 4$ and $d_1 = 2$, $M_1 = \{0011, 0101, 0110, 1001, 1010, 1100\}$. Because the element whose index in M_1 is $c_1 = 10_{binary} = 2_{decimal}$ is 0110, $s_1 = 0110$. Similarly, $s_3 = 0010$, $s_5 = 0000$, $s_6 = 1111$ and $s_7 = 0101$. A vector v_{k_i} is then constructed for each c_{k_i} . Because $s_1 = b_0 \oplus b_1 \oplus b_3 = 0110$, $(j_1, t_1, z_1) = (0, 1, 3)$. Therefore, $v_1 = [1, 1, 0, 1, 0, 0110]$. Similarly, $v_3 = [1, 0, 1, 1, 0, 0010]$, $v_5 = [1, 0, 0, 1, 1, 0000]$, $v_6 = [0, 1, 1, 1, 0, 1111]$ and $v_7 = [0, 1, 1, 0, 1, 0101]$. The matrix Q is constructed as follows:

$$Q = \begin{pmatrix} v_1 \\ v_3 \\ v_5 \\ v_6 \\ v_7 \end{pmatrix} = \left(\begin{array}{cccc|c} 1 & 1 & 0 & 1 & 0 & 0110 \\ 1 & 0 & 1 & 1 & 0 & 0010 \\ 1 & 0 & 0 & 1 & 1 & 0000 \\ 0 & 1 & 1 & 1 & 0 & 1111 \\ 0 & 1 & 1 & 0 & 1 & 0101 \end{array} \right) \quad (7.3)$$

We apply Gauss-elimination on Q to obtain Q' :

$$Q' = \left(\begin{array}{ccccc|c} 1 & 0 & 0 & 0 & 0 & 1010 \\ 0 & 1 & 0 & 0 & 0 & 0111 \\ 0 & 0 & 1 & 0 & 0 & 0011 \\ 0 & 0 & 0 & 1 & 0 & 1011 \\ 0 & 0 & 0 & 0 & 1 & 0001 \end{array} \right) \quad (7.4)$$

From Q' , the secret blocks are obtained as: $b_0 = 1010, b_1 = 0111, b_2 = 0011, b_3 = 1011$ and $b_4 = 0001$. Finally, S is reconstructed as $S = b_0 || \dots || b_4$.

7.3.3 Share Repair

Suppose that the participant \mathcal{P}_{corr} is corrupted. The dealer finds only three shares which belongs to three participants to repair \mathcal{P}_{corr} . The idea to find these three shares is given in the following remark:

Remark 2. Let s_{corr} denote the XOR used to construct the share c_{corr} in \mathcal{P}_{corr} . $s_{corr} = b_{j_{corr}} \oplus b_{t_{corr}} \oplus b_{z_{corr}}$. We choose two integers $\alpha, \beta \in \{0, \dots, m-1\}$ such that $\alpha, \beta \neq j_{corr}, t_{corr}, z_{corr}$. The idea to find three shares to repair \mathcal{P}_{corr} is that:

$$\begin{aligned} b_{j_{corr}} \oplus b_{t_{corr}} \oplus b_{z_{corr}} &= (b_{j_{corr}} \oplus b_{t_{corr}} \oplus b_{\alpha}) \\ &\oplus (b_{j_{corr}} \oplus b_{z_{corr}} \oplus b_{\beta}) \\ &\oplus (b_{j_{corr}} \oplus b_{\alpha} \oplus b_{\beta}) \end{aligned}$$

The three shares which will be used to repair \mathcal{P}_{corr} is the shares that are constructed from the XORs: $(b_{j_{corr}} \oplus b_{t_{corr}} \oplus b_{\alpha})$, $(b_{j_{corr}} \oplus b_{z_{corr}} \oplus b_{\beta})$ and $(b_{j_{corr}} \oplus b_{\alpha} \oplus b_{\beta})$.

Concretely, the dealer performs the **ShareRepair** algorithm which takes \mathcal{P}_{corr} as the inputs, and outputs the share c_{corr} and the metadata d_{corr} of \mathcal{P}_{corr} as follows.

- **ShareRepair**(\mathcal{P}_{corr}) $\rightarrow \{c_{corr}, d_{corr}\}$:
 - Find indices of the XOR $(b_{j_{corr}} \oplus b_{t_{corr}} \oplus b_{z_{corr}})$:

$$(j_{corr}, t_{corr}, z_{corr}) \leftarrow \text{LocateIndices}(m, corr).$$
 - Choose $\alpha, \beta \in \{0, \dots, m-1\}$ such that $\alpha, \beta \neq j_{corr}, t_{corr}, z_{corr}$.
 - Sort $\{j_{corr}, t_{corr}, \alpha\}$, $\{j_{corr}, z_{corr}, \beta\}$ and $\{j_{corr}, \alpha, \beta\}$ in an ascending orders. Let $\{j_{r_1}, t_{r_1}, z_{r_1}\}$, $\{j_{r_2}, t_{r_2}, z_{r_2}\}$ and $\{t_{r_3}, t_{r_3}, z_{r_3}\}$ denote the results of these sorts, respectively.
 - Find three participants used for the repair:
 - * $\mathcal{P}_{r_1} \leftarrow \text{LocateParticipant}(j_{r_1}, t_{r_1}, z_{r_1})$.
 - * $\mathcal{P}_{r_2} \leftarrow \text{LocateParticipant}(j_{r_2}, t_{r_2}, z_{r_2})$.
 - * $\mathcal{P}_{r_3} \leftarrow \text{LocateParticipant}(j_{r_3}, t_{r_3}, z_{r_3})$.

- Require $\mathcal{P}_{r_1}, \mathcal{P}_{r_2}$ and \mathcal{P}_{r_3} to provide $\{c_{r_1}, d_{r_1}\}, \{c_{r_2}, d_{r_2}\}$ and $\{c_{r_3}, d_{r_3}\}$, respectively.
- Find the XORs (s_{r_1}, s_{r_2} and s_{r_3}) by picking the c_{r_1} -th, c_{r_2} -th and c_{r_3} -th element in the set M_{r_1}, M_{r_2} and M_{r_3} , respectively:
 - * $s_{r_1} \leftarrow \text{FindXOR}(|b|, d_{r_1}, c_{r_1})$.
 - * $s_{r_2} \leftarrow \text{FindXOR}(|b|, d_{r_2}, c_{r_2})$.
 - * $s_{r_3} \leftarrow \text{FindXOR}(|b|, d_{r_3}, c_{r_3})$
- Recover s_{corr} as: $s_{corr} \leftarrow s_{r_1} \oplus s_{r_2} \oplus s_{r_3}$.
- Find the metadata d_{corr} by counting the number of ‘1’ bits in s_{corr} .
- Compute the share c_{corr} using the FindShare sub-algorithm like the share generation algorithm.
- **LocateParticipant**($j_{r_i}, t_{r_i}, z_{r_i}$) $\rightarrow \mathcal{P}_{r_i}$: this is the sub-algorithm which is used in the ShareRepair algorithm:
 - Set a value: $count = -1$.
 - For $\forall j \in \{0, \dots, m-3\}, \forall t \in \{j+1, \dots, m-2\}, \forall z \in \{t+1, \dots, m-1\}$:
 - * Increase $count$ by 1: $count = count + 1$.
 - * Check if: $(j == j_{r_i})$ and $(t == t_{r_i})$ and $(z == z_{r_i})$, then return $count$.

Example 7.3.3. This example follows Example 7.3.1 and Example 7.3.2. Suppose that \mathcal{P}_4 is corrupted. $\{j_{corr}, t_{corr}, z_{corr}\} = \{0, 2, 4\}$ because \mathcal{P}_4 uses $b_0 \oplus b_2 \oplus b_4$ to construct its share. Choose $\alpha = 1$ and $\beta = 3$ because $1, 3 \neq 0, 2, 4$.

$$\begin{aligned}
 b_0 \oplus b_2 \oplus b_4 &= (b_0 \oplus b_2 \oplus b_1) \\
 &\oplus (b_0 \oplus b_4 \oplus b_3) \\
 &\oplus (b_0 \oplus b_1 \oplus b_3)
 \end{aligned}$$

Let $\{j_{r_1}, t_{r_1}, z_{r_1}\} = \{0, 1, 2\}$, $\{j_{r_2}, t_{r_2}, z_{r_2}\} = \{0, 3, 4\}$ and $\{j_{r_3}, t_{r_3}, z_{r_3}\} = \{0, 1, 3\}$. The participants are chosen for repairing \mathcal{P}_4 as follows:

- Given $\{j_{r_1}, t_{r_1}, z_{r_1}\} = \{0, 1, 2\}$, \mathcal{P}_0 is chosen because \mathcal{P}_0 uses $(b_0 \oplus b_1 \oplus b_2)$.
- Given $\{j_{r_2}, t_{r_2}, z_{r_2}\} = \{0, 3, 4\}$, \mathcal{P}_5 is chosen because \mathcal{P}_5 uses $(b_0 \oplus b_3 \oplus b_4)$.
- Given $\{j_{r_3}, t_{r_3}, z_{r_3}\} = \{0, 1, 3\}$, \mathcal{P}_1 is chosen because \mathcal{P}_1 uses $(b_0 \oplus b_1 \oplus b_3)$.

The participants $\mathcal{P}_0, \mathcal{P}_5$ and \mathcal{P}_1 are then required to provide $(c_0, d_0), (c_5, d_5)$ and (c_1, d_1) to the dealer. The XORs used for the repair are found as follows:

- Given $(c_0, d_0) = (11, 3)$, the dealer finds $s_0 = 1110$.
- Given $(c_5, d_5) = (0, 0)$, the dealer finds $s_5 = 0000$.

- Given $(c_1, d_1) = (10, 2)$, the dealer finds $s_1 = 0110$.

The dealer computes $s_{corr} = s_0 \oplus s_5 \oplus s_1 = 1000$. Because the number of ‘1’ bits in 1000 is 1, $d_{corr} = 1$. Because $d_{corr} = 1$ and $|b| = 4$, $M_{corr} = \{0001, 0010, 0100, 1000\}$. Because $|M_{corr}| = 4$, at most $\log_2 4 = 2$ bits are required to represent c_{corr} . The share c_{corr} is the index of s_{corr} in M_{corr} , which is $c_{corr} = 3_{decimal} = 11_{binary}$.

7.4 Secrecy and Share Size

This section analyses two properties of our proposed SW-SSS scheme.

7.4.1 Secrecy

We firstly consider the secret reconstruction condition. Let *epoch* be a time step in which the participants are checked. If a corrupted participant is detected, it will be repaired in the epoch.

Theorem 19. *The secret S can be reconstructed if in any epoch, at least m out of n participants are healthy, and the matrix consisting of the coefficient vectors of the XORs has full rank.*

Proof. $S = b_0 || \dots || b_{m-1}$. To reconstruct S , we view m secret blocks (b_0, \dots, b_{m-1}) as m unknowns that need to be solved. To solve these unknowns, at least m shares $(c_{i_1}, \dots, c_{i_{m-1}})$ along with their metadata $(d_{i_1}, \dots, d_{i_{m-1}})$ are required to make the matrix have full rank.

$$\begin{cases} s_{i_0} = b_{j_0} \oplus b_{t_0} \oplus b_{z_0} \\ s_{i_1} = b_{j_1} \oplus b_{t_1} \oplus b_{z_1} \\ \dots \\ s_{i_{m-1}} = b_{j_{m-1}} \oplus b_{t_{m-1}} \oplus b_{z_{m-1}} \end{cases} \quad (7.5)$$

Therefore, the number of required participants is at least m , in order to ensure that the equation system is solvable. Theorem 19 yields to a constrain between n and m that $n > m$. Because each of n shares is constructed from any three out of m secret blocks, another constraint between n and m is that $n \leq \binom{m}{3}$. Therefore, m and n should be

chosen such that $m < n \leq \binom{m}{3}$. □

The secrecy is now analysed as follows. Let $H(S)$ be the entropy of the random variable which is induced by S . Let L denote the number of participants whose shares are collectively constructed from m secret blocks. The secrecy of the SW-SSS scheme is given in the following theorem:

Theorem 20. *The secrecy of t random variables $\{C_{i_1}, \dots, C_{i_t}\}$ which represent any t shares $\{c_{i_0}, \dots, c_{i_{t-1}}\}$ and t random variables $\{D_{i_0}, \dots, D_{i_{t-1}}\}$ which represent any t metadata $\{d_{i_0}, \dots, d_{i_{t-1}}\}$ is:*

$$H(S|(C_{i_0}, D_{i_0}), \dots, (C_{i_{t-1}}, D_{i_{t-1}})) = \begin{cases} H(S), & \text{if } t < L \\ \frac{m-t}{m}H(S), & \text{if } L \leq t < m \\ 0, & \text{if } m \leq t \end{cases} \quad (7.6)$$

Proof. s_i is the original coded sequence (the XOR) that can be uniquely determined by the share c_i and its metadata d_i . From the property of the conditional entropy:

$$H(S|(c_{i_0}, d_{i_0}), \dots, (c_{i_{t-1}}, d_{i_{t-1}})) \leq H(S|s_{i_0}, \dots, s_{i_{t-1}}) \quad (7.7)$$

The equality holds if S is uniformly distributed. For each case of t , the secrecy is given as follows:

- Case 1 ($t < L$): $\{s_{i_0}, \dots, s_{i_{t-1}}\}$ are constructed from inadequate m secret blocks b_0, \dots, b_{m-1} . The matrix consisting of the coefficient vectors of s_{i_j} does not have full rank. Thus, $H(S|s_{i_0}, \dots, s_{i_{t-1}}) = H(S)$. This yields:

$$H(S|(C_{i_0}, D_{i_0}), \dots, (C_{i_{t-1}}, D_{i_{t-1}})) = H(S). \quad (7.8)$$

- Case 2 ($L \leq t$): the matrix consisting of the coefficient vectors of s_{i_j} has rank t . Thus, $H(S|s_{i_0}, \dots, s_{i_{t-1}}) = \frac{m-t}{m}H(S)$. This yields:

$$H(S|(C_{i_0}, D_{i_0}), \dots, (C_{i_{t-1}}, D_{i_{t-1}})) = \frac{m-t}{m}H(S) < H(S) \quad (7.9)$$

- Case 3 ($m \leq t$): from (2), $\frac{m-t}{m}H(S) = 0$. This yields:

$$H(S|(C_{i_0}, D_{i_0}), \dots, (C_{i_{t-1}}, D_{i_{t-1}})) = 0 \quad (7.10)$$

This completes the proof. \square

7.4.2 Share Size

The share sizes in the previous schemes (Shamir-SSS, XOR-SSS, NC-SSS), the revisited XOR network coding-based SSS and the SW-SSS are given as follows:

- In the previous schemes, the share size is $|S| = m \cdot |b|$ (ideal SSS).
- In the revisited XOR network coding-based SSS, the share size is $(m + |b|)$ as mentioned in Section 7.2.
- In the SW-SSS, the share size is at most $\log_2 \binom{|b|}{d_i}$ as mentioned in Section 7.3.1.

For $\forall |b|$ and $\forall d_i \in \{0, \dots, |b|\}$, we have $\log_2 \binom{|b|}{d_i} < |b|$. Thus, $\log_2 \binom{|b|}{d_i} = \frac{|b|}{x}$ for a certain $x > 1$. It is clear that $m + |b| > \frac{|b|}{x}$.

7.5 Efficiency Analysis

Let (\times) and (\oplus) denote the complexity of a multiplication operation in a finite field and the complexity of a XOR, respectively. The \oplus operation is much faster than \times operation (denoted by $(\times) \gg (\oplus)$). The efficiency comparison between the previous schemes (Shamir-SSS [110], XOR-SSS [130], NC-SSS [132]), the revisited XOR network coding-based SSS and the SW-SSS scheme is given in Table 7.2.

7.5.1 Storage Cost

The storage costs in the previous schemes (Shamir-SSS, XOR-SSS, NC-SSS), the revisited XOR network coding-based SSS and the SW-SSS are given as follows:

- In the previous schemes, the storage cost is the same as the share size because each \mathcal{P}_i only stores a share. Namely, the storage cost is $O(m|b|)$.
- In the revisited XOR network coding-based SSS, similar to the previous schemes, the storage cost is the same as the share size because each \mathcal{P}_i only stores a share. Namely, the storage cost is $O(m|b|)$.
- In the SW-SSS, each \mathcal{P}_i stores the share c_i ($|c_i| = \log_2 \binom{|b|}{d_i}$) and the metadata d_i ($|d_i| = \log_2 |b|$ because $d_i \in [1, |b|]$). Therefore, the storage cost is $O(\log_2 \binom{|b|}{d_i} + \log_2 |b|)$. For $\forall |b|$ and $\forall d_i \in [0, |b|]$, $\log_2 \binom{|b|}{d_i} < |b|$. Thus, $\log_2 \binom{|b|}{d_i} = \frac{|b|}{x}$ for some $x > 1$, and $(\frac{|b|}{x} + \log_2 |b|) < (|b| + m)$ if $\log_2 |b| < m$. This inequality holds if the parameters are chosen such that $|b| < 2^m$. If S is divided such that any three blocks are different in the same number of bits (d_0, \dots, d_{n-1} are the same), \mathcal{P}_i does not need to store d_i because it becomes a shared information. The storage cost is thus the same as the share size.

7.5.2 Computation Cost

Share Generation. The computation costs of the share generation algorithm in the previous schemes (Shamir-SSS, XOR-SSS, NC-SSS), the revisited XOR network coding-based SSS and the SW-SSS are given as follows:

- In the Shamir-SSS, each share is computed from a polynomial on multiplications. Thus, the computation cost is $O(n \log n)(\times)$.
- In the XOR-SSS scheme, the computation cost is $O(n_p n)(\oplus)$ where n_p is the smallest prime such that $n_p \geq n$.
- In the NC-SSS, n shares are computed from a linear combination of m secret blocks. Thus, the computation cost is $O(mn)(\times)$.

Table 7.2: Efficiency comparison between the SW-SSS and previous schemes

	Previous schemes			This thesis	
	Shamir-SSS	XOR-SSS	NC-SSS	Revisited SSS	SW-SSS
Feature	XOR-based	No	Yes	No	Yes
	Arbitrary threshold	Yes	Yes	Yes	Yes
	Direct share repair	No	No	Yes	Yes
Storage		$O(m b)$	$O(m b)$	$O(m + b)$	$O(\frac{ b }{x} + \log_2 b)$
Computation	ShareGen	$O(n \log n)(\times)$	$O(n_p n)(\oplus)$	$O(mn)(\oplus)$	$O(n)(\oplus)$
	Reconst	$O(m^2)(\times)$	$O(n_p^2)(\oplus)$	$O(m^2)(\oplus)$	$O(m^2)(\oplus)$
	ShareRepair	N/A	N/A	$O(m)(\oplus)$	$O(1)(\oplus)$
Communication	ShareGen	$O(nm b)$	$O(nm b)$	$O(n(m + b))$	$O(n(\frac{ b }{x} + \log_2 b))$
	Reconst	$O(m^2 b)$	$O(m^2 b)$	$O(m(m + b))$	$O(m(\frac{ b }{x} + \log_2 b))$
	ShareRepair	N/A	N/A	$O(m(m + b))$	$O(\frac{ b }{x} + \log_2 b)$

- In the revisited XOR network coding-based SSS, the dealer also computes the shares as in the NC-SSS. However, it uses the XOR instead of a linear combination over field multiplications. Thus, the computation cost is $O(mn)(\oplus)$.
- In the SW-SSS, n shares are computed from the XORs of a tuple of three secret blocks. Thus, the computation cost is $O(n)(\oplus)$.

Secret Reconstruction. The computation costs of the secret reconstruction algorithm in previous schemes (Shamir-SSS, XOR-SSS, NC-SSS), the revisited XOR network coding-based SSS and the SW-SSS are given as follows:

- In the Shamir-SSS, the Gaussian elimination on multiplications is applied to solve the secret and $(m - 1)$ coefficients. Thus, the computation cost is $O(m^2)(\times)$.
- In the XOR-SSS, the dimension of the matrix used for the Gaussian elimination is $(n_p \times n_p)$, not $(m \times m)$ where n_p is the smallest prime such that $n_p \geq n$. Thus, the computation cost is $O(n_p^2)(\oplus)$.
- In the NC-SSS, the Gaussian elimination on multiplications is applied to solve m coefficients. Thus, the computation cost is $O(m^2)(\times)$.
- In the revisited XOR network coding-based SSS, the Gaussian elimination on XORs is applied to solve m coefficients. Thus, the computation cost is $O(m^2)(\oplus)$.
- In the SW-SSS, the computation cost is the same as the revisited XOR network coding-based SSS. Namely, the computation cost is $O(m^2)(\oplus)$.

Share Repair. The computation costs of the share repair algorithms in the previous schemes (Shamir-SSS, XOR-SSS, NC-SSS), the revisited XOR network coding-based SSS and the SW-SSS are given as follows:

- In the Shamir-SSS, the share repair is not supported. Thus, the computation cost is N/A.
- In the XOR-SSS, the share repair is not supported. Thus, the computation cost is N/A.
- In the NC-SSS, a corrupted share is repaired using m healthy shares using the field linear combinations. The computation cost is $O(m)(\times)$.
- In the revisited XOR network coding-based SSS, a corrupted share is repaired using m healthy shares using the XORs. The computation cost is $O(m)(\oplus)$.
- In the SW-SSS, a new share is computed from three healthy shares. Thus, the computation cost is $O(1)$.

7.5.3 Communication Cost

Share Generation. During the share generation, the dealer distributes n share to n participants. The communication costs in the previous schemes (Shamir-SSS, XOR-SSS, NC-SSS), the revisited XOR network coding-based SSS and the SW-SSS are given as follows:

- In the previous schemes, the cost is $O(nm|b|)$ because the share size is $m|b|$.
- In the revisited XOR network coding-based SSS, the cost is $O(n(m + |b|))$ because the share size is $(m + |b|)$.
- In the SW-SSS, the cost is $O(n(\frac{|b|}{x} + \log_2 |b|))$ because the size of a share and its metadata is $(\frac{|b|}{x} + \log_2 |b|)$.

Secret Reconstruction. In the previous schemes (Shamir-SSS, XOR-SSS, NC-SSS), the revisited XOR network coding-based SSS and the SW-SSS, during the secret reconstruction, m participants are required to provide their shares to the dealer. The communication cost in each scheme is thus m times the storage cost of that scheme. Concretely:

- In the previous schemes, the communication cost is $O(m^2|b|)$.
- In the revisited XOR network coding-based SSS, the communication cost is $O(m(m + |b|))$.
- In the SW-SSS, the communication cost is $O(m(\frac{|b|}{x} + \log_2 |b|))$.

Share Repair. The communication costs of the share repair in the previous schemes (Shamir-SSS, XOR-SSS, NC-SSS), the revisited XOR network coding-based SSS and the SW-SSS are given as follows:

- In the Shamir-SSS, the share repair is not supported. Thus, the communication cost is N/A.
- In the XOR-SSS, the share repair is not supported. Thus, the communication cost is N/A.
- In the NC-SSS, m healthy participants are required to provide their shares to the dealer for the share repair. The size of a share is $m|b|$. Thus, the communication cost is $O(m^2|b|)$.
- In the revisited XOR network coding-based SSS, m healthy participants are required to provide their shares to the dealer for the share repair. The size of a share is $(m + |b|)$. Thus, the communication cost is $O(m(m + |b|))$.
- In the SW-SSS, only three healthy shares are required for the share repair. The size of a share and its metadata is $(\frac{|b|}{x} + \log_2 |b|)$. Thus, the communication cost is $O(\frac{|b|}{x} + \log_2 |b|)$.

7.6 Implementation

7.6.1 Speeding up the FindShare algorithm

To find the share c_i , we need to use the FindShare algorithm in which we construct a list M_i consisting of all permutations and then find the index of the XOR s_i in M_i . For example, in the example of the share generation, the XOR $s_0 = 1110$, the metadata $d_0 = 3$, the secret size is $|b| = 4$. To find c_0 , we construct $M_0 = \{0111, 1011, 1101, 1110\}$ and then find the index of s_0 in M_0 , which is 3. Then, $c_0 = 3$.

In our implementation, we use another way to find c_i given $s_i, d_i, |b|$ without the need to construct M_i . We describe it in the FindShare.New algorithm which takes $(|b|, d_i, s_i)$ as the inputs and outputs c_i as follows:

- FindShare.New($|b|, d_i, s_i$) $\rightarrow c_i$:
 - Set a value $res \leftarrow 0$.
 - Set a value $mask$ as the bits shifted to the left by $(|b| - 1)$ places: $mask = 1 \ll (|b| - 1)$.
 - For $\forall l \in \{|b|, \dots, 1\}$:
 - * Check if the *bitwise and* operation between s_i and $mask$ is not 0, then:
 - Update res by: $res = res + gmpy2.Combination(l - 1, d_i)$.
 - Decrease d_i by 1: $d_i = d_i - 1$
 - * Shift $mask$ to the right by 1 place: $mask \gg= 1$
 - * Decrease l by 1: $l = l - 1$
 - Return res

In the implementation, we use the *gmpy2* library embedded in Python to compute a combination operation as mentioned later in Section 7.6.3. This algorithm is $O(n)$ where n is the number of bits (and not the length of the list).

7.6.2 Speeding up the FindXOR algorithm

To find the XOR s_i , we need to use the FindXOR algorithm in which we construct a list M_i consisting of all permutations and then map the share c_i (which also means the index) in M_i . For example, in the example of the secret reconstruction, the share $c_1 = 2$, the metadata $d_1 = 2$, the secret size is $|b| = 4$. To find c_1 , we construct $M_1 = \{0011, 0101, 0110, 1001, 1010, 1100\}$ and then find the element whose index in M_1 is 2, which is 0110. Then, $s_1 = 0110$.

In our implementation, we use another way to find s_i given $c_i, d_i, |b|$ without the need to construct M_i . We describe it in the FindXOR.New algorithm. The key idea of this algorithm is introduced as follows. Let Z denote the number of ‘1’ bits (which is d_i), and let N denote the number of ‘0’ bits (which is $|b| - d_i$). We know that the number of permutations starting with ‘0’ is equal to the number of permutations of $(N - 1)$ ‘0’s and Z ‘1’s, let’s call it K .

- If $c_i > K$, the permutation starts with '1', c_i remains the same.
- If $c_i \leq K$, the permutation starts with '0', remove K from c_i .

Fix the first bit and loop this process with $c_i = c_i - K$ and the correct number of '0's and '1's. This algorithm is also $O(n)$ where n is the number of bits (and not the length of the list).

Concretely, the `FindXOR_New` algorithm takes $(|b|, d_i, c_i)$ as the inputs, and outputs s_i as follows:

- `FindXOR_New` $(|b|, d_i, c_i) \rightarrow s_i$:
 - Set $Z = d_i$ which is the number of '1' bits.
 - Set $N = |b| - Z$ which is the number of '0' bits.
 - Construct a list $lst = [0...01...1]$ which consists of N '0' bits and Z '1' bits.
 - Set $l \leftarrow lst$
 - Set *result* as an empty string
 - For $i \in \{0, \dots, len(lst)\}$:
 - * Compute K as a combination between the length of l and Z : $K = \binom{len(l) - 1}{Z}$.
 - * Check if $(c_i < K)$, then:
 - Pad a '0' bit in the final position of *result*.
 - Remove a '0' bit in l .
 - * Else:
 - Pad a '1' bit in the final position of *result*.
 - Remove a '1' bit in l .
 - Decrease Z by 1: $Z = Z - 1$.
 - Update $c_i = c_i - K$
 - * Decrease i by 1: $i = i - 1$
 - Pad the first element of l to *result*: $result += l[0]$.
 - Return *result*.

7.6.3 Performance Evaluation

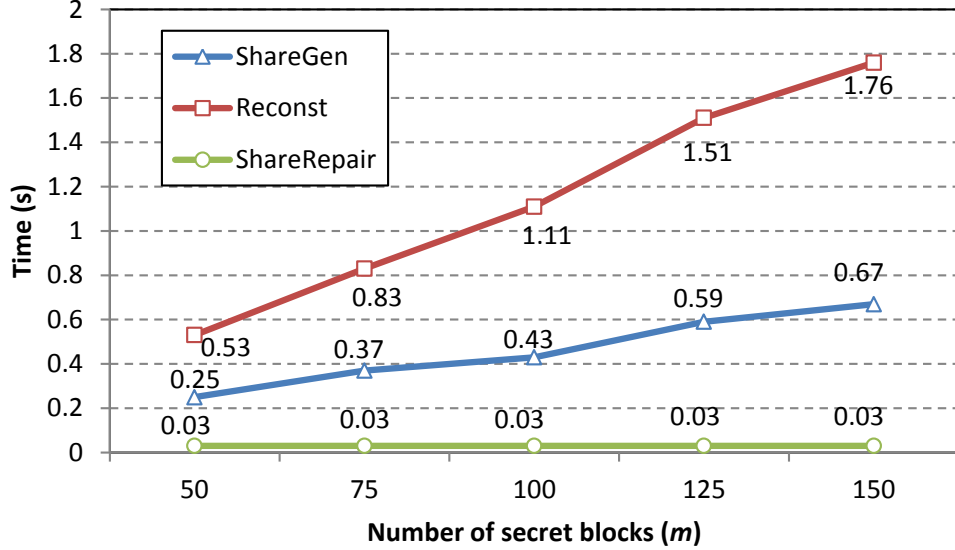


Figure 7.2: Computation performance of the SW-SSS scheme

We show the implementation of the SW-SSS scheme to prove that it is applicable to a real system. The program that is written by Python 2.7.3 is executed using a computer with Intel Core i5, 2.40 GHz, 4 GB of RAM, Win 7 64-bit OS. The `gmpy2` library is used. Each result is the average of 100 runs. The parameters m and n are set as follows:

- Each secret block, b_i , is set to be 2^{10} bits.
- The number of participants, n , is set to be $n = m + 1$.

The simulation results in Figure 7.2 are observed with three sets of the computation performance: **ShareGen**, **Reconst**, and **ShareRepair** by varying the number of secret blocks, m . The graphs reveal that the computation time of the **ShareRepair** algorithm is almost constant and is independent on m . The computation time of the **ShareGen** and **Reconst** algorithms linearly increases with m . The average slopes of increment in the **ShareGen** and **Reconst** algorithms are 0.004 and 0.012, respectively. From these results, if the secret size, $|S|$, is 2^{26} bits ($m = 65,500$), which is almost an upper bound of the size of a secret (e.g., secret key, digital signature), the computation time of the **ShareGen**, **Reconst** and **ShareRepair** algorithms is merely 374.72 seconds (6 minutes), 905.28 seconds (15.1 minutes) and 0.031 seconds, respectively.

7.7 Summary

In this chapter, we firstly revisit the XOR-based NC and then apply it for the SSS in order to support the share repair, in order to obtain the arbitrary parameters, and in order to reduce the share size. This chapter then proposes the main SSS to optimize the share size, named the SW-SSS. The key idea is based on the binning idea of the SWC, which is commonly used in data compression in a network. The security analysis is provided based on the entropy theory. The efficiency analysis is discussed based on the complexity theory. The simulation results of the SW-SSS scheme reveal that it is applicable to a real SSS system. Future research is required to investigate the implementation of the previous schemes.

Chapter 8

Conclusion and Future works

This chapter concludes our thesis by summing up what have been done and what contributions were achieved. At the end of this chapter, we discuss some limits of the study and point out suggestions for future research.

8.1 Conclusion

In this thesis, we propose three schemes as follows:

- MD-POR: this is our main proposed POR which has the following contributions:
 - *Direct repair*: If a corrupted server is detected, the healthy servers will send their coded blocks directly to the new server without sending them back to the clients. The new server can verify the provided coded blocks and can compute the new coded blocks for itself without burdening the clients.
 - *Multi-client*: Multiple clients who own different secret keys can participate in the system. Their data are mixed together without losing the data confidentiality of individual clients.
 - *Symmetric key setting*: The scheme is constructed using symmetric key setting for the efficiency.
 - *Public authentication*: Not only the client but also any entity who has a given information can check the cloud servers while learning nothing about the secret key of each client. We employ a TPA on behalf of the clients to check the servers periodically.
- DD-POR: This scheme is an improvement of the MD-POR scheme to support dynamic operations unlike the MD-POR scheme. This means that the client not only can read the data but also can modify, insert, and delete the data. However, the DD-POR scheme is only a partial improvement of the MD-POR scheme because in this DD-POR scheme, we can only deal with a single client instead of multiple clients as the MD-POR scheme. Furthermore, the DD-POR does not deal with

the public authentication as the MD-POR scheme. Namely, the DD-POR has the following contributions:

- Direct repair: like the MD-POR scheme, when a server is corrupted, the healthy servers will provide their coded blocks and tags directly to the new server without sending them back to the client. Then, the new server can check them, and can compute the new coded blocks and the tags for itself.
 - Dynamic operations: unlike the MD-POR the client not only can check and retrieve the data, but also can modify, insert and delete the data.
 - Symmetric key setting: The scheme is constructed using symmetric key setting for the efficiency.
- ND-POR: This scheme has been proposed in which the network coding and the dispersal coding technique are combined in order to:
 - Reduce the costs of two important phases: the check phase and the repair phase.
 - Prevent replay attack, pollution attack and large corruption and specially, small corruption attack.

In future work, we focus on how to support direct repair (the MD-POR and DD-POR schemes). The repair phase has not been optimized because the healthy servers need to provide the aggregated coded blocks to the client, then the client computes the new coded blocks and stores them on the new server. A new mechanism can be considered in which the healthy servers send their coded blocks directly to the new server without sending back to the client. This mechanism can reduce the burden for the client and also reduce the communication. To support this mechanism, a signature scheme can be employed such as [84, 103] that allows the new server to verify the coded blocks provided from the healthy servers instead of the client, and to construct the new coded blocks by itself.

- SW-SSS: we firstly revisit the XOR network coding and apply it to the SSS. The revisited XOR network coding-based SSS has the following four advantages:
 - The shares are constructed using the XOR for fast computation.
 - The parameters (m, n) can be chosen arbitrarily.
 - The direct share repair is supported.
 - The size of a share is smaller than the size of the secret.

We then show that the SWC, which is used to compress a data stream in a network, can be also applied for SSS to reduce the share size of the revisited XOR network coding-based SSS. In other words, the SW-SSS improves the fourth advantage and still satisfies the first three advantages of the revisited XOR network coding-based SSS.

8.2 Future work

The research in this thesis has several future works:

1. Improve the DD-POR scheme to multiple clients and public authentication: As mentioned in the contribution chapter, the DD-POR scheme is an improvement of the MD-POR scheme such that the DD-POR scheme can support the dynamic operations (which means that the client can modify, insert and delete his/her data stored in the server) while the MD-POR cannot. However, the DD-POR scheme is only a partial improvement of the MD-POR scheme because the DD-POR scheme only supports a single client instead of multiple client like the MD-POR scheme, and because the DD-POR scheme does not meet the public authentication feature.
2. Implement the DD-POR scheme: this is because the efficiency of the DD-POR scheme is only analysed using the complexity theory without an implementation.
3. Implement the previous schemes which are used to compare with the MD-POR and SW-SSS schemes.

Bibliography

- [1] Amazon Simple Storage Service (Amazon S3). aws.amazon.com/s3.
- [2] SRB Storage Resource Broker. http://www.sdsc.edu/srb/index.php/Main_Page.
- [3] Kumar, Aswini, Whitchcock, Andrew, ed., “Google’s BigTable, First an overview. BigTable has been in development since early 2004 and has been in active use for about eight months” (2005), <http://googlecloudplatform.blogspot.com.au/2015/05/introducing-Google-Cloud-Bigtable.html>.
- [4] HP Public Cloud. <http://www.hpcloud.com/>
- [5] Dropbox. <https://www.dropbox.com/>
- [6] Google Drive. <https://www.google.com/drive/>
- [7] iCloud. <https://www.icloud.com/>
- [8] A. Juels and B. S. Kaliski, “PORs: Proofs of retrievability for large files”. In: *Proceedings of the 14th ACM conference on Computer and communications security - CCS’07*, pp. 584-597 (2007).
- [9] H. Shacham and B. Waters, “Compact Proofs of Retrievability”. In: *Proceedings of the 14th International Conference on the Theory and Application of Cryptology and Information Security: Advances in Cryptology - ASIACRYPT’08*, pp. 90-107 (2008).
- [10] K. Bowers, A. Juels and A. Oprea, “Proofs of retrievability: theory and implementation”. In: *Proceedings of the 2009 ACM workshop on Cloud computing security - CCSW’09*, pp. 43-54 (2009).
- [11] A. Juels, B. S. Kaliski, K. D. Bowers and A. M. Oprea, “Proof of retrievability for archived files”. In: *US Patent*, publication number: US8381062 B1 (2013).
- [12] J. Xu and E. C Chang, “Towards efficient proofs of retrievability”. In: *Proceedings of the 7th ACM Symposium on Information, Computer and Communications Security - ASIACCS’12*, pp. 79-80 (2012).

- [13] Y. Dodis, S. Vadhan, and D. Wichs. “Proofs of retrievability via hardness amplification”. In: *Proceedings of the 6th Theory of Cryptography Conference on Theory of Cryptography - TCC’09*, pp. 109-127 (2009).
- [14] W. J. Bolosky, J. R. Douceur, D. Ely and M. Theimer, “Feasibility of a serverless distributed file system deployed on an existing set of desktop PCs”. In: *Proceedings of ACM conference on Measurement and modeling of computation systems - SIGMETRICS’00*, pp. 34-43 (2000).
- [15] S. Ratnasamy, P. Francis, M. Handley, R. Karp and S. Shenker, “A scalable content-addressable network”. In: *Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications - SIGCOMM’01*, pp. 161-172 (2001).
- [16] P. Druschel and A. Rowstron, “Storage management and caching in PAST, a largescale, persistent peer-to-peer storage utility”. In: *Proceedings of the 18th ACM symposium on Operating systems principles - SOSP’01*, pp. 188-201 (2001).
- [17] Z. Zhang, Q. Lian, S. Lin, W.Chen, Y.Chen and C. Jin, “Bitvault: A highly reliable distributed data retention platform”. In: *ACM SIGOPS Operating Systems Review - Systems work at Microsoft Research*, vol. 41, no. 2, pp. 27-36 (2007).
- [18] R. Curtmola, O. Khan, R. Burns and G. Ateniese, “MR-PDP: Multiple-Replica Provable Data Possession”. In: *Proceedings of the 28th International Conference on Distributed Computing Systems - ICDCS’08*, pp. 411-420 (2008).
- [19] V. Pless, “Introduction to the Theory of Error-Correcting Codes”. In: *Wiley-Interscience Series in Discrete Mathematics*, John Wiley and Sons, ISBN 0-471-08684-3 (1982).
- [20] M. K. Aguilera, R. Janakiraman and L. Xu, “Efficient fault-tolerant distributed storage using erasure codes”, *Technical Report*, Washington University in St. Louis (2004), Available at: www.nisl.wustl.edu/Papers/Tech/aguilera04efficient.pdf
- [21] J. Kubiatowicz, D. Bindel, Y. Chen, S. Czerwinski, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, W. Weimer, C. Wells and B. Zhao, “Oceanstore: An architecture for global-scale persistent storage”. In: *ACM SIGPLAN Notices*, vol. 35, no. 11, pp. 190-201 (2000).
- [22] F. W. Chang, M. Ji, S. A. Leung, J. MacCormick, S. E. Perl and L. Zhang, “Myriad: Cost-effective Disaster Tolerance”. In: *Proceedings of the Conference on File and Storage Technologies - FAST’02*, pp. 103-116 (2002).
- [23] S. Frolund, A. Merchant, Y. Saito, S. Spence and A. Veitch, “A decentralized algorithm for erasure-coded virtual disks”. In: *Proceedings of the International Conference on Dependable Systems and Networks - DSN’04*, pp. 125-134 (2004).

- [24] M. K. Aguilera, R. Janakiraman and L. Xu, "Using erasure codes efficiently for storage in a distributed system". In: *Proceedings of the International Conference on Dependable Systems and Networks - DSN'05*, pp. 336-345 (2005).
- [25] J. Hendricks, G. R. Ganger and M. Reiter, "Verifying distributed erasure-coded data". In: *Proceedings of the 26th ACM symposium on Principles of distributed computing - PODC'07*, pp. 139-146 (2007).
- [26] K. D. Bowers, A. Juels and A. Oprea, "HAIL: A High-Availability and Integrity Layer for Cloud Storage". In: *Proceedings of the 16th ACM conference on Computer and communications security - CCS'09*, pp. 187-198 (2009).
- [27] H. Y. Lin and W. G. Tzeng, "A Secure Erasure Code-Based Cloud Storage System with Secure Data Forwarding". In: *IEEE Transactions on Parallel and Distributed Systems*, vol. 23, no. 6, pp. 995-1003 (2012).
- [28] O. Goldreich, "Towards a Theory of Software Protection and Simulation by Oblivious RAMs". In: *Proceedings of the nineteenth annual ACM symposium on Theory of computing - STOC'87*, pp. 182-194 (1987).
- [29] R. Ostrovsky, "Efficient computation on oblivious rams". In: *Proceedings of the nineteenth annual ACM symposium on Theory of computing - STOC'90*, pp. 514-523 (1990).
- [30] O. Goldreich and R. Ostrovsky, "Software protection and simulation on oblivious RAMs". In: *Journal of the ACM*, vol. 43, no. 3, pp. 431-473 (1996).
- [31] M. T. Goodrich, M. Mitzenmacher, O. Ohrimenko and R. Tamassia, "Oblivious RAM simulation with efficient worst-case access overhead". In: *Proceedings of the 3rd ACM Cloud Computing Security Workshop - CCSW'11*, pp. 95-100 (2011).
- [32] E. Shi, T. H. H. Chan, E. Stefanov and M. Li, "Oblivious ram with $o((\log n)^3)$ worst-case cost". In: *Proceedings of the 17th International Conference on the Theory and Application of Cryptology and Information Security - ASIACRYPT'11*, pp. 197-214 (2011).
- [33] E. Stefanov, M. Dijk, E. Shi, C. Fletcher, L. Ren, X. Yu and S. Devadas, "Path ORAM: an extremely simple oblivious RAM protocol". In: *Proceedings of the ACM SIGSAC conference on Computer and communications security - CCS'13*, pp. 299-310 (2013).
- [34] E. Shi, E. Stefanov and C. Papamanthou, "Practical dynamic proofs of retrievability". In: *Proceedings of the 2013 ACM SIGSAC conference on Computer and communications security - CCS'13*, pp. 325-336 (2013).
- [35] E. Stefanov and E. Shi, "Oblivistore: High performance oblivious cloud storage". In: *IEEE Symposium on Security and Privacy - SP'13*, pp. 253-267 (2013).

- [36] D. Cash, A. Kupcu and D. Wichs, “Dynamic Proofs of Retrievability via Oblivious RAM”. In: *Proceedings of the 32nd Annual International Conference on the Theory and Applications of Cryptographic Techniques - EUROCRYPT’13*, vol. 7881, pp. 279-295 (2013).
- [37] D. Apon, J. Katz, E. Shi and A. Thiruvengadam, “Verifiable Oblivious Storage”. In: *Proceedings of the 17th International Conference on Practice and Theory of Public-Key Cryptography - PKC’14*, pp. 131-148 (2014).
- [38] R. W. Yeung, “Multilevel diversity coding with distortion”. In: *IEEE Transactions on Information Theory*, vol. 41, no. 2, pp. 412-422 (1995).
- [39] J. R. Roche, R. W. Yeung and K. P. Hau, “Symmetrical multilevel diversity coding”. In: *IEEE Transactions on Information Theory*, vol. 43, no. 3, pp. 1059-1064 (1997).
- [40] R. W. Yeung and Z. Zhang, “Distributed source coding for satellite communications”. In: *IEEE Transactions on Information Theory*, vol. 45, no. 4, pp. 1111-1120 (1999).
- [41] R. W. Yeung and Z. Zhang, “On symmetrical multilevel diversity coding”. In: *IEEE Transactions on Information Theory*, vol. 45, no. 2, pp. 609-621 (1999).
- [42] R. Ahlswede, N. Cai, S. Y. R. Li and R.W. Yeung, “Network information flow”. In: *IEEE Transactions on Information Theory*, vol. 46, no. 4, pp. 1204-1216 (2000).
- [43] S. Y. R. Li, R. W. Yeung and N. Cai, “Linear Network Coding”. In: *IEEE Transactions on Information Theory*, vol. 49, no. 2, pp. 371-381 (2003).
- [44] R. Koetter and M. Medard, “An Algebraic Approach to Network Coding”. In: *IEEE/ACM Transactions on Networking (TON)*, vol. 11, no. 5, pp. 782-795 (2003).
- [45] T. Ho, R. Koetter, M. Medard, D. R. Karger and M. Effros, “The benefits of coding over routing in a randomized setting”. In: *Proceedings of the IEEE International Symposium on Information Theory - ISIT’03*, p. 442 (2003).
- [46] P. A. Chou, Y. Wu, K. Jain, “Practical network coding”. In: *Proceedings of the 41st Annual Allerton Conference on Communication, Control, and Computing* (2003).
- [47] T. Ho, M. Medard, J. Shi, M. Effros and D. R. Karger, “On randomized network coding”. In: *Proceedings of the 41st Annual Allerton Conference on Communication, Control, and Computing* (2003).
- [48] T. Ho, M. Medard, R. Koetter, D. R. Karger, M. Effros, J. Shi and B. Leong, “A random linear network coding approach to multicast”. In: *IEEE Transactions on Information Theory*, vol. 52, no. 10, pp. 4413-4430 (2006).
- [49] J. Li, S. Yang, X. Wang, X. Xue and B. Li, “Tree-structured data regeneration with network coding in distributed storage systems”. In: *Proceedings of the 17th International Workshop on Quality of Service - IWQoS’09*, pp. 1-9 (2009).

- [50] J. Li, S. Yang, X. Wang, X. Xue and B. Li, "Tree-structured Data Regeneration in Distributed Storage Systems with Network Coding". In: *Proceedings of the 29th conference on Information communications - INFOCOM'10*, pp. 2892-2900 (2010).
- [51] A. Dimakis, P. Godfrey, Y. Wu, M. Wainwright and K. Ramchandran, "Network coding for distributed storage systems". In: *IEEE Transactions on Information Theory*, vol. 56, no. 9, pp. 4539-4551 (2010).
- [52] S. Agrawal and D. Boneh, "Homomorphic MACs: MACBased Integrity for Network Coding". In: *Proceedings of the 7th Conference on Applied Cryptography and Network Security - ACNS'09*, pp. 292-305 (2009).
- [53] C. Cheng and T. Jiang, "An Efficient Homomorphic MAC with Small Key Size for Authentication in Network Coding". In: *IEEE Transactions on Computers*, vol. 62, no. 10, pp. 2096-2100 (2012).
- [54] C. Cheng, T. Jiang and Q. Zhang, "TESLA-Based Homomorphic MAC for Authentication in P2P System for Live Streaming with Network Coding". In: *IEEE Journal on Selected Areas in Communications*, vol. 31, no. 9, pp. 291-298 (2013).
- [55] Jonathan Katz and Brent Waters, "Compact signatures for network coding". In: *Cryptology ePrint Archive*, vol. 316 (2008), Available at: <http://eprint.iacr.org/2008/316>.
- [56] R. Johnson, D. Molnar, D. Song and D. Wagner, "Homomorphic Signature Schemes". In: *Proceedings of the Cryptographer's Track at the RSA Conference on Topics in Cryptology - CT-RSA'02*, pp. 244-262 (2002).
- [57] N. Attrapadung and B. Libert, "Homomorphic network coding signatures in the standard model". In: *Proceedings of the 14th conference on Practice and theory in public key cryptography - PKC'11*, pp. 680-696 (2011).
- [58] D. M. Freeman, "Improved security for linearly homomorphic signatures: a generic framework". In: *Proceedings of the 15th conference on Practice and Theory in Public Key Cryptography - PKC'12*, vol. 7293, pp. 697-714 (2012).
- [59] D. Catalano, D. Fiore and B. Warinschi, "Efficient network coding signatures in the standard model". In: *Proceedings of the 15th conference on Practice and theory in public key cryptography - PKC'12*, pp. 680-696 (2012).
- [60] S. Acedanski, S. Deb, M. Medard, and R. Koetter, "How good is random linear coding based distributed networked storage?". In: *Proceedings of the 1st Workshop on Network Coding, Theory and Applications - NETCOD'05* (2005).
- [61] B. Chen, R. Curtmola, G. Ateniese and R. Burns, "Remote Data Checking for Network Coding-based Distributed Storage Systems". In: *Proceedings of the ACM Cloud Computing Security Workshop - CCSW'10*, pp. 31-42 (2010).

- [62] A. Le and A. Markopoulou, “NC-Audit: Auditing for network coding storage”. In: *International Symposium on Network Coding - NetCod’12*, pp. 155-160 (2012).
- [63] N. Cao, S. Yu, Z. Yang, W. Lou and Y. T. Hou, “LT Codes-based Secure and Reliable Cloud Storage Service”. In: *Proceedings of the 31st IEEE conference on Computer Communications - INFOCOM’12*, pp. 693-701 (2012).
- [64] Y. Hu, H. C. H. Chen, P. P. C. Lee and Y. Tang, “NCCloud: Applying Network Coding for the Storage Repair in a Cloud-of-Clouds”. In: *Proceedings of the 10th USENIX Conference on File and Storage Technologies - FAST’12* (2012).
- [65] H. C. H. Chen, Y. Hu, P. P. C. Lee and Y. Tang, “NCCloud: A Network-Coding-Based Storage System in a Cloud-of Clouds”. In: *IEEE Transactions on Computers*, vol. 63, no. 1, pp. 31-44 (2014).
- [66] F. Chen, T. Xiang, Y. Yang and S. S. M. Chow, “Secure Cloud Storage Meets with Secure Network Coding”. In: *Proceedings of the 33rd Annual IEEE International Conference on Computer Communications - INFOCOM’14*, pp. 673-681 (2014).
- [67] B. Chen B and R. Curtmola, “Robust dynamic remote data checking for public clouds”. In: *Proceedings of the ACM conference on Computer and communications security - CCS’12*, pp. 1043-1045 (2012).
- [68] C. Wang, Q. Wang, K. Ren and W. Lou, “Privacy-preserving public auditing for data storage security in cloud computing”. In: *Proceedings of the 29th conference on Information communications - INFOCOM’10*, pp. 525-533 (2010).
- [69] Q. Wang, C. Wang, J. Li, K. Ren and W. Lou, “Enabling Public Verifiability and Data Dynamics for Storage Security in Cloud Computing”. In: *Proceedings of the 14th European Symposium on Research in Computer Security - ESORICS’09*, pp. 355-370 (2009).
- [70] Q. Wang, C. Wang, K. Ren, W. Lou and J. Li, “Enabling Public Auditability and Data Dynamics for Storage Security in Cloud Computing”. In: *IEEE Transactions on Parallel and Distributed Systems*, vol. 22, no. 5, pp. 847-859 (2011).
- [71] C. Wang, Q. Wang, K. Ren, N. Cao and W. Lou, “Toward Secure and Dependable Storage Services in Cloud Computing”. In: *IEEE Transactions on Services Computing*, vol. 5, no. 2, pp. 220-232 (2012).
- [72] Q. Zheng and S. Xu, “Fair and dynamic proofs of retrievability”. In: *Proceedings of the first ACM conference on Data and application security and privacy - CO-DASPY’11*, pp. 237-248 (2011).
- [73] Z. Mo, Y. Zhou and S. Chen, “A dynamic Proof of Retrievability (PoR) scheme with $O(\log n)$ complexity”. In: *Proceedings of the 2012 IEEE International Conference on Communications ICC’12*, pp. 912-916 (2012).

- [74] Y. Zhu, G. Ahn, H. Hu, S. S. Yau, H. G. An, C. Hu, “Dynamic Audit Services for Outsourced Storages in Clouds”. In: *IEEE Transactions on Services Computing*, vol. 6, no. 2, pp. 227-238 (2011).
- [75] Y. Yu, Y. Mu, J. Ni, J. Deng and K. Huang, “Identity Privacy-Preserving Public Auditing with Dynamic Group for Secure Mobile Cloud Storage”. In: *Proceedings of the 8th International Conference on Network and System Security - NSS’14*, pp. 28-40 (2014).
- [76] A. Herzberg, M. Jakobsson, H. Krawczyk, and M. Yung, “Proactive public key and signature systems”. In: *Proceedings of the 4th ACM conference on Computer and communications security - CCS’97*, pp. 100110 (1997).
- [77] A. Herzberg, S. Jarecki, H. Krawczyk, and M. Yung, “Proactive secret sharing, or: How to cope with perpetual leakage”. In: *Proceedings of the 15th Annual International Cryptology Conference - CRYPTO’95*, vol. 1963 of LNCS, pp. 339352 (1995).
- [78] A. Juels and A. Oprea, “New Approaches to Security and Availability for Cloud Data”. In: *Communications of the ACM*, vol. 56, no. 2, pp. 64-73, DOI: 10.1145/2408776.2408793 (2013).
- [79] A. Le and A. Markopoulou, “On detecting pollution attacks in inter-session network coding”. In: *Proceedings of the 31st IEEE conference on Computer Communications - INFOCOM’12*, pp. 343-351 (2012).
- [80] B. Chen and R. Curtmola, “Towards Self-Repairing Replication-Based Storage Systems Using Untrusted Clouds”. In: *Proceedings of the third ACM conference on Data and application security and privacy - CODASPY’13*, pp. 377-388 (2013).
- [81] B. Chen, A. K. Ammala and R. Curtmola, “Towards Server-side Repair for Erasure Coding-based Distributed Storage Systems”. In: *Proceedings of the 5th ACM Conference on Data and Application Security and Privacy - CODASPY’15*, pp. 281-288 (2015).
- [82] S. Agrawal, D. Boneh, X. Boyen and D. M. Freeman, “Preventing pollution attacks in multi-source network coding”. In: *Proceedings of the 13th International Conference on Practice and Theory in Public Key Cryptography - PKC’10* (2010).
- [83] L. Czap and I. Vajda, “Signatures for multi-source network coding”. In: *IACR Cryptology ePrint Archive* (2010). Available at: <http://eprint.iacr.org/2010/328>
- [84] W. Yana, M. Yang, L. Li and H. Fang, “Short signature scheme for multi-source network coding”. In: *Journal on Computer Communications*, vol. 35, issue. 3, pp. 344-351, Elsevier Science Publishers (2012).

- [85] J. Zhang, J. Shao, Y. Ling, M. Ji, G. Wei1 and B. Ying, “Efficient multiple sources network coding signature in the standard model”. In: *Concurrency and Computation: Practice and Experience*, Wiley publisher, DOI: 10.1002/cpe.3322 (2014).
- [86] A. Le and A. Markopoulou, “Cooperative Defense Against Pollution Attacks in Network Coding Using SpaceMac”. In: *IEEE Journal Selected Areas in Communications*, vol. 30, no. 2, pp. 442-449 (2012).
- [87] F. Oggier and H. Fathi, “An Authentication Code Against Pollution Attacks in Network Coding”. In: *IEEE/ACM Transactions on Networking*, vol. 19, no. 6, pp. 1587-1596, DOI: 10.1109/TNET.2011.2126592 (2011).
- [88] X. Wu, Y. Xu, C. Yuen and L. Xiang, “A Tag Encoding Scheme against Pollution Attack to Linear Network Coding”. In: *IEEE Transactions on Parallel and Distributed Systems*, vol. 25, no. 1, pp. 33-42, doi:10.1109/TPDS.2013.24 (2014).
- [89] Z. Yu, Y. Wei, B. Ramkumar and Y. Guan, “An Efficient Signature-Based Scheme for Securing Network Coding Against Pollution Attacks”. In: *Proceedings of the IEEE 27th Conference on Computer Communication - INFOCOM’08*, doi: 10.1109/INFOCOM.2008.199 (2008).
- [90] X. Wu, Y. Xu, L. Xiang and W. Xu, “A Hybrid Scheme against Pollution Attack to Network Coding”. In: *Proceedings of the 2011 International Symposium on Network Coding - NetCod’11*, pp. 1-4, doi: 10.1109/ISNETCOD.2011.5979070 (2011).
- [91] J. Dong, R. Curtmola, and C. Nita-Rotaru, “Practical Defenses Against Pollution Attacks in Wireless Network Coding”. In: *ACM Transactions on Information and System Security*, vol. 14, no. 1, article 7 (2011).
- [92] A. Le and A. Markopoulou, “TESLA-Based Defense against Pollution Attacks in P2P Systems with Network Coding”. In: *Proceedings on IEEE International Symposium on Network Coding - NetCod’11*, pp. 1-7 (2011).
- [93] L. Carter and M. Wegman, “Universal Hash Functions”. In: *Journal of Computer and System Sciences*, vol. 18, no. 2, pp. 143-154 (1979).
- [94] J. Baylis, “Error-Correcting Codes: A Mathematical Introduction”. Boca Raton, FL: CRC Press (1998).
- [95] J. H. Conway and N. J. A. Sloane, “Error-Correcting Codes”. Chapter 3.2 in *Sphere Packings, Lattices, and Groups*, 2nd edition New York: Springer-Verlag, pp. 75-88 (1993).
- [96] F. J. MacWilliams and N. J. A. Sloane, “The Theory of Error-Correcting Codes”. Amsterdam, Netherlands: North-Holland (1977).
- [97] M. Riley and I. Richardson, “An introduction to Reed Solomon codes: principles, architecture and implementation”. Prentice-hall (2001).

- [98] L. R. Welch, “The Original View of ReedSolomon Codes”. PDF, Lecture Notes (1997).
- [99] V. Shoup, “On fast and provably secure message authentication based on universal hashing”. In: *Proceedings of the 16th Cryptology Conference on Advances in Cryptology - CRYPTO’96*, pp. 313-328 (1996).
- [100] D. R. Stinson, “Cryptography - Theory and Practice”. CRC Press, Boca Raton (1995).
- [101] O. Goldreich, S. Goldwasser and S. Micali, “How to Construct Random Functions”. In: *Journal of the ACM*, vol. 33, no. 4, pp. 792-807 (1986).
- [102] G. Ateniese, R. Burns, R. Curtmola, J. Herring, L. Kissner, Z. Peterson, and D. Song, “Provable data possession at untrusted stores”. In: *Proceedings of the 14th ACM Conference on Computing and communication security - CCS’07*, pp. 598-609 (2007).
- [103] D. Catalano, D. Fiore and B. Warinschi, “Efficient network coding signature in the standard model”. In: *Proceedings of the 15th Conference on Practice and Theory in Public Key Cryptography - PKC’12*, pp. 680-696 (2012).
- [104] H. Handschuh and B. Preneel, “Key-Recovery Attacks on Universal Hash Function Based MAC Algorithms”. In: *Proceedings of the 28th Conference on Cryptology: Advances in Cryptology - CRYPTO’08*, pp. 144-161 (2008).
- [105] R. Curtmola, O. Khan, and R. Burns, “Robust remote data checking”. In: *Proceedings of the 4th ACM international workshop on Storage security and survivability - StorageSS’08*, pp. 63-68 (2008).
- [106] B. Chen, and R. Curtmola, “Auditable Version Control Systems”. In: *Proceedings of the 21th Annual Network and Distributed System Security Symposium - NDSS’14* (2014).
- [107] F. Didier, “Efficient erasure decoding of Reed-Solomon codes”. arXiv:0901.1886v1 [cs.IT] (2009).
- [108] G. Ateniese, R. Di Pietro, L. Mancini, and G. Tsudik, “Scalable and efficient provable data possession”. In: *Proceedings of the 4th Conference on Security and privacy in communication networks - SecureComm’08*, Article no. 9 (2008).
- [109] G. Ateniese, R. Burns, R. Curtmola, J. Herring, O. Khan, L. Kissner, Z. Peterson, and D. Song, “Remote data checking using provable data possession”. In: *Journal ACM Transactions on Information and System Security (TISSEC)*, vol 14, no. 1, May 2011, Article No. 12 (2011).
- [110] A. Shamir, “How to share a secret”. In: *Communication of the ACM*, vol. 22, no. 11, pp. 612-613 (1979).

- [111] G. R. Blakley, "Safeguarding cryptographic keys". In: *Proceedings of the AFIPS National Computer Conference*, vol.48, pp.313-317 (1979).
- [112] E. Karnin, J. Greene and M. Hellman, "On secret sharing systems". In: *IEEE Transactions on Information Theory*, vol. 29, no. 1, pp. 35-41 (1983).
- [113] R. Capocelli, A. Santis, L. Gargano and U. Vaccaro, "On the size of shares for secret sharing schemes". In: *Journal of Cryptology*, vol. 6, no. 3, pp. 157-167 (1993).
- [114] G. R. Blakley and C. Meadows C, "Security of ramp schemes". In: *Proceedings of the CRYPTO on Advances in cryptology*, LNCS 196, Springer-Verlag, pp. 242-269 (1984).
- [115] H. Yamamoto, "On secret sharing systems using (k, L, n) threshold scheme". In: *IE-ICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences*, vol. J68-A,no. 9, pp. 945-952 (1985).
- [116] K. Kurosawa, K. Okada, K. Sakano, W. Ogata and T. Tsujii, "Non perfect secret sharing schemes and matroids". In: *Workshop on the Theory and Application of Cryptographic Techniques - EUROCRYPT'93*, LNCS 765, Springer-Verlag, pp.126-141 (1993).
- [117] W. Ogata and K. Kurosawa, "Some basic properties of general nonperfect secret sharing schemes". In: *Journal of Universal Computer Science*, vol. 4, no. 8, pp. 690-704 (1998).
- [118] K. Okada and K. Kurosawa, "Lower bound on the size of shares of nonperfect secret sharing schemes". In: *Proceedings of the 4th International Conference on the Theory and Applications of Cryptology - ASIACRYPT'94*, LNCS 917, SpringerVerlag, pp. 34-41 (1994).
- [119] Y. Wang, "Efficient LDPC Code Based Secret Sharing Schemes and Private Data Storage in Cloud without Encryption". *Technical report*, UNC Charlotte (2012).
- [120] H. Ishizu and T. Ogihara, "A study on long-term storage of electronic data". In: *IEICE General Conference*, vol. D-9-10, no. 1, pp. 125 (2004).
- [121] Y. Fujii, M. Tada, N. Hosaka, K. Tochikubo and T. Kato, "A fast $(2, n)$ -threshold scheme and its application". In: *Proceedings of the CSS conference (in Japanese)*, pp 631-636 (2005).
- [122] N. Hosaka, K. Tochikubo, Y. Fujii, M. Tada and T. Kato, " $(2, n)$ -threshold secret sharing systems based on binary matrices". In: *Symposium on SCIS (in Japanese)*, pp. 2D 1-4 (2007).
- [123] Y. Suga, "New constructions of $(2, n)$ -threshold secret sharing schemes using exclusive-OR operations". In: *Proceedings of the 7th Conference on Innovative Mobile and Internet Services in Ubiquitous Computing (IMIS'13)*, pp. 837-842 (2013).

- [124] J. Kurihara, S. Kiyomoto, K. Fukushima and T. Tanaka, “A fast $(3,n)$ -threshold secret sharing scheme using exclusive-OR operations”. In: *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences*, vol. E91-A, no. 1, pp. 127-138 (2008).
- [125] N. Shiina, T. Okamoto and E. Okamoto, “How to convert 1-out-of- n proof into k -out-of- n proof”. In: *Symposium on SCIS (in Japanese)*, pp 1435-1440 (2004).
- [126] H. Kunii and M. Tada, “A note on information rate for fast threshold schemes”. In: *Proceedings of CSS’06*, pp. 101106 (2006).
- [127] J. Kurihara, S. Kiyomoto, K. Fukushima and T. Tanaka, “A new (k,n) -threshold secret sharing scheme and its extension”. In: *Proceedings of the 11th conference on Information Security - ISC’08*, pp.455-470 (2008).
- [128] L. Chunli, X. Jia, L. Tian L, J. Jing, M. Sun, “Efficient Ideal Threshold Secret Sharing Schemes Based on EXCLUSIVE-OR Operations”. In: *Proceedings of the 4th Conference on Network and System Security - NSS’10*, pp.136-143 (2010).
- [129] Y. Wang and Y. Desmedt, “Efficient Secret Sharing Schemes Achieving Optimal Information Rate”. In: *Proceedings of the IEEE Information Theory Workshop - ITW’14*, pp. 516-520 (2014).
- [130] J. Kurihara, S. Kiyomoto, K. Fukushima and T. Tanaka, “A fast $(k-L-N)$ -Threshold Ramp secret sharing scheme”. In: *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences*, doi:10.1587/transfun.E92.A.1808 (2009).
- [131] K. V. Rashmi, N. B. Shah, and P. V. Kumar, “Optimal Exact-Regenerating Codes for Distributed Storage at the MSR and MBR Points via a ProductMatrix Construction”. In: *IEEE Transactions on Information Theory*, vol. 57, no. 8, pp. 5227-5239 (2011).
- [132] M. Kurihara and H. Kuwakado, “Secret Sharing Schemes Based on Minimum Bandwidth Regenerating Codes”. In: *Proceedings of the International Symposium on Information Theory and its Applications - ISITA’12*, pp.255-259 (2012).
- [133] Z. Tang, H. W. Lim and H. Wang, “Revisiting a Secret Sharing Approach to Network Codes”. In: *Proceedings of the 6th international conference on Provable Security - ProvSec’12*, pp. 300-317. (2012).
- [134] N. B. Shah, K. V. Rashmi and K. Ramchandran, “Secure network coding for distributed secret sharing with low communication cost”. In: *Proceedings of the 2013 IEEE International Symposium on Information Theory Proceedings - ISIT’13*, pp. 2404-2408 (2013).
- [135] N. Cai and W. Y. Raymond, “Secure network coding”. In: *Proceedings of the IEEE International Symposium on Information Theory - ISIT’02*, pp. 323-329 (2002).

- [136] S. Katti, H. Rahul, W. Hu, D. Katabi, M. Medard and J. Crowcroft, “XORs in the air: practical wireless network coding”. In: *IEEE/ACM Transactions on Networking*, vol. 16, no. 3, pp. 497-510 (2008).
- [137] Z. Yu, Y. Wei, B. Ramkumar and Y. Guan, “An Efficient Scheme for Securing XOR Network Coding against Pollution Attacks”. In: *Proceedings of the 28th Conference on Computer Communication - INFOCOM’09*, pp.406-414 (2009).
- [138] A. Khreishah, I. M. Khalil, P. Ostovari and J. Wu, “Flow-based XOR Network Coding for Lossy Wireless Networks”. In: *IEEE Transactions on Wireless Communications*, vol. 11, no. 6, pp. 2321-2329 (2012).
- [139] K. Izawa, A. Miyaji and K. Omote, “Lightweight Integrity for XOR Network Coding in Wireless Sensor Networks”. In: *Proceedings of the 8th international conference on Information Security Practice and Experience - ISPEC’12*, pp. 245-258 (2012).
- [140] D. Slepian and J. K. Wolf, “Noiseless coding of correlated information sources”. In: *IEEE Transactions on Information Theory*, vol. 19, no. 4, pp. 471-480 (1973).
- [141] A. Wyner, “Recent results in the Shannon theory”. In: *IEEE Transactions on Information Theory*, vol. 20, Jan. 1974, pp. 210 (1974).
- [142] D. A. Wyner and J. Ziv, “The rate-distortion function for source coding with side information at the decoder”. In: *IEEE Transactions on Information Theory*, vol. 22, no. 1, pp. 1-10 (1976).
- [143] M. C. Thomas (1975), “A proof of the data compression theorem of Slepian and Wolf for ergodic sources”. In: *IEEE Transactions on Information Theory*, vol. 21, no. 2, pp. 226-228 (1975).
- [144] R. Cristescu, B. B. Lozano and M. Vetterli, “Networked Slepian-Wolf: theory, algorithms, and scaling laws”. In: *IEEE Transactions on Information Theory*, vol. 51, no. 12, pp. 4057-4073 (2005).
- [145] E. Tuncel, “Slepian-Wolf coding over broadcast channels”. In: *IEEE Transactions on Information Theory*, vol. 52, no. 4, pp. 1469-1482 (2006).
- [146] V. Stankovi, A. D. Liveris, Z. Xiong Z and C. N. Georghiades, “Design of Slepian-Wolf Codes by Channel Code Partitioning”. In: *Proceedings of the conference on Data Compression - DCC’04*, pp. 302-311 (2004).
- [147] V. Stankovic, A. D. Liveris, Z. Xiong Z and C. N. Georghiades, “On code design for the Slepian-Wolf problem and lossless multiterminal networks”. In: *IEEE Transactions on Information Theory*, vol. 52, no. 4, pp. 1495-1507 (2006).
- [148] J. Chen, H. Dake and A. Jagmohan, “Slepian-Wolf Code Design via Source-Channel Correspondence”. In: *Proceedings of the 2006 IEEE International Symposium on Information Theory - ISIT’06*, pp. 2433-2437 (2006).

- [149] S. Cheng, “Slepian-Wolf Code Designs”, (2010), Available at:
http://tulsagrad.ou.edu/samuel_cheng/information_theory_2010/swcd.pdf.
- [150] S. Li and A. Ramamoorthy, “Algebraic codes for Slepian-Wolf code design”. In: *Proceedings of the 2011 IEEE International Symposium on Information Theory - ISIT'11*, pp. 1861-1865 (2011).
- [151] N. Gehrig and P. L. Dragotti, “Symmetric and A-Symmetric Slepian-Wolf Codes with Systematic and Non-Systematic Linear Codes”. In: *IEEE Communications Letters*, vol. 9, no. 1, pp. 61-63 (2005).
- [152] S. S. Pradhan and K. Ramchandran, “Distributed source coding using syndromes (discuss): design and construction”. In: *Proceedings of the Data Compression Conference - DCC'99*, pp. 158-167 (1999).

Publications

JOURNAL

- [1] Kazumasa Omote and Thao Phuong Tran, “ND-POR: A POR based on Network Coding and Dispersal Coding”. In: *IEICE Transactions on Information and Systems*, vol. E98-D, no. 8, pp.-, Aug 2015 (to appear) (*authors in alphabetical order*) [*ISI-indexed*, *SJR-indexed*].
- [2] Kazumasa Omote and Thao Phuong Tran, “MD-POR: Multi-source and Direct Repair for Network Coding-based Proof of Retrievability”. In: *International Journal of Distributed Sensor Networks (IJDSN) on Advanced Big Data Management and Analytics for Ubiquitous Sensors*, vol. 2015, article ID. 586720, Jan 2015 (*authors in alphabetical order*) [*ISI-indexed*, *SJR-indexed*].

INTERNATIONAL CONFERENCE

- [1] Kazumasa Omote and Thao Phuong Tran, “DD-POR: Dynamic Operations and Direct Repair in Network Coding-based Proof of Retrievability”. In: *Proceedings of the 21th Annual International Computing and Combinatorics Conference - COCOON’15*, Aug 2015, Springer-Verlag (*authors in alphabetical order*) [*CORE rank A*].
- [2] Kazumasa Omote and Thao Phuong Tran, “SW-SSS: Slepian-Wolf Coding-based Secret Sharing Scheme”, In: *Proceedings of the 8th conference on Computational Intelligence in Security for Information Systems - CISIS’15*, June 2015, Spain, Springer-Verlag (*authors in alphabetical order*) [*CORE rank B*].
- [3] Kazumasa Omote and Thao Phuong Tran, “A New Efficient and Secure POR Scheme based on Network Coding”. In: *Proceedings of the 28th IEEE International Conference on Advanced Information Networking and Applications - AINA’14*, May 2014, Victoria, Canada (*authors in alphabetical order*) [*CORE rank B*].
- [4] Kazumasa Omote and Thao Phuong Tran, “POR-2P: Network Coding-based POR for Data Provision-Payment System”. In: *Proceedings of the 10th International Conference on Risks and Security of Internet and Systems (CRISIS’10)*, Greece, July 2015 (*authors in alphabetical order*) [*CORE rank C*].

- [5] Thao Phuong Tran, Lee Chin Kho and Azman Osman Lim, “SW-POR: A Novel POR Scheme using Slepian-Wolf Coding for Cloud Storage”. In: *Proceedings of the 11th IEEE International Conference on Autonomic and Trusted Computing - ATC’14*, December 2014, Indonesia [**CORE rank C**].
- [6] Kazumasa Omote and Thao Phuong Tran, “MDNC: Multi-source and Direct Repair in Network Coding-based Proof of Retrievability Scheme”. In: *Proceedings of the 15th International Workshop on Information Security Applications - WISA’14*, Aug 2014, Jeju, Korea, Springer-Verlag (*authors in alphabetical order*) [**CORE rank C**].
- [7] Thao Phuong Tran, Kazumasa Omote, Nguyen Gia Luyen and Nguyen Dinh Thuc, “Improvement of multi-user searchable encrypted data scheme”. In: *Proceedings of the 7th IEEE International Conference for Internet Technology and Secured Transactions - ICITST’12*, London, England, December 2012, pp. 396-401.
- [8] Thao Phuong Tran, Luyen G. Nguyen, Thuc D. Nguyen and Hue T. B. Pham, “A New Multi Multi-user Searchable Data Encryption Scheme”. In: *Proceedings of the 9th IEEE International Conference on Computing and Communication Technologies - RIVF’12*, Ho Chi Minh, Vietnam, February 2012.

DOMESTIC CONFERENCE

- [1] Kazumasa Omote and Thao Phuong Tran, “S-POR: An Extremely Simple Network Coding-based Proof of Retrievability”, In: Computer Security - CSEC, IPSJ SIG vol. 2015-CSEC68-No55, Tokyo, Japan, Mar 2015 (*authors in alphabetical order*).
- [2] Kazumasa Omote and Thao Phuong Tran, “A practical and efficient network-coding-based POR scheme”, In: Computer Security Symposium - CSS, Hokkaido, Japan, October 2014 (*authors in alphabetical order*).
- [3] Kazumasa Omote and Thao Phuong Tran, “Improvement of Network coding-based System for Ensuring Data Integrity in Cloud Computing”, In: IPSJ SIG Technical Report, vol. 2012-CSEC-58 No.21, Hokkaido, Japan (2012) (*authors in alphabetical order*).

Appendix A

Appendix

A.1 The Algorithms of the SW-SSS Scheme

A.1.1 Share Generation

Algorithm 1 ShareGen

Input: m, n, S

Output: $\{c_0, d_0\}, \dots, \{c_{n-1}, d_{n-1}\}$

```
1:  $S = b_0 || \dots || b_{m-1}$ 
2:  $|b| = |S|/m$ 
3:  $count \leftarrow 0$ 
4: for  $j \leftarrow 0$  to  $m - 3$  do
5:   for  $t \leftarrow j + 1$  to  $m - 2$  do
6:     for  $z \leftarrow t + 1$  to  $m - 1$  do
7:        $s_i \leftarrow b_j \oplus b_t \oplus b_z$ 
8:        $d_i \leftarrow s_i.count('1')$ 
9:        $c_i \leftarrow \text{FindShare}(|b|, s_i, d_i)$ 
10:       $count++$ 
11:      if  $(count == n - 1)$  then
12:        return  $\{c_0, d_0\}, \dots, \{c_{n-1}, d_{n-1}\}$ 
13:      end if
14:    end for
15:  end for
16: end for
```

Algorithm 2 FindShare

Input: $|b|, s_i, d_i$ **Output:** c_i

- 1: $M_i \leftarrow \text{list_permutation}(|b|, d_i)$
 - 2: $c_i \leftarrow \text{index}(M_i, s_i)$
 - 3: **return** c_i
-

A.1.2 Secret Reconstruction

Algorithm 3 Reconst

Input: $\{c_{k_0}, d_{k_0}\}, \dots, \{c_{k_{m-1}}, d_{k_{m-1}}\}$ **Output:** $\{b_0, \dots, b_{m-1}\}$

- 1: **for** $i \leftarrow 0$ **to** $m - 1$ **do**
 - 2: $s_{k_i} \leftarrow \text{FindXOR}(|b|, d_{k_i}, c_{k_i})$
 - 3: $j_{k_i}, t_{k_i}, z_{k_i} \leftarrow \text{LocateIndices}(m, k_i)$
 - 4: $v_{k_i} \leftarrow []$
 - 5: **for** $x \leftarrow 0$ **to** $m - 1$ **do**
 - 6: **if** $(x == j_{k_i})$ **or** $(x == t_{k_i})$ **or** $(x == z_{k_i})$ **then**
 - 7: $v_{k_i}[x] \leftarrow 1$
 - 8: **else** $v_{k_i}[x] \leftarrow 0$
 - 9: $v_{k_i}[m] \leftarrow s_{k_i}$
 - 10: **end if**
 - 11: **end for**
 - 12: **end for**
 - 13: $Q \leftarrow [v_{k_0}, v_{k_1}, \dots, v_{k_{m-1}}]^T$
 - 14: $Q' \leftarrow \text{GaussEliminate}(Q)$
 - 15: $\{b_0, \dots, b_{m-1}\} \leftarrow \text{filter}(Q')$
 - 16: $S \leftarrow b_0 || \dots || b_{m-1}$
 - 17: **return** S
-

Algorithm 4 FindXOR

Input: $|b|, d_{k_i}, c_{k_i}$ **Output:** s_{k_i}

- 1: $M_{k_i} \leftarrow \text{list_permutation}(|b|, d_{k_i})$
 - 2: $s_{k_i} \leftarrow M_{k_i}[c_{k_i}]$
 - 3: **return** s_{k_i}
-

Algorithm 5 LocateIndices

Input: m, k_i **Output:** $j_{k_i}, t_{k_i}, z_{k_i}$

```
1:  $count \leftarrow -1$ 
2: for  $j \leftarrow 0$  to  $m - 3$  do
3:   for  $t \leftarrow j + 1$  to  $m - 2$  do
4:     for  $z \leftarrow t + 1$  to  $m - 1$  do
5:        $count++$ 
6:       if ( $count == k_i$ ) then
7:          $j_{k_i} \leftarrow j$ 
8:          $t_{k_i} \leftarrow t$ 
9:          $z_{k_i} \leftarrow z$ 
10:      end if
11:    end for
12:  end for
13: end for
14: return  $j_{k_i}, t_{k_i}, z_{k_i}$ 
```

A.1.3 Share Repair

Algorithm 6 ShareRepair

Input: \mathcal{P}_{corr} **Output:** c_{corr}, d_{corr}

```
1:  $j_{corr}, t_{corr}, z_{corr} \leftarrow \text{LocateIndices}(m, corr)$ 
2: Choose  $\alpha, \beta \in \{0, \dots, m - 1\}$  such that  $\alpha, \beta \neq j_{corr}, t_{corr}, z_{corr}$ 
3:  $\{j_{r_1}, t_{r_1}, z_{r_1}\} \leftarrow \text{AscendingSort}(j_{corr}, t_{corr}, \alpha)$ 
4:  $\{j_{r_2}, t_{r_2}, z_{r_2}\} \leftarrow \text{AscendingSort}(j_{corr}, z_{corr}, \beta)$ 
5:  $\{j_{r_3}, t_{r_3}, z_{r_3}\} \leftarrow \text{AscendingSort}(j_{corr}, \alpha, \beta)$ 
6:  $\mathcal{P}_{r_1} \leftarrow \text{LocateParticipant}(j_{r_1}, t_{r_1}, z_{r_1})$ 
7:  $\mathcal{P}_{r_2} \leftarrow \text{LocateParticipant}(j_{r_2}, t_{r_2}, z_{r_2})$ 
8:  $\mathcal{P}_{r_3} \leftarrow \text{LocateParticipant}(j_{r_3}, t_{r_3}, z_{r_3})$ 
9:  $\mathcal{P}_{r_1}, \mathcal{P}_{r_2}, \mathcal{P}_{r_3}$  are required to provide  $\{c_{r_1}, d_{r_1}\}, \{c_{r_2}, d_{r_2}\}, \{c_{r_3}, d_{r_3}\}$  to the dealer
10:  $s_{r_1} \leftarrow \text{FindXOR}(|b|, d_{r_1}, c_{r_1})$ 
11:  $s_{r_2} \leftarrow \text{FindXOR}(|b|, d_{r_2}, c_{r_2})$ 
12:  $s_{r_3} \leftarrow \text{FindXOR}(|b|, d_{r_3}, c_{r_3})$ 
13:  $s_{corr} \leftarrow s_{r_1} \oplus s_{r_2} \oplus s_{r_3}$ 
14:  $d_{corr} \leftarrow s_{corr}.count('1')$ 
15:  $c_{corr} \leftarrow \text{FindShare}(|b|, s_{corr}, d_{corr})$ 
16: return  $\{c_{corr}, d_{corr}\}$ 
```

Algorithm 7 LocateParticipant

Input: $j_{r_i}, t_{r_i}, z_{r_i}$ **Output:** \mathcal{P}_{r_i}

```
1:  $count \leftarrow -1$ 
2: for  $j \leftarrow 0$  to  $m - 3$  do
3:   for  $t \leftarrow j + 1$  to  $m - 2$  do
4:     for  $z \leftarrow t + 1$  to  $m - 1$  do
5:        $count++$ 
6:       if  $(j == j_{r_i})$  and  $(t == t_{r_i})$  and  $(z == z_{r_i})$  then
7:         return  $count$ 
8:       end if
9:     end for
10:  end for
11: end for
12: return  $count$ 
```

A.1.4 Speeded Up Algorithms

Algorithm 8 FindShare_New

Input: $|b|, d_i, s_i$ **Output:** c_i

```
1:  $res = 0$ 
2:  $mask = 1 \ll (|b| - 1)$ 
3: for  $l \leftarrow |b|$  to 1 do
4:   if  $s_i \& mask$  then
5:      $res = res + gmpy2.Combination(l - 1, d_i)$ 
6:      $d_i = d_i - 1$ 
7:   end if
8:    $mask \gg= 1$ 
9:    $l = l - 1$ 
10: end for
11: return  $res$ 
```

Algorithm 9 FindXOR_New

Input: $|b|, d_i, c_i$ **Output:** s_i

```
1:  $Z = d_i$  // number of '1' bits
2:  $N = |b| - Z$  // number of '0' bits
3:  $lst = [0...01...1]$  //list consisting of  $N$  '0' bits and  $Z$  '1' bits
4:  $l = lst$ 
5:  $result = ''$  // empty string
6: for  $i \leftarrow 0$  to  $len(lst)$  do
7:    $K = \text{gmpy2.Combination}(len(l) - 1, Z)$ 
8:   if  $c_i < K$  then
9:      $result += '0'$ 
10:     $l.remove('0')$ 
11:   else
12:      $result += '1'$ 
13:      $l.remove('1')$ 
14:      $Z -= 1$ 
15:      $c_i = c_i - K$ 
16:   end if
17:    $i = i + 1$ 
18: end for
19:  $result += l[0]$ 
20: return  $result$ 
```
