AN APPROACH TO IMPROVE THE TIME COMPLEXITY OF DYNAMIC PROVABLE DATA POSSESSION

MOHAMMED K. HAWI

DECEMBER 2016

AN APPROACH TO IMPROVE THE TIME COMPLEXITY OF DYNAMIC PROVABLE DATA POSSESSION

A THESIS SUBMITTED TO
THE GRADUATE SCHOOL OF NATURAL AND APPLIED SCIENCES OF
ÇANKAYA UNIVERSITY

BY
MOHAMMED K. HAWI

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF
MASTER OF SCIENCE
IN
THE DEPARTMENT OF
COMPUTER ENGINEERING

DECEMBER 2016

Title of the Thesis: **An Approach to Improve the Time Complexity of Dynamic Provable Data Possession.**

Submitted by **Mohammed HAWI**

Approval of the Graduate School of Natural and Applied Sciences, Çankaya University.

[illegible]

Prof. Dr. Halil Tanyer EYYUBOĞLU
Director

I certify that this thesis satisfies all the requirements as a thesis for the degree of Master of Science.

[illegible]

Prof. Dr. Müslim BOZYİĞİT
Head of Department

This is to certify that we have read this thesis and that in our opinion it is fully adequate, in scope and quality, as a thesis for the degree of Master of Science.

[illegible]

Assist. Prof. Dr. Nurdan SARAN
Supervisor

Examination Date: 13.12.2016

Examining Committee Members

Assist. Prof. Dr. Reza ZARE HASSANPOUR	(Çankaya Univ.)	[illegible]
Assist. Prof. Dr. Nurdan SARAN	(Çankaya Univ.)	[illegible]
Assist. Prof. Dr. Ihsan Tolga MEDENI	(Yıldırım Beyazıt Univ.)	[illegible]

STATEMENT OF NON-PLAGIARISM PAGE

I hereby declare that all information in this document has been obtained and presented in accordance with academic rules and ethical conduct. I also declare that, as required by these rules and conduct, I have fully cited and referenced all material and results that are not original to this work.

Name, Last Name	:	MOHAMMED K. HAWI
Signature	:	[illegible]
Date	:	13.12.2016

ABSTRACT

AN APPROACH TO IMPROVE THE TIME COMPLEXITY OF DYNAMIC PROVABLE DATA POSSESSION

HAWI, Mohammed
M.Sc., Department of Computer Engineering
Supervisor: Assist. Prof. Dr. Nurdan SARAN

December 2016, 49 pages

In this thesis, we aim to take some actions for alleviating the fears when the data storage over outsourcing, and guarantee the integrity of the files in cloud computing. In this study, we have suggested some ideas to improve FlexDPDP scheme [13]. Particularly, proposed scheme successfully reduces the time complexity for verifying operations between the client and the server. The proposed scheme is a fully dynamic model. We involved some parameters to ensure the integrity of the metadata. In spite of the fact that auxiliary storage expenditure by Client-side (the client stores approximately 0.025% size of the raw file). The remarkable enhancement in this proposed scheme is reducing the complexity. The complexity of the communications and the computations decreased to $O(1)$ in both Client-side and Server-side during the dynamically update (insertion, modification and deletion operations) and challenge operations.

Keywords: dynamic provable data possession, cryptographic hash function, RSA-tree modular, proofs of storage security**,** Cloud Computing**,** Flex-List.

ÖZ

DİNAMİK KANITLANABİLİR VERİ DEPOLANMASINDA ZAMAN KARMAŞIKLIĞINI GELİŞTİRMEK İÇİN BİR YAKLAŞIM

HAWI, Mohammed

Bilgisayar Mühendisliği Yüksek Lisans Bölümü

Danışman: Assist. Prof. Dr. Nurdan SARAN

Aralık 2016, 49 sayfa

Bu tezde, Bulut Bilişim (cloud computing) işlemindeki dosyaların bütünlüğünü garanti altına alınması ve böylece veri depolamak için dış kaynak kullanıldığında endişelerin hafifletilmesi için bazı işlemler yapılması hedeflenmiştir. Dosyaların bütünlüğünün sağlanması için pek çok makale yayınlanmıştır. Önerilen çalışmada, Flex List veri yapısını kullanan DPDP (Dinamik İspatlanabilir Veri Bulundurma) [13]. Yöntemini geliştirmek için bazı fikirler öne sürülmüştür.. Ancak bilhassa bu yöntem ile istemci ve sunucu arasındaki doğrulama işlemleri için başarıyla zaman kazandırılmıştır. Önerilen şema tamamen dinamik bir modeldir. Meta verilerin bütünlüğünü sağlamak için bazı parametrelere yer verdik. İstemci tarafında yardımcı yedek depolama giderine rağmen (istemci ham dosyanın yaklaşık % 0,025 boyutunu depolar), önerilen yöntemdeki dikkate çekici iyileştirme karmaşıklığın azaltılmasıdır. Hesaplamaların karmaşıklığı, veri yükseltme (ekleme, değiştirme, silme) ve zorlama işlemleri sırasında hem istemci hem de sunucu tarafında $O(1)$'e düşürülmüştür.

Anahtar kelimeler: Dinamik kanıtlanabilir veri bulundurma, Şifreleme ile ilgili özetleme fonksiyonu, RSA-agaç modüler, Depolama güvenliği belgeleri, Bulut Bilişim, Esnek-Liste (Flex-List).

ACKNOWLEDGEMENTS

Foremost, I praise Allah, the almighty for granting me the opportunity and ability for proceeding successfully.

Due to the guidance and assistance from several people, this thesis has appeared in current form. Therefore I want to offer my profound gratitude and sincere thanks to all of them. Profoundly thanks to my supervisor Assist. Prof. Dr. Nurdan SARAN for providing valuable advice with insightful discussions. Profoundly gratitude sincere thanks to my family, for unfailing support and providing me with continuous encouragement, during my life and especially the years of study. Thank you. Perhaps, I cannot express enough about my gratitude and fortunate fairly; to all my friends especially my roommates in abroad; for assisting me in many different ways, for handling the paperwork, for making our apartment as friendly as possible, for the joyful gatherings. Therefore and more, thank you so much I will never ever forget you.

Last but not least, I dedicate this thesis to my earlier heaven, gentle soul and loving mother, to the beloved memory of my father (you will always be remembered), successfully they have made me the person who I am becoming.

TABLE OF CONTENTS

LIST OF FIGURES

FIGURES

LIST OF TABLES

TABLES

LIST OF ABBREVIATIONS

DPDP	Dynamic Provable Data Possession
CC	Cloud Computing
CSP	Cloud Service Providers
IaaS	Infrastructure-as-a-Service
PaaS	Platform-as-a-Service
SaaS	Software-as-a-Service
PDP	Provable Data Possession
HVT	Homomorphic Verifiable Tags
POR	Proof of Retrievability
RBASL	Rank-Based Authenticated Skip List
ASL	Authenticated Skip List
Flex-List	Flexible Length-Based Authenticated Skip List
CHF	Cryptographic Hash Function
CPV	Challenge Prove-Verify
DPG	Data Possession Game

CHAPTER 1

INTRODUCTION

This chapter briefly describes Cloud Computing paradigm, which is an emerging computing model over participating with a gather of the resources in Section (1.1). As well as, epitomizes the fundamental defy facing cloud service provider to be widely deploying and usage in Section (1.2). Supplementary with this chapter characterizes the motivation of adapted working in Section (1.3). Finally, it outlines of the contributions and thesis regulation in Section (1.4).

1.1 Cloud Computing

Cloud Computing (CC) is an ingenious technology grabbing attention. This technological service has a scalable to enable high estimation to be used readily on the Internet. This scheme is spreading computational along with a spacious framework of virtual spread computing sources.

CC is a tendency toward the management readily and the extensive flexibility. Which is impacting on the individuals and corporations for them data, when its storage outsourced like the clouds or server (e.g., box.net, Dropbox, Amazon S3, iCloud). Exemplify, there is an intention for usage the computing services such as usage the general services like gas, water etc.

Its Architecture has subdivided into Frontend and Backend whereas:

- **Frontend** symbolizes to cloud users, establishments, or implementations utilizes that from the cloud facilities.
- **Backend** is a large information centers beside a lot of the applications various, system software, and memory systems. Such as illustrated in Figure 1.

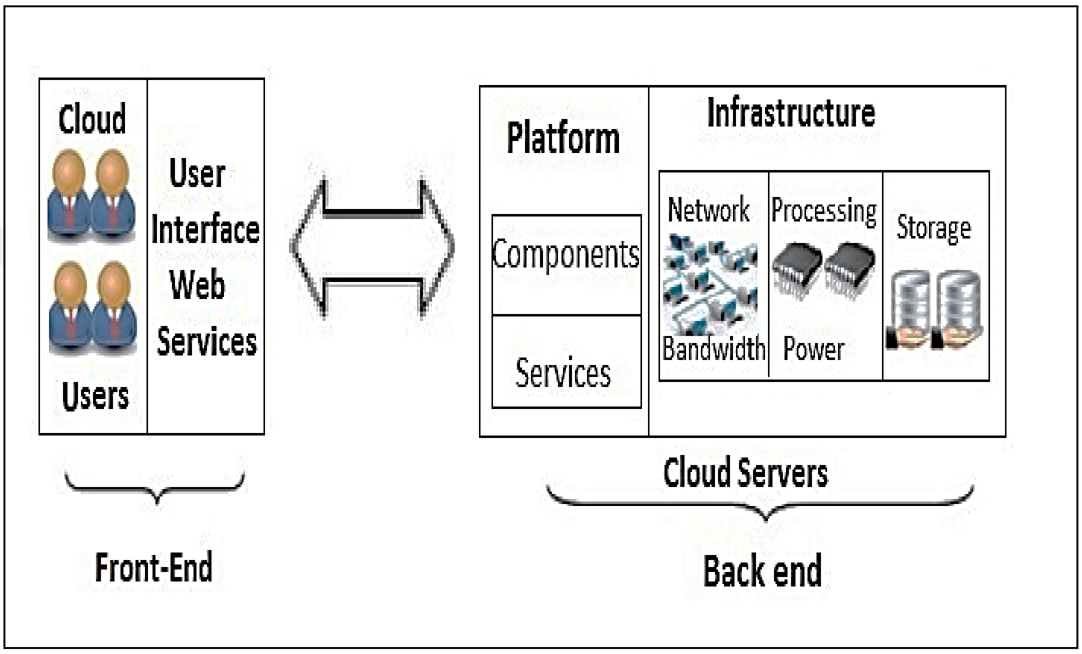


Figure 1 CC architecture

1.2 Cloud Service Provider

Cloud Service Providers (CSP): has substantial resources for controlling servers, storage allocation, and management database of its servers or clouds. These resources as a virtual infrastructure to provide the applications for hosting services. Those services are may be used from the client's side to manage its data, which it is stored onto servers or clouds.

CSP is a web join to Client for storage data at a range of the CC, which operates in a cooperation and apportions method. However, the clients are enabled to employ any applications inclusive the operating system and itself.

Agree with [1] CC Architecture might be located with diverse models:

- **Public**: is overtly attainable by generic patronages and organizations, till exchange specified fees affording by usage the CSP's services.
- **Private:** is devoted to an organization which may manage the infrastructure or leave that management to a third party.
- **Hybrid:** is taking shape of many clouds either *private* or *public*. The organizations afford and handle some exterior and interior resources.

Moreover, CSP depends on the metadata servers to set Clients IDs to data IDs that is storing and determining as groups. The CSP hosts are a collection of software and pieces of advanced equipment into its hardware. Which it is used by the developers to impact on their applications [1] and [2], its might classify into:

1. **Infrastructure-as-a-Service (IaaS)** is a service that can provide the functionalities of an entire infrastructure including storage, networks, any platform and any number of computers. The customers could use this service by configuring a Virtual Machine on the infrastructure, on which an operating system installed. They employ the middleware for connection with other applications.
2. **Platform-as-a-Service (PaaS)** is proposing an advance platform on a vertex of the services delivered with IaaS, which applications are working with, software developers not needed to install the software building tools on the computer.
   The provider's service are enabled its users for installing their applications on a platform, which can provide a simulated with various kinds of hardware.
3. **Software-as-a-Service (SaaS)** is a highly customary service in the CSP, which transmits the software's implementation onto the web. SaaS provider transmits their products when facilitated usage the whole Server on one base. Otherwise, it utilizes another Server equipment dependent on the Client interests. This operation is usually done by utilizing a licensed model where applications may be licensed directly organizing by a group of users.

They manage multiple licenses between the users' organizations.

Additionally, there is one more infrastructure is Middleware, which is a base that suitably supports progress and disseminates execution of the applications [2]. That gives a protection among the applications, then the users will get an entry to the applications through any Internet device. Figure 2, demonstrates these types with a clear view of their usage.

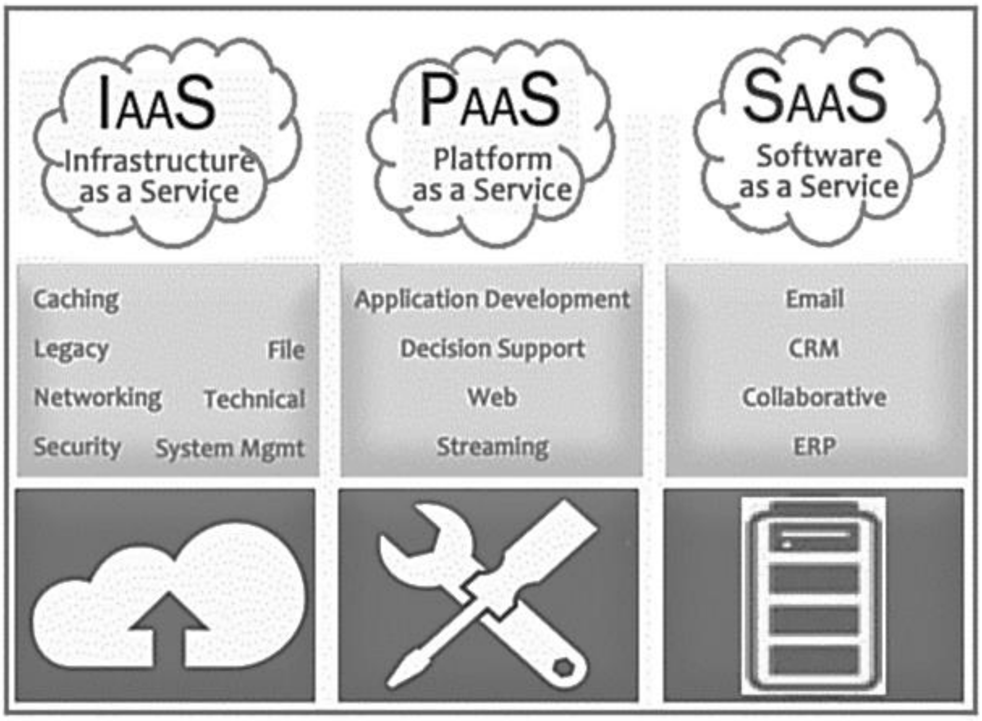


Figure 2 Categories of the CSP model

1.3 Motivation

Data outsource such as the cloud may not physically possession. The data proprietor must capable of verifying that data with stored one's at the Cloud Storage Provider is robust and secure.
Cloud must be appropriate for any user, a user may investigate of possessing data storage without requiring a typical copy of the metadata. The data proprietors have requirements about data to be confidential when verifying its reliability.

- **Communication complexity:** Value of the communication among the clients and server must be less complexity.
- **Storage cost:** auxiliary storage from client and server desired by the scheme must be isolated down of raw data.
- **Data recovery:** each method must be assisting within recovery the data in the situation of data loss.
- **Provable integrity:** The scheme must be secure and the data should not be altered.

Overhead, the research questions revolved around the adapted scheme can be illustrated by the following:

- How to improve the time complexity and how to bolster the security of the Dynamic Provable Data Possession scheme (DPDP)?
- What is the difference in outcomes between the adapted scheme and recent schemes?

1.4 Thesis Outlines

We have improved a system and practical control on the entire DPDP scheme whereas the Server has the capability to guarantee the data integrity of the Client when the clients do not download the total metadata.
The details of this thesis existing in particular chapters as described here:

- Chapter 2 have a background of the various Static Provable Data Possession (PDP) and Dynamic DPDP schemes, their challenges, and features. We start with the static one at its PDP schemes on data, thereafter we complete adapted survey around the dynamic at its DPDP schemes on data.
- Chapter 3 have a review of the various Skip-List and Flex-List methods, their structures, and features. We start with the Skip-List methods with different versions, after that, we complete adapted survey around the Flex-List method with definitions several operations that are dealing with.
- Chapter 4 have adapted proposed system, within added some extension parameters and measures the size of these extensions in the auxiliary

storage section. Moreover, it is shown the dynamics of setup these parameters, that shown in details how this system is dealing with the upgrade operations.

- Chapter 5 have a comparison with other dynamic schemes, through calculates the complexity of adapted scheme. Then shown the result we have got according to all that work.
- Finally, Chapter 6 totalizes the conclusions of adapted scheme and awarded a guidance for future work.

CHAPTER 2

BACKGROUND

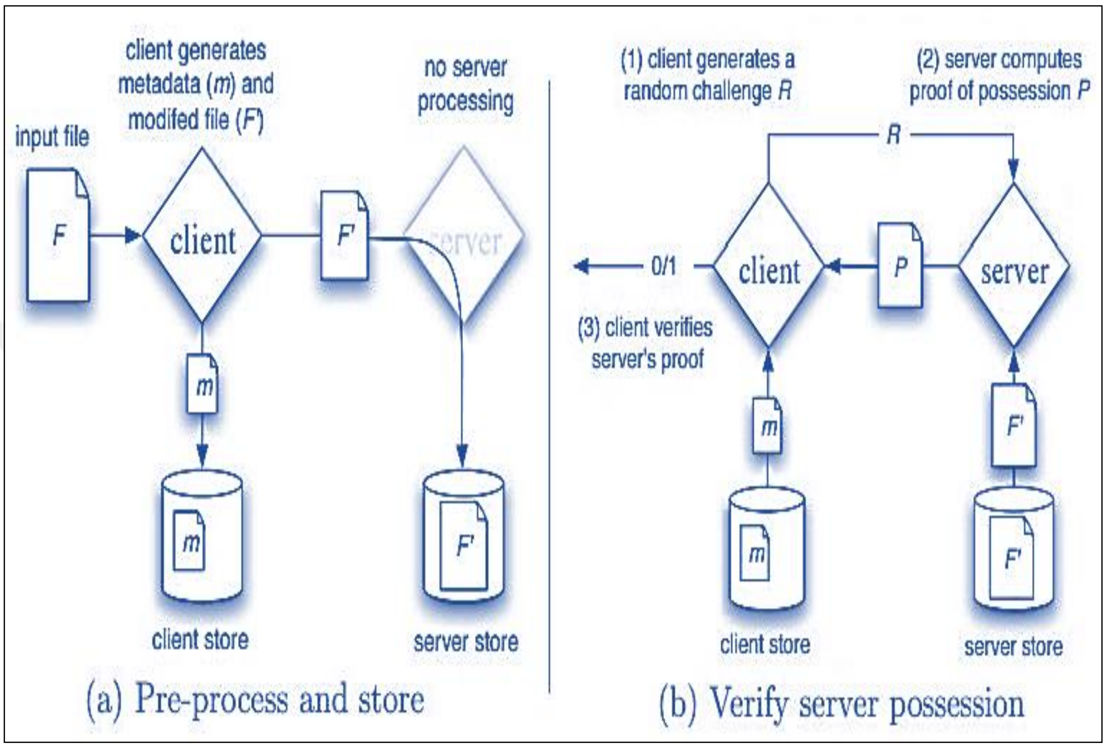
In Section (2.1), we have reviewed the PDP schemes of static data and provide the rapport backward of their features, then give the challenges of them.
Section (2.2), the basics design of outsourcing data by providing the DPDP schemes have been described, then the challenges have been given.

2.1 Static (Provable Data Possession)

PDP is a strategy permits for substance toward demonstrating the datum. The ownership of the datum is approving the possess datum in excess of the remote servers. In the common PDP schemes, the data proprietor creates a digest for a data record for utilizing later to confirm purposes through test the traditional reaction per the remote cloud/server [3], [4] and [5]. The proprietor refers a report to be secured on a remote server, which could be untrusted, while remove a clone parts from the record. Whereas the main data proprietor or the trusted substance that grants a few information to the proprietor, to confirm that the Server has the datum record. The server needs toward accurate where the process reaction to test a vector sent from verifier.
Briefly, PDP authorizes the verifier to be an effectively and safely to approve over the remote server, Figure 3 shows the protocol of the scheme [3].
In another words, PDP is a cryptographic convention that empowers a customer to review his/her information over a remote server without getting the entire record. Although checking the data reality when it is putting away upon the server, the techniques during attacks take a high cost and give a low outcome.

Cryptographic *tags* give a reasonable ensures, even though it maybe require the whole record be retrieved.

Generally, PDP models possess the property when it is provided a probabilistic assurance and it is intensified an exponentially through redundancy.



input file
F
client generates metadata (m) and modifed file (F')
client
F'
no server processing
server
m
client store
server store
(a) Pre-process and store
(1) client generates a random challenge R
(2) server computes proof of possession P
R
0/1
client
P
server
(3) client verifies server's proof
m
F'
client store
server store
(b) Verify server possession

Figure 3 The protocol of PDP [3]

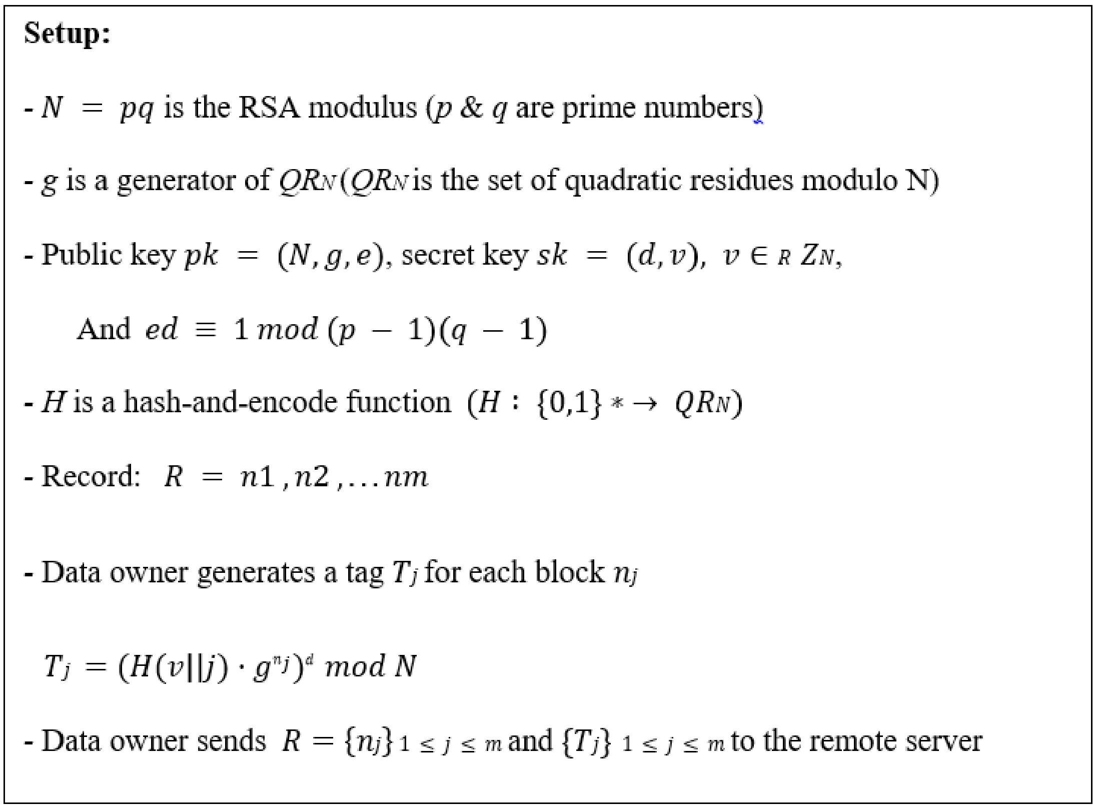
2.1.1 PDP Scheme Based on RSA and HVTs

The scheme proposed in [3]; confirms data integrity utilizing the RSA for generating the Tags; pre-processes the data before storage in outsourcing and present the idea of the *Homomorphic Verifiable Tags* (HVTs), Tags depend on RSA signature. RSA-Tags have *Homomorphic* possessions where numerous data blocks can be confirmed in meantime.

Fundamentals situation of this scheme demonstrates the integrity of numerous from reproductions and manages the operation of the static data, because of the extent of RSA-Tags is significant. Approved clients are needed to identify which duplicate has existed particularly reclaimed from the CSP to detect the harmful block before decoding. The schemes in [3] and [5] are working with static data only.

These HVTs proceed for confirmation the metadata of data blocks. HVT modular considers a record R to be a limited requested accumulation of n node (data blocks since $R = (n1, n2 \ldots nm)$), while gives two qualities, Tn_i *and* Tn_j as generate the tag Tn for node n.

Figure 4, explains a summary of the PDP scheme. The remote server gives a proof for the data, the data is not deleted or changed through reacting the challenge referred to the verifier. While the PDP authorizes a subsection of data per every challenge, it is the probabilistic prove of data possession.



Setup:

- $N = pq$ is the RSA modulus (p & q are prime numbers)
- g is a generator of QR_N (QR_N is the set of quadratic residues modulo N)
- Public key $pk = (N, g, e)$, secret key $sk = (d, v)$, $v \in_R Z_N$,

  And $ed \equiv 1 \, mod \, (p - 1)(q - 1)$
- H is a hash-and-encode function $(H : \{0,1\} * \rightarrow QR_N)$
- Record: $R = n1, n2, \ldots nm$
- Data owner generates a tag T_j for each block n_j

  $T_j = (H(v||j) \cdot g^{n_j})^d \, mod \, N$
- Data owner sends $R = \{n_j\}_{1 \leq j \leq m}$ and $\{T_j\}_{1 \leq j \leq m}$ to the remote server

Figure 4 The setup of PDP scheme [3]

The schemes in [3] and [4] have determined numerous requirements of other PDP conventions; that enable when taking a large HVTs which depend on RSA generally; each data blocks have a HVT altogether of N bits; Consequently, towards accomplish 80 $bits$ security level, and created tag of size 1024 $bits$.

Briefly, HVTs are unforgeable metadata check and built from the record blocks in a manner. The verifier can be satisfied with the record blocks as precisely, by confirming the accumulated tag/authenticate.

2.1.2 Similar and Other Schemes of PDP

Proof of Retrievability (POR) [6], provided a various methodology from PDP, however both of them have a related way to deal with the validity of the remote data by testing or checking. POR is considered as a first scheme to characterize the sort of conventions. Both of PDP and POR are correspond together by; divides the record (the data file) into blocks; stores the data remotely; checks the record's integrity, and utilizes test reaction "challenge-response" protocol.

POR has one more property that tests the reaction protocol concedes as "extractor," so that an effective calculation can be utilized to reproduce the raw record.

The POR methods utilize unique blocks are called as the sentinels, and covered up between different blocks in the record. The POR [6] is particular via the number of sentinel's blocks, which they are embedded inside the record, and it permits a few numbers of challenges on the records.

In [8], authors use some functions like a one-way function and another one to achieve method to decrease the communication cost, but it does not satisfy the required verification standard.

The scheme that offered in [9] verifies the data integrity by utilizing the RSA-based homomorphic-hash-function, which has a limitation similar with [8], by forcing the service provider to archive within an exponential growth of the whole file, that computational overhead is hefty with large files especially.

In [10], authors cancel the issue of exponentiating of the whole file; through chunks interested in blocks; fingerprints all blocks, and utilizes the RSA for blocks, but, it is supporting the private verifiability only.

2.1.3 PDP Scheme Challenges

There are many criteria used to improve this scheme, it is useful and powerful or not, constructed by the following:

- **Owner pre-calculation:** when the owner perform such operations on the metadata, to prepare the data before being outsourced, that will be called a calculation.
- **Verifier capacity overhead:** to store some metadata on the verifier side, to

be utilized later, the additional capacity will be required.

- **Server capacity overhead:** metadata needs an additional storage at the server which is called overhead capacity.
- **Server calculation:** high level of security operations, which are applied by the server.
- **Verifier computation:** a verification methods that are performed by the verifier.
- **Communication cost:** the cost is simply a bandwidth which is necessary when the challenge takes a place.
- **Unbounded difficulties:** the method authorities a number of examining the data record.
- **Fragmentation:** divides the record into smaller blocks.
- **Kind of certification:** probabilistic insurance that relies upon spot checking.
- **Demonstrate information ownership:** whether the method demonstrates the possession of the data itself, or demonstrates that the server is putting away something in any event as expansive as the native record.

2.2 Dynamic (Dynamic Provable Data Possession)

DPDP in [7] is a setting for distributed storage data, which primarily presents the rank-based authenticated skip list (RBASL) such as a new data structure, as shown in Figure 5.

In the DPDP model, there is a client who needs to outsource record and a server that assumes responsibly for the capacity of the data. The client pre-processes the record and keeps up metadata for screening the confirmations from the server, then it refers the data case to the server.

The client needs to check whether its data integrity or not, challenges some random blocks. the server generates the proof for challenge and backward, the client at that point checks the data integrity of the record using this proof.

As indicated by this method, record R is divide into v nodes (blocks) $v_1, v_2, \ldots v_n$. The tag $T(v_i)$ of blocks v_i is located in the appropriate position at the i^{th} below-level nodes of the RBASL. Blocks v_i will be located somewhere else by the cloud.

Each node v of the RBASL stores the portion of nodes at below-level that can become for v, that value of node is known as the rank of v and indicated by $r(v)$.
The furthest-top-left node of the Skip-List will have alluded to the begin node w_i. For a node n, the records of the furthest-left and furthest-right nodes at the below-level Reachable from v_i. It indicated by low (v_i) and high (v_i) individually as shown in Figure 5. The hash value of the nodes in Skip-List when updated or modified nodes are reachable and recomputed with the path, whereas the nodes updated are referred to the user for authentication.

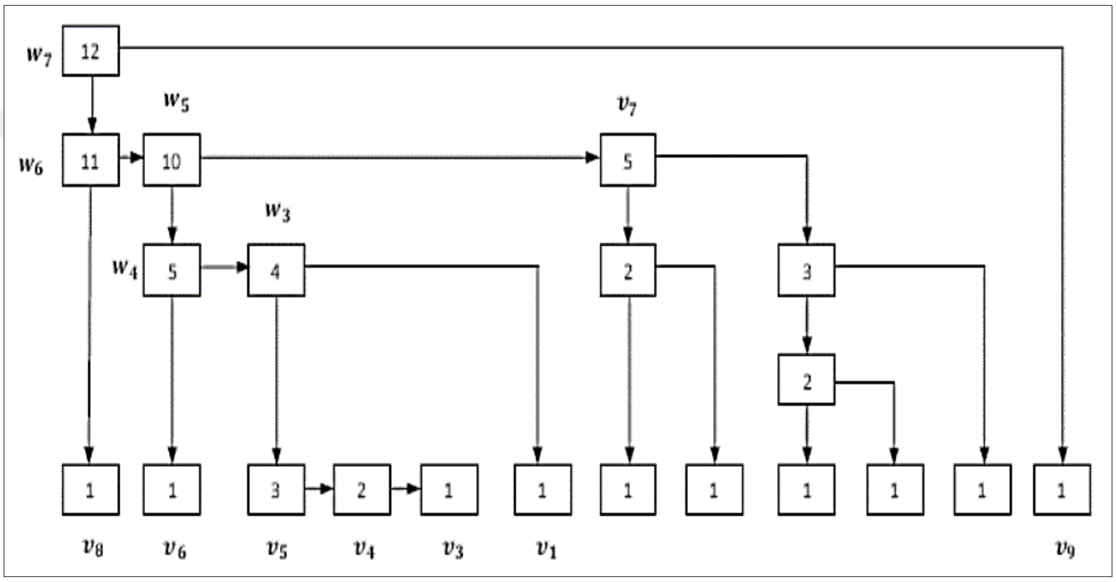


Figure 5 The RBASL in DPDP scheme [7]

RBASL has utilized to update operations and tags through that it has achieved the dynamic behavior of data blocks. RBASL nodes alongside with verify are influenced by the dynamic operations of file blocks solely.
Each level and node of RBASL in specific conditions must be stored also the aggregate hash value of the nodes from left to right in below-level, these hash values entries act as the metadata.
The integrity verification in [7] scheme is similar to the conventional schemes but it supports data dynamics and verifies data block within update operations.

2.2.1 RSA-Based DPDP Scheme

There is a second scheme in [7] that has displayed a variation of the DPDP scheme utilizing RSA-trees rather than (RBASL) as the second plan of its worked.

The second scheme of DPDP (based on RSA-trees) permits that any individual who identifies the proprietor's public key be able to challenge the outlying server and check the server “is yet having the proprietor's records?”.

In [11] modify the method in [10] that makes the verification publicly on dynamic data. It permits a person identifies the public key to challenge the remote server and verify it by keeping the possessing data. In the case of doubt with data integrity and possession validity, when transmitted between the proprietor and the CSP. Hence, the outsider inspector can figure out the secure of data. These outsiders utilized in this scheme for auditing system, where no private data is leaked. The scheme displayed in [11] guarantees that the information is kept privately over the outside confirmation.

2.2.2 Similar and Other Schemes of DPDP

PDP scheme in [4] offers “dynamicity model based on Cryptographic Hash Function and symmetric key encryption in dynamic version”. However, the issue of updated block in PDP is cannot explicitly, because it is supported only the append operation.

There are several schemes covered DPDP with using RBASL, one of them in [12] divides record into blocks, makes a tag for each block to ensure the respectability of the index blocks and enrolls a hash value for each tag to ensure the wellbeing of the tags.

The added matrixes in [12] have increased in a few amount of the raw file size as shown in Figure 6. Due to the lack of supply variable block sized operations with an RBASL, this scheme is not fully dynamic like adapted scheme.

Flex-DPDP scheme [13] offers a new data structure (Flex-List), which provides a variable-size of blocks utilized for constructing the DPDP protocol.

Another work of the Flex-DPDP [14] utilizes a balanced Flex-List for permitting a variable-sized of that block which provides the flexibility to deal with the arbitrary length changes.

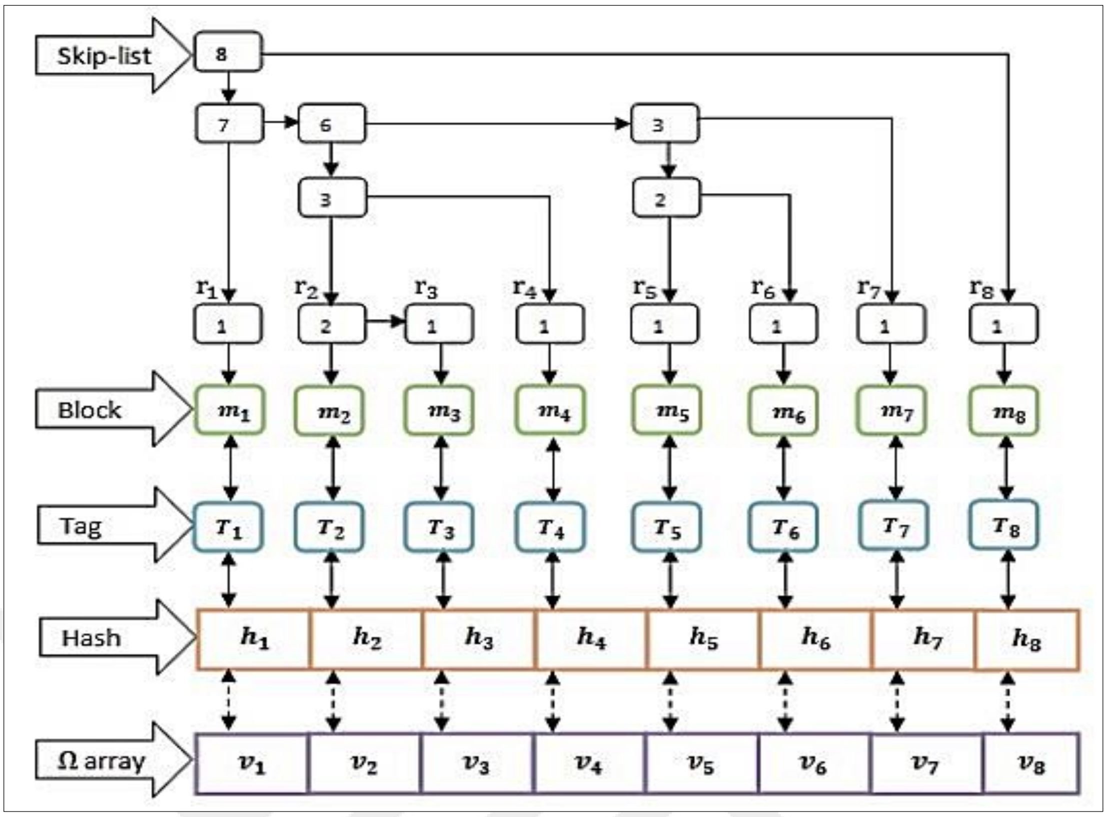

Skip-list
8
7
6
3
3
2
r_1
r_2
r_3
r_4
r_5
r_6
r_7
r_8
1
2
1
1
1
1
1
1
Block
m_1 m_2 m_3 m_4 m_5 m_6 m_7 m_8
Tag
T_1 T_2 T_3 T_4 T_5 T_6 T_7 T_8
Hash
h_1 h_2 h_3 h_4 h_5 h_6 h_7 h_8
Ω array
v_1 v_2 v_3 v_4 v_5 v_6 v_7 v_8

Figure 6 Improved DPDP scheme [12]

2.2.3 DPDP Scheme Challenges

The scheme in [7] faces with the challenges by the following procedures:

- **Key-generation:** controlled by data proprietor (*customer*), and the outcomes are secret key sk and public key pk. The secret key is preserved by the proprietor, but the public key is transmitted to remote server.
- **Update-preparation**: controlled by the proprietor with nominate a part of the file to store onto the remote server. The vectors are sk and pk as the input parameters and make the comprehensive update cross the metadata Mc (e.g., full re-write, modify the block, delete block, or insert block).
  The outcomes are transmitted to remote server, which they are; an encoded form of file $e(F)$; encoded metadata $e(info)$ passed off the update and for getting the metadata $e(M)$.
- **Update-execution:** when gets an update demand from the proprietor, the server runs this step in responding to the demand. The vectors are pk, file

denoted as $Fi - 1$, metadata $Mi - 1$, and the values produced by the client during Update-preparation step.

The outcomes are the new release from the file Fi, metadata Mi, and metadata Mc is transmitted to the proprietor for proof PMc.

- **Update-verification:** running by the proprietor to verify a behavior of the server through the updates. The vectors are all inputs of Update-preparation step, the metadata Mc, and the proof PMc (Mc and PMc are submitted by the server as the outcomes of the Update-execution step).

  The outcome calculation is either *accept* or *reject* signal.
- **Challenge:** controlled by the proprietor to challenge the server and confirm the integrity of storage data remotely. It picks as info (sk, pk), and the most recent proprietor metadata Mc.

  The outcome is a challenge C that is transmitted from proprietor to server.
- **Proof-computation:** After getting C from the proprietor, the server runs Proof computation in order to test that. The vectors are pk, the most recent version of the file, the metadata, and the challenge C.

  The outcome is a proof P that is transmitted to proprietor.
- **Proof-verification:** controlled by the proprietor to prove P got from the server. The vectors are (sk, pk), the customer metadata Mc, the challenge C, and the proof P transmitted by the server.

  The outcome is *accept* or *reject* generally.

CHAPTER 3

DATA STRUCTURES IN DATA POSSESSION SCHEMES

There are several data structures that have been used in data possession schemes such as skip list, rank based skip list, Flex-List etc. In this chapter, we have summarized these structures and associated their effects in data possession.

3.1 Skip List

Skip List is a data structure (probabilistic method) introduced as the other option of balanced trees [15], and “it is a hierarchical structure of Linked-Lists”.
It utilizes for storage a sorted set of items among the constructions that empower dynamic operations. It offers search, modify, insert, and remove operations with logarithms multifaceted nature in high probability.
It is easy to structure but difficult to execute without complex balancing and rebuilding operations, which keeps nodes requested by key-values. Each node n in the typical Skip-List stores with right and down pointers, to be utilized during search system for a particular target at leaf-level nodes (in below-level nodes) [15].
In the typical Skip List nodes consist of key, level and data, where the data is being at below-level nodes. While the connections consist of connection-below and connection-after (e.g., $n_2.below = n_3\ and\ n_2.after = n_4$).
If a search for the node with key 27 will be conducted, the search begins from n_1 (root) and keeps track of n_2 (the next connection). $n_1's$ connection-after leads to the node that has the greater key-value than its key-search as $\infty > 27$.
At that point, within n_2 keeps track of $C1$ connection to n_4, when key-value of n_4 is smaller than or equal to the key-search.
Generally, whether key-value of the node where connection-after leads to smaller or equal to the key-search then must keep track of that connection, otherwise remain

track of that connection-below. By utilizing the same resolution mechanism, hence keeping track of that blue-connections till the key-search is hit at the below-level. Otherwise, it does not exist and return the node with key immediately.

Figure 7 summarizes the typical Skip-List; the path of search node with key 27 is shown within the blue-connections; Numbers on the left characterize the levels; Numbers exclusive the nodes are the key-values; and dashed-lines refer to the pointless connections and nodes.

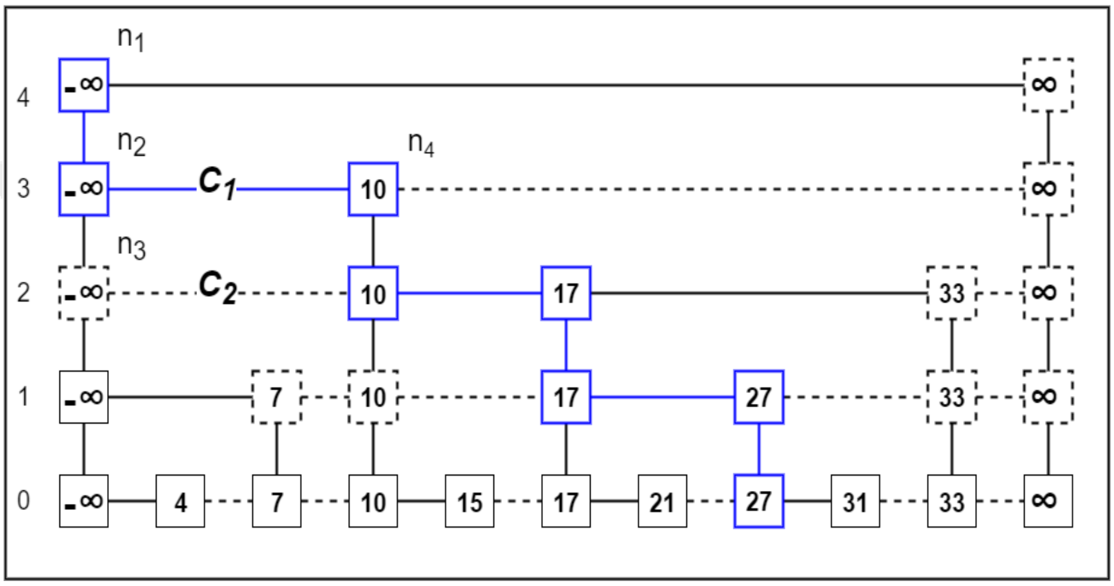


Figure 7 Skip-List model

3.2 Improvements of Skip List

Noticeably there are several connections never used in Skip-List such as $C2$ in Figure 7, even after any search within key larger or equal to 10, definitely that will keep track of $C1$. While the search for the smaller key would not progress through $C2$, Hence these connection are unnecessary.

After removing the unnecessary connections, then some nodes (whose have not connection-after e.g., n_3) are unnecessary too. Therefore these unnecessary nodes do not have a new condition in Skip-List.

In spite of the fact that it does not alter the asymptotic complication, it is valuable not to incorporate them for time and space proficiency.

Detecting a connection is necessary if and only if it is on at least one search path.

Where detecting a node is necessary if and only if it is at the below-level or it has a necessity at the connection-after.

For example the optimized version of Skip-List that is illustrated in Figure 7, can be expressed in Figure 8 per the identical blue-connections, but without the unnecessary nodes and connections.

Overall, [7] enrolls a new data structure (RBASL), and [14] extends RBASL and enrolls a new data structure, Flex-List. These data structures, RBASL and Flex-List, are constructed based on Skip-List. The following section shows these models carefully.

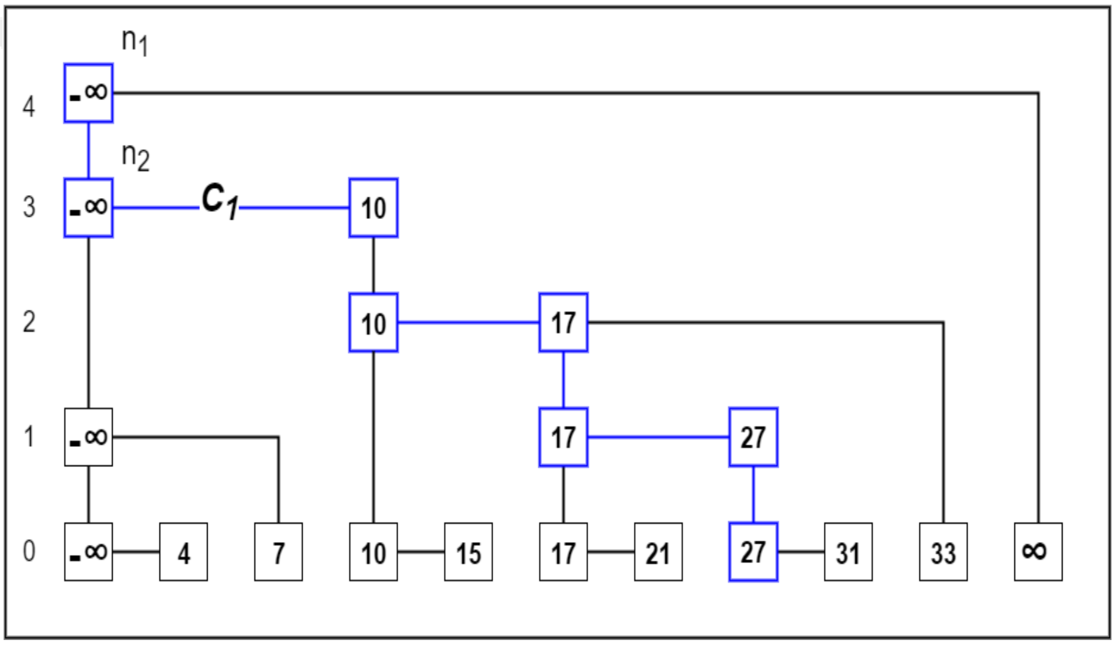


Figure 8 Skip-List (optimized version of Figure 7)

3.2.1 Authenticated Skip List

Authenticated Skip List (ASL) is developed with the utilization of an impact safe (collision-resistant) hash function and saving a hash value in every node. Nodes at below-level remain connections to the blocks, which can connected toward a various structures e.g., records, catalogs [16].

ASL may supply a proof to demonstrate whether a particular component has a place with the set constitute with the list or not. Every node *n* in authenticated Skip-List stores $rgt(v), dwn(v)$ and a name $f(v)$ recursively by relating a hash function to $f(rgt(v))$ and $f(dwn(v))$.

A hash value is computed per the accompanying efforts: *level* and *key* of the *node,* and the hash value of the both node-after and node-below. Through the efforts to the hash function, all nodes are reliant on their after and below neighbors.

The root-node is reliant on each leaf-node, and because of the collision-resistant of the hash function, knowing the hash value of the root is enough for checking the integrity later.

Note that if there is no node below or an element of data (which called tag in the accompany segments), utilized rather than the hash of the below neighbor.

3.2.2 Rank-Based Authenticated Skip List

Rank-Based Authenticated Skip List (RBASL) is vary in relation to the ASL by a method for how its listed data [7]. An RBASL has rank data (utilized as a part of hashing rather than the key value), which means what number of the nodes is reachable from that node. In spite of $rgt(v), dwn(v)$ and $f(v)$, every node *n* in RBASL stores the quantity of the nodes on the base level.

For more details about this scheme (as mentioned in chapter.2 section (2.2)) and in Figure 5 an example of RBASL is shown.

3.3 Flex-List

Flexible Length-Based Authenticated Skip List (Flex-List) is verified data structure and constructed on the length of data and dealing with variable-length blocks of data [13], "The Flex-List optimization of removing unnecessary links and nodes results in 50% fewer nodes and links on top of the leaf nodes". While the speedup of parallel build Flex-List from 8 cores reduces the time of building [14].

Respectively, the nodes preserves a hash value that computed by rank, level, the hash value of below neighbor, and the hash value of the after neighbor. The rank is the quantity of bytes that consist of node and level, which is the status of a node in Flex-List.

Respectively below-level nodes preserves a connection to the metadata (a block of the stored file) that it alludes to the length of the data, and computes a tag ascertained by metadata. Rank qualities are including the below and after neighbors' positions. The leaf-level nodes' hash value are slightly distinctive, denote that root node needy to wholly leaf-level nodes.

Additionally, Flex-List has sentinel nodes as the initial and latest nodes (E_4 and E_{13}). These nodes do not have any information where their length qualities are 0, and they don't influence the rank value of alternate nodes. The sentinel nodes do not create a new conditions, they are valuable to make the calculations simpler and more justifiable.

In Figure 9, the numbers at the bottom represent the lengths of the data blocks associated with the leaf-level nodes. The numbers in the nodes represent the ranks, and the numbers at the left side of the figure denote levels.

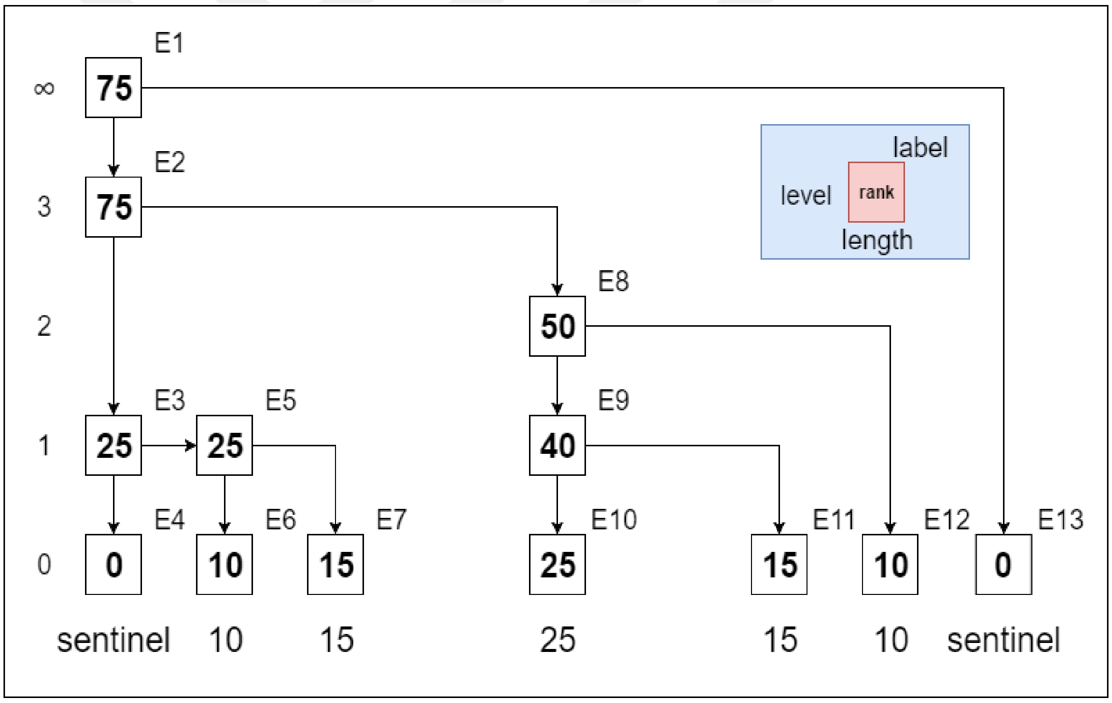


Figure 9 A Flex-List model

In Figure 9, consider node E_2 keeps two active connections, rank quality (75) and level value (3). By keep tracking the connections to reach the node-below E_3 and the node-after E_9.

The rank bytes of the genuine data can be reached by going to the specific node (e.g., the rank bytes of E_3 is $25B$). Computing the rank value can be done by adding its

values of below and after nodes (rank of node-after is taking 0 if the connection is $NULL$). The level performs the length of the node in Flex-List.
Once more, in Figure 9 consider E_5 which has a connection-after to E_7, however, it does not have a connection-below.

3.3.1 Search Example of Flex-List

Assume the block that chosen to represent the search example has the byte with value 30 through that Flex-List, see Figure 9.
Starting from the root-node (E_1) and checking a rank's node-below as a procedure for drawing the search path until achieved the desired. Since $30 < 75$ realizes that search path must keep tracking the connection-below to reach E_2. Continuously, by the same search path will get $30 > 25$, for that must keep tracking the connection-after at the second time to reach E_8.
Moreover, by keep tracking the connection-after when leaving behind several bytes on the left, whereas cannot be reached later. At that point, the operation must be subtracted these bytes from search the index. Then the search operation will complete its work with this index $30 - 25 = 5$ (rank of E_3). Sequentially, searching until reach E_9 and E_{10} respectively (the length of data for E_{10} is 25). At that point, the searching is stopped since the index 5 cannot be reached.

3.3.2 Insertion Examples of Flex-List

Insertion operation to Flex-List is a little bit complicated; should maintain on data structure optimally; removing nodes that turn out to be pointless after the insertion operation, and whether the connection point is hollow then creates a new node.
Figure 10 demonstrate the Insert Example by inserting a data block with length 10 to index 50, the main insertion point in this Example is E_8.
Starting at the root E_1 with the same principle in the Search Example, by keep tracking the connections until reached the node-before.
Continuously through following the connections will reach E_{10}, which it does not have a connection-after. Where E_{10} has achieved the desired to add a new nodes index (P_1).

At that point, creating P_1 and interfaced with E_{10} (as a leaf-level node), and determined both of the rank and hash value for the new node P_1.
After that, during follow all the connections inversely till the root, the Insert operation will be finished and achieved the desired.

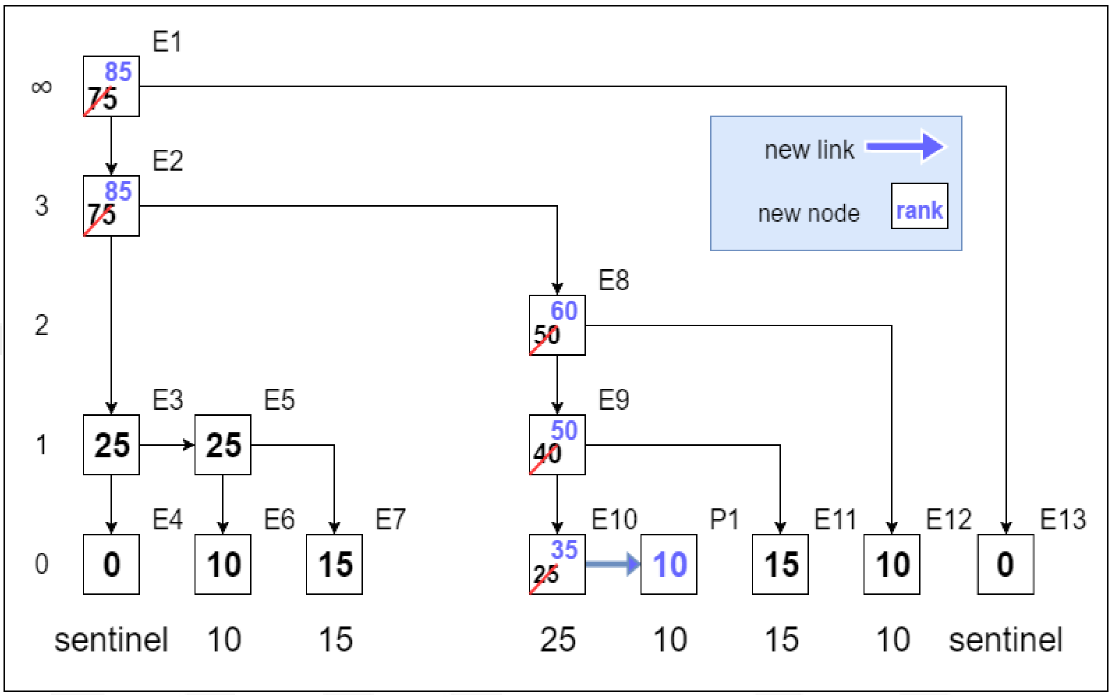


Figure 10 Insert on Flex-List

3.3.3 Removal Examples of Flex-List

Basically, the Removal operation is an opposite of the Insert operation. In the running case to delete the node P_1 with data block 10 from Figure 10 will obtain on the same Flex-List in Figure 9.

CHAPTER 4

PROPOSED SYSTEM

According to the proposed model in [12] there are some parameters added to RBASL to improve the DPDP scheme (as mentioned in Section 2.2). Unfortunately, there are some corner nodes, and lack of supply variable block sized operations in RBASL. Hence all that had been fixed by Flex-list, where that is encourage us to make the Flex-list as our data structure for constructing an adapted scheme.
There are several studies that-have used the measurement of metadata, which is available in every node in bytes (not the number of blocks) like Flex-list. The Flex-list in [13] utilizes the rank of the node as a length of the metadata that opened through that node. Whereas, the pre-processing step by parallelization gets 60% gain at server-side and 90% gain at client-side. Overall, the computational complexities of Flex-DPDP scheme are logarithmic in [14]. Among these mentioned methods, we have constructed an adapted scheme.

4.1 Preliminaries

We have explained the work dynamism of the Flex-List [13] data structure in section 3.3, by utilizing the rank's node in bytes "not the number of blocks".

The work in [12] is simply adding four parameters to the RBASL. In the adapted scheme, three parameters from the [12] are added to the Flex-List with some modification on their functions.

The metadata is being saved at the below-level i^{th} that urges to generate tags (Tag.Rank) to those nodes separately. Since Tag.Ranks (TR) will be carried on the network with the metadata packages, therefore computes the hashes (CHF) of them became a fact to increase the integrity.

Although, those operations are managed in both client and server sides, the client creates an additional array (GD matrix) towards the authentication of the metadata integrity.

In Figure 11, adapted scheme for client-side is shown with the New-Adds on Flex-List, and in Figure 12, adapted scheme for server-side is shown with the New-Adds on Flex-List.

According to all that the Server can look for a specific rank of the metadata from Flex-List.

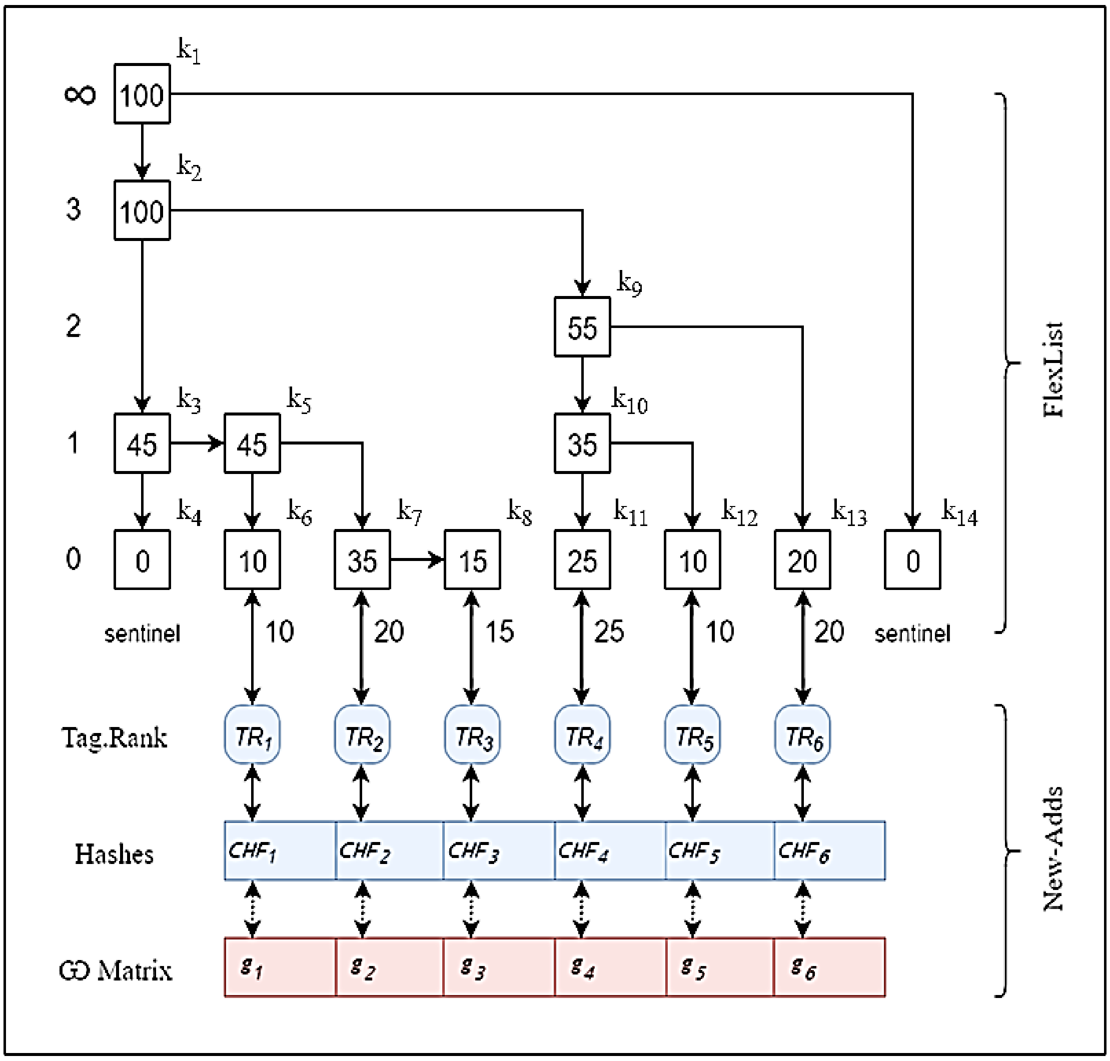


Figure 11 Structure of adapted scheme in client side

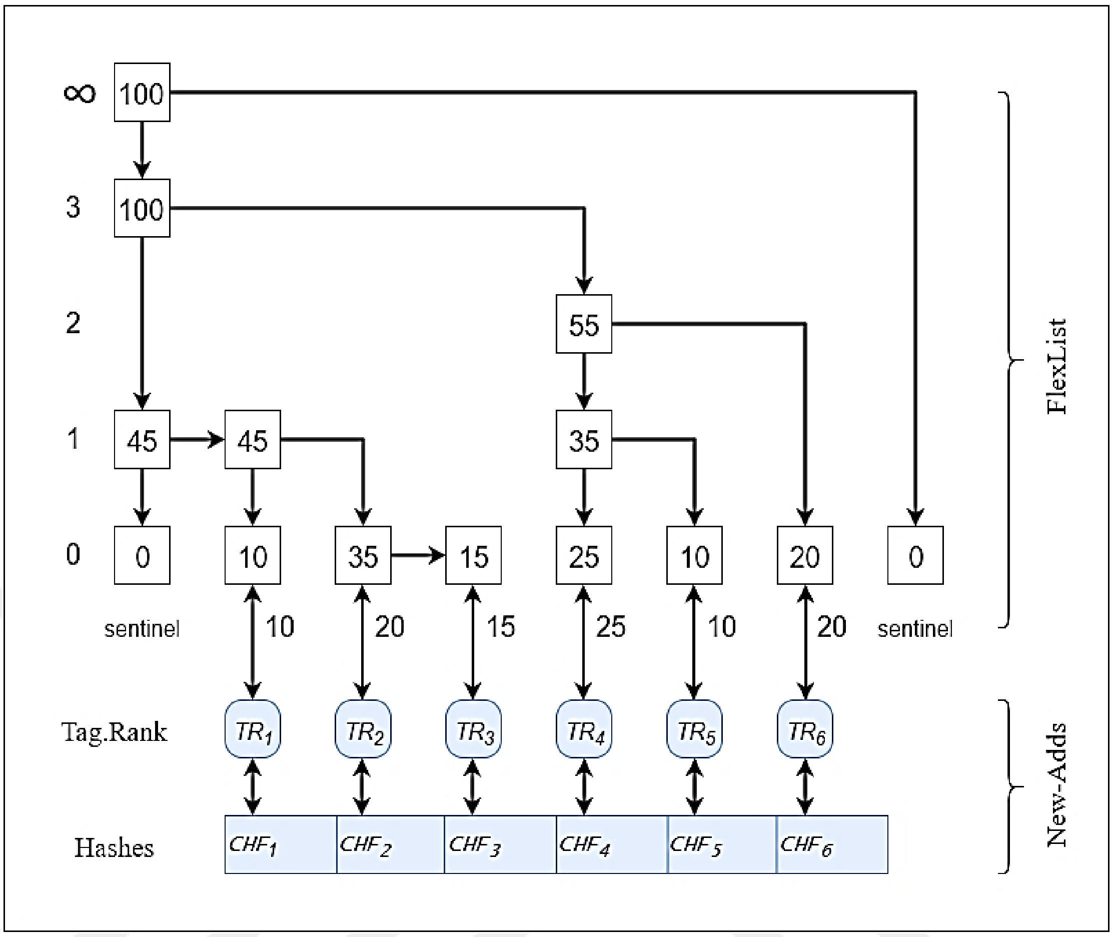
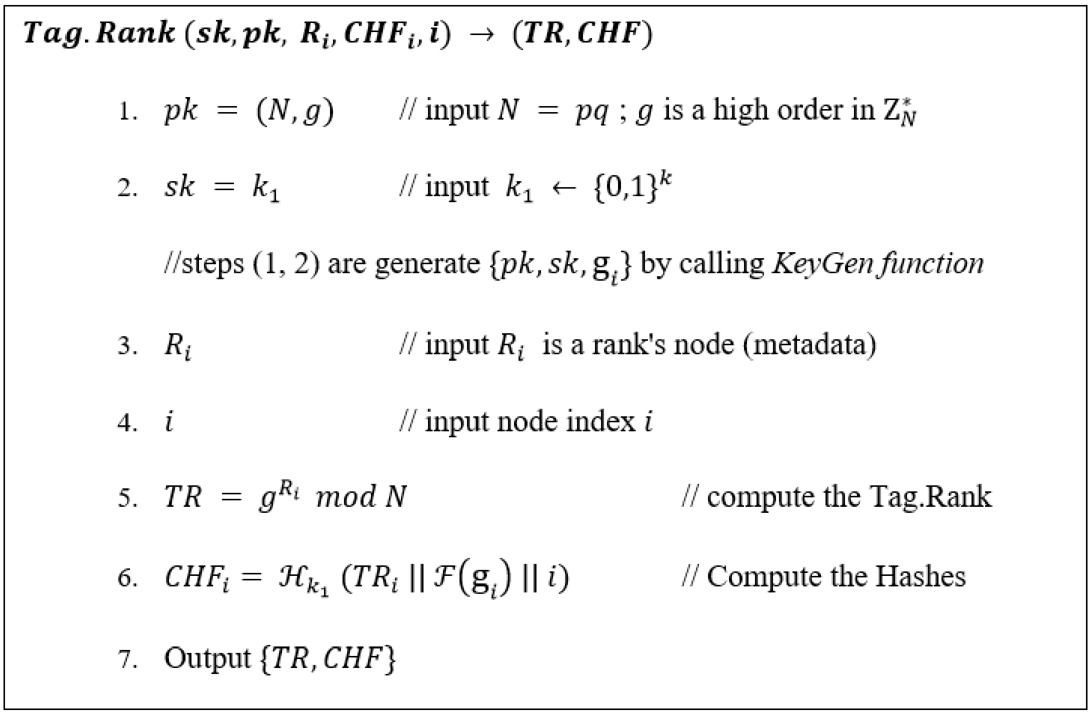


Figure 12 Structure of adapted scheme in server side

4.2 Setup the New-Adds

We can notice that there are differences between Figure 11 and Figure 12 in the New-Adds. The New-Adds are classified into two chunks, Tag.Rank and Hashes, in server side. In client side, the New-Adds are classified into three chunks, Tag.Rank, Hashes, and Ꙍ Matrix.

Figure 13, summarizes how to build Tag.Rank (TR_i) and Hashes (CHF_i) for each node of metadata, rank's nodes (R_i) in below-level of Flex-List. In Figure 13, *KeyGen function* produces a public key and a private key $\{sk, pk\}$.



```
$Tag.Rank\ (sk, pk,\ R_i, CHF_i, i) \rightarrow (TR, CHF)$
    1. $pk = (N, g)$        // input $N = pq$ ; $g$ is a high order in $Z_N^*$
    2. $sk = k_1$           // input $k_1 \leftarrow \{0,1\}^k$
       //steps (1, 2) are generate $\{pk, sk, g_i\}$ by calling KeyGen function
    3. $R_i$                // input $R_i$ is a rank's node (metadata)
    4. $i$                  // input node index $i$
    5. $TR = g^{R_i} \ mod\ N$                      // compute the Tag.Rank
    6. $CHF_i = \mathcal{H}_{k_1} (TR_i \,||\, \mathcal{F}(g_i) \,||\, i)$     // Compute the Hashes
    7. Output $\{TR, CHF\}$
```

Figure 13 Setting the Tag.Rank and the Hashes parameters

The Tag.Rank parameter generates a tag for each rank's node (R_i) in below level except for the sentinel's nodes (present in Figures (11 and 12) by k_4 and k_{14}) because of they have zero value. Equation 4.1 calculates the Tag.Rank as in [21].

$$TR = g^{R_i} \ mod\ N \qquad (4.1)$$

Once $N = pq$ "p and q are the product of two large prime numbers used in RSA group Z_N^* , g is a high-order generator".

The Hashes parameter computes the Cryptographic Hash Function (CHF) by Tag.Ranks and ꞬƆ matrix, which meant to increase the integrity of Tag.Rank**.** Equation 4.2 calculates the hashes as in [22].

$$CHF_i = \mathcal{H}_{k_1} (TR_i \,||\, \mathcal{F}(g_i) \,||\, i) \qquad (4.2)$$

Once "$\boldsymbol{\mathcal{H}}$ is a *cryptographic hash function*, $\boldsymbol{\mathcal{F}}$ is a *pseudo-random function*" and $||$ denote concatenation.

During generation the public key and secret key by the Client side, the ꞬƆ Matrix parameter (g_i) is generated too so that guarantee the integrity with initial value zero.

Each g_i is belong to any given rank's node in below level of the Flex-List. However, the Ꞡ matrix, which kept by Client only, is representing how many modification times takes place on the responsible nodes.

The Client sends the handled metadata and the public key upon the Server with saving the Ꞡ Matrix and the secret key confidentially. The handled metadata consists of Flex-List nodes in below-level i^{th}, Tag.Rank parameter, and Hashes matrix.

Figure 14, shown the pseudo-code of setting up the New-Adds in both the Client and Server sides.

Pseudo-code of the New-Adds

```
1. Input FlexList_size
2. Input $g_i = 0$; // by KeyGen( node.index)
3. Generate public key, secret key
4. For(node.index  != 0)
        a. If ( flexlist.node.below_level != nodes.sentinel )
                i. $TR_i$ = CalculateTag( flexlist.node.below_level, len,
                   vecter.data); //By the RSA-group
                ii. $CHF_i$ = CalculateHash( $TR_i$, PseudoRandom($g_i$) , len,
                   vecter.data, node.index); //By the SHA2
                iii. Keep on referring all the pointers
           End If
        b. Else If ( flexlist.node.below_level = sentinels nodes)
                i. node.index + 1
   End For
5. Return $sk$, $g_i$ //Save Ꞡ Matrix and secret key privately by the Client
6. Return $TR_i$ , $CHF_i$ //Send them to the Server with public key
```

Figure 14 Pseudo-Code of the New-Adds

Doing the same steps in Figure 14 by the Server but without generate the Ꮆ Matrix, while the Hashes will be calculated like this code:

CHF_i = *CalculateHash*(TR_i, *PseudoRandom*(g_i) , *len*, *vecter.data*, *node.index*);

4.3 The Upgrade Operations

The main operations that occur between the Client and the Server are the modification, the insertion, and the deletion. We have provided some figures of these operations to illustrate the idea of the adapted scheme. For explaining these operations, we are assuming the Flex-List originally have n nodes and the operations upgrade the data structure upon the New-Adds. Additionally,

Table (1) summarizes the notation have been used in this section, in adapted algorithms especially.

Table 1 Characters consumed in the upgrade algorithms

Symbol	Description
$*$	Insert a parameter
$'$	Upgrade a parameter
"	Multiple upgrades onto the parameter
$\boldsymbol{i}$	The index parameter in bellow-level of Flex-List
$\boldsymbol{k}$, $\boldsymbol{s}$	Random number
$\boldsymbol{\mathcal{F}}$	pseudo-random function
$\boldsymbol{n}$	No. of elements of the metadata.
$\boldsymbol{g}$	High-order
$\boldsymbol{\mathcal{H}}$	Cryptographic Hash Function
g	The Ꮆ Matrix
$\boldsymbol{R}$	Rank of the node
$\boldsymbol{TR}$	The Tag.Rank parameter
$\boldsymbol{CHF}$	The Hashes array

Setup Insertion Operation

Client **Server**

1. $g_{n+1}^{*} = 0$ // inserts at the close of ω Matrix
2. $Tag.Rank\ (TR_i\, , CHF_n)$ //computes TR_i^{*} and CHF_{n+1}^{*}
3. $\{i,\ R_i^{*}, TR_i^{*}, CHF_i^{*}\}$ //transmits to Server

Upload to Server

$\longrightarrow$

1. R_i^{*} //inserts as R_{i+1} next to the R_i node
2. TR_i^{*} //inserts in Tag.Rank group
3. CHF_{n+1}^{*} // appends to Hashes group

Update Server

$\longleftarrow$

1. CHF_R //selects a Hashes
2. $g_s\ =\ g^s$ // computes a random number s
3. $\{\ i+1,\ CHF_R,\ g^s\ \}$ //transmits to Server

- Proof if the Server has the new node R_{i+1}

$\longrightarrow$

1. R_{i+1} ? //yes, Server finds the node in $(i+1)^{th}$ of FlexList
2. $TR_{i+1} \leftrightarrow CHF'_{(i+1)}$

   // corresponds Hashes with Tag.Rank
3. $TR_s\, , CHF_R$ //computes them

   //since $[TR_s\ =\ {g_s}^{R_{i+1}}\ mod\ N]$ and

   // $[CHF\ =\ \mathcal{H}_R\ (TR_{i+1}\ ||\ CHF_{(i+1)}{}'\ )]$

- Verify if the value of $(TR_s\, , CHF)$ are equal to their values in Client

Return $(TR_s\, , CHF)$ to Client

$\longleftarrow$

1. $(TR_i^{*})^{s}\ ? = TR_s$ // computes if equal
2. $\mathcal{H}_R\ (TR_{i+1}\ ||\ CHF_{(i+1)}{}'\ )\ ? =\ CHF$ // computes if equal

Figure 15 Algorithm of Insertion Operation

4.3.1 Insertion Operation

If we want to insert the new R_i^* node into the metadata (next the i^{th} in below-level nodes), we must manage the operation in both of the Client and the Server sides as shown in Figure 15. The Insertion operation is enrolled in both of the Client side and the Server side as the following: Firstly in Client side, we add g_{i+1}^* parameter into the end of GD Matrix while computing the TR_i^* with the CHF_i^* from R_i^*. Moreover, the client transmits $\{i, R_i^*, TR_i^*, CHF_i^*\}$ parameters to the Server.

Secondly in Server side, a R_{i+1} next to the R_i node of the i^{th} is inserted in below-level nodes. While insertion the R_i^* node in Flex-List and TR_i^* in Tag.Rank group, it appends CHF_{n+1}^* into the end of the Hashes group. The pointers, which appear in Figure 16 with the blue arrows, preserve the pointers among the parameters during the whole process. Moreover, the Client will transmit the CHF_R and g^s to the Server (selects CHF for R and random number s by computing g_s) to confirm whether the insertion operation has been done successfully.

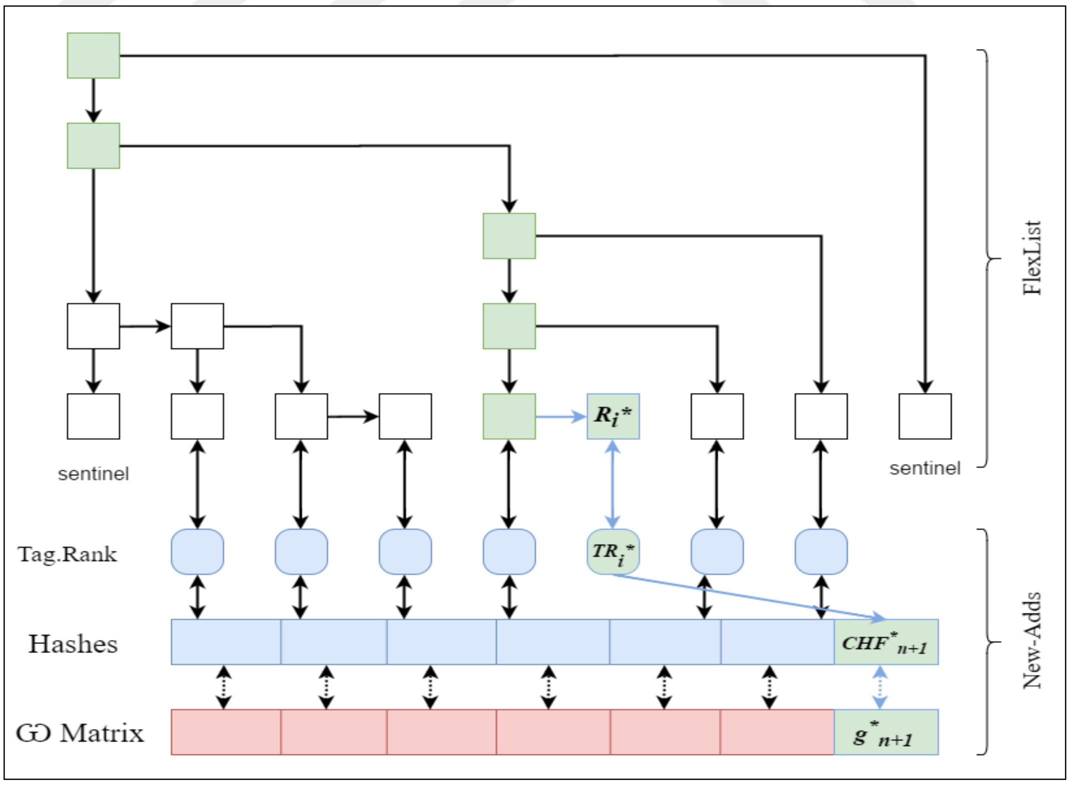


Figure 16 Insertion Operation

```
Setup Modification Operation
            Client                                            Server
  1. Index $i$          //transmits to Server

                         Upload to Server $\longrightarrow$
                              1. $R_i$ , $TR_i$   //searches into the Flex-List
                              2. $CHF_i{}'$ , $i\,'$  // congruent with Tag.Rank
                              3. $\{i\,',\ R_i\,,\ TR_i\,, CHF_i{}'\}$ // transmits to Client

                         Update Server $\longleftarrow$
  1. $TR_i$ ?, $CHF_i{}'$ ?        //computes for verification
  2. $g_i' \rightarrow g_i^{*\prime}$ , $R_i \rightarrow R_i^{*\prime}$ //upgrades the GƆ matrix and the Node
  3. $TR_i^*$ , $CHF_i^{*\prime}$          //computes the Tag.Rank and the Hashes
  4. $\{R_i^*, TR_i^*, CHF_i^{*}{}'\}$     // transmits to Server

                         Upload to Server $\longrightarrow$
                    $R_i \rightarrow R_i^{*\prime}$ , $TR_i \rightarrow TR_i^{*\prime}$ , $CHF_i{}'\ ' \rightarrow CHF_i^{*\prime}$  //upgrades all
                         Update Server $\longleftarrow$
  1. $CHF_R$      //selects a Hashes
  2. $g_s\ =\ g^s$     // computes a random number $s$
  3. $(\ i\ ,\ CHF_R,\ g^s\ )$     //transmits to Server

                   -   Proof if the Server has the $R_{i+1}$ node $\longrightarrow$
                                   1. $R_{i+1}$ ? //finds node in $i^{th}$ of Flex-List
                                   2. $TR_i \leftrightarrow CHF_i{}'$  // corresponds together
                                   3. $TR_s$ , $CHF_R$ // computes them
                                        //since $[TR_s\ =\ g_s{}^{R_i}\ mod\ N]$ and
                                        // $[CHF = \mathcal{H}_R\ (TR_i\ ||\ CHF_i')]$

       -   Verify if the values of $\{TR_s\,, CHF\}$ are equal to their values in Client

                   Return $(TR_s\,, CHF)$ to Client $\longleftarrow$
    1. $(TR_i^*)^s\ ?=\ TR_s$                        // computes if equal
    2. $\mathcal{H}_R\ (TR_{i+1}\ ||\ CHF_{(i+1)}{}'\ )\ ?=\ CHF$        // computes if equal
```

Figure 17 Algorithm of Modification Operation

4.3.2 Modification Operation

The modification operation can be illustrated step by step according to Figure 17. First of all, during the upgrade on the metadata in the i^{th} node, the rank's nodes will change from R_i to R_i^*.

When the Client transmits the index i to the Server, the Server itself will check whole the Flex-List to detect the R_i node at the i^{th} below-level which congruent with TR_i and CHF_i. After that, the Server will return (i, R_i, TR_i, CHF_i) to the Client. The Client confirms the integrity of these parameters through Ꞷ matrix.

If the Client updated the R_i node to R_i^* then there will be another updating on the $\{g_i, TR_i, CHF_i\}$ parameters to $\{g_i^{*'}, TR_i^*, CHF_i^{*'}\}$. Appending to these upgrade, the Client will transmit $\{R_i^*, TR_i^*, CHF_i^{*'}\}$ to the Server. On the other hand, the Server will upgrade the $\{R_i^*, TR_i^*, CHF_i^{*'}\}$ values.

Moreover, the Client will transmit the CHF_R and g^s to the Server (selects CHF for R and random number s by computing g_s) to confirm whether the modification operation has been done successfully.

4.3.3 Deletion Operation

If we want to delete the R_i node from the metadata in the i^{th} below-level nodes, we must manage the operation in both of Client and Server sides as shown in Figure 18.

Firstly, the Client transmits the index i to the Server. Whereas the Server will check whole the Flex-List to detect the R_i node at the i^{th} below-level which congruent with $\{TR_i, CHF'_i, i'\}$ parameters. Hence, the Client will remove the node; the Tag.Rank; and the last value of Hashes CHF_n which congruent with TR'_n.

Secondly, the Server will return (i') only. Then, the Client will upgrade CHF'_i to $CHF_i^{*'}$ and transmit it to the Server. The Server itself upgrade CHF'_i to $CHF_i^{*'}$ for removing CHF_n .

Setup Deletion Operation

<u>**Client**</u> <u>**Server**</u>

1. Index i //transmits to Server

Upload to Server $\longrightarrow$

1. R_i , TR_i //searches into the Flex-List
2. $CHF_i{}'$, i' //congruent with Tag.Rank
3. R_i , TR_i //removes from the Flex-List
4. $\{i', CHF_n, TR_n{}'\}$ //transmits to Client

//since CHF_n is the final value congruent with $TR_n{}'$

Update Flex-List $\longleftarrow$

1. TR_n , CHF_n //congruent between them
2. g_n // appends into the Ꞛ matrix
3. $CHF_n \mathbin{?=} \mathcal{H}_{k1}\,(TR_n{}' \,||\, \mathcal{F}(g_n) \,||\, n\,)$
   // computes for verification since $g_i{}' > g_n$ when $g_i^{*\prime} = g_i{}' + 1$
4. $CHF_n \rightarrow CHF_i{}'$ //changes the pointer
5. $CHF_i{}'$ //transmits to Server

Upload to Server $\longrightarrow$

1. $CHF_i{}' \rightarrow \mathrm{CHF}_i^{*\prime}$ //upgrades the Hashes
2. CHF_n //remove it

Update Server $\longleftarrow$

1. CHF_R //selects a Hashes
2. $g_s = g^s$ // computes a random number s
3. $(\,i'\,,\ CHF_R,\ g^s\,)$ //transmits to Server

Check if the Server has the $CHF_i{}'$ $\longrightarrow$

1. $CHF_i{}'$ //searches Hashes to find i'^{th}
2. $R_i{}'' \leftrightarrow TR_i{}''$ // congruent with $CHF_i{}'$
3. TR_s , CHF_R //computes them

//since $[TR_s = {g_s}^{R_i''} \bmod N]$ and $[CHF = \mathcal{H}_R\,(TR_i{}'' \,||\, CHF_i')]$

Return $(TR_s\,, CHF)$ to Client for verifying $\longleftarrow$

1. $(TR_n')^s \mathbin{?=} TR_s$ // computes if equal
2. $\mathcal{H}_R\,(TR_n{}' \,||\, CHF_i^{*\prime}\,) \mathbin{?=} CHF$ // computes if equal

Figure 18 Algorithm of Deletion Operation

The pointers, which appear in Figure 19 with the blue arrows, preserve the pointers among the parameters during the whole process.

Moreover, from the Client will transmit the CHF_R and g^s to the Server (selects CHF for $_R$ and random number s by computing g_s) to confirm whether the deletion operation has been done successfully.

The following hints are meant to illustrate Figure 19:

- The green nodes are representing the search path.
- The gray nodes are representing the deleted nodes.
- The red arrows are representing the deleted pointers.
- The blue arrows are representing the new pointers.

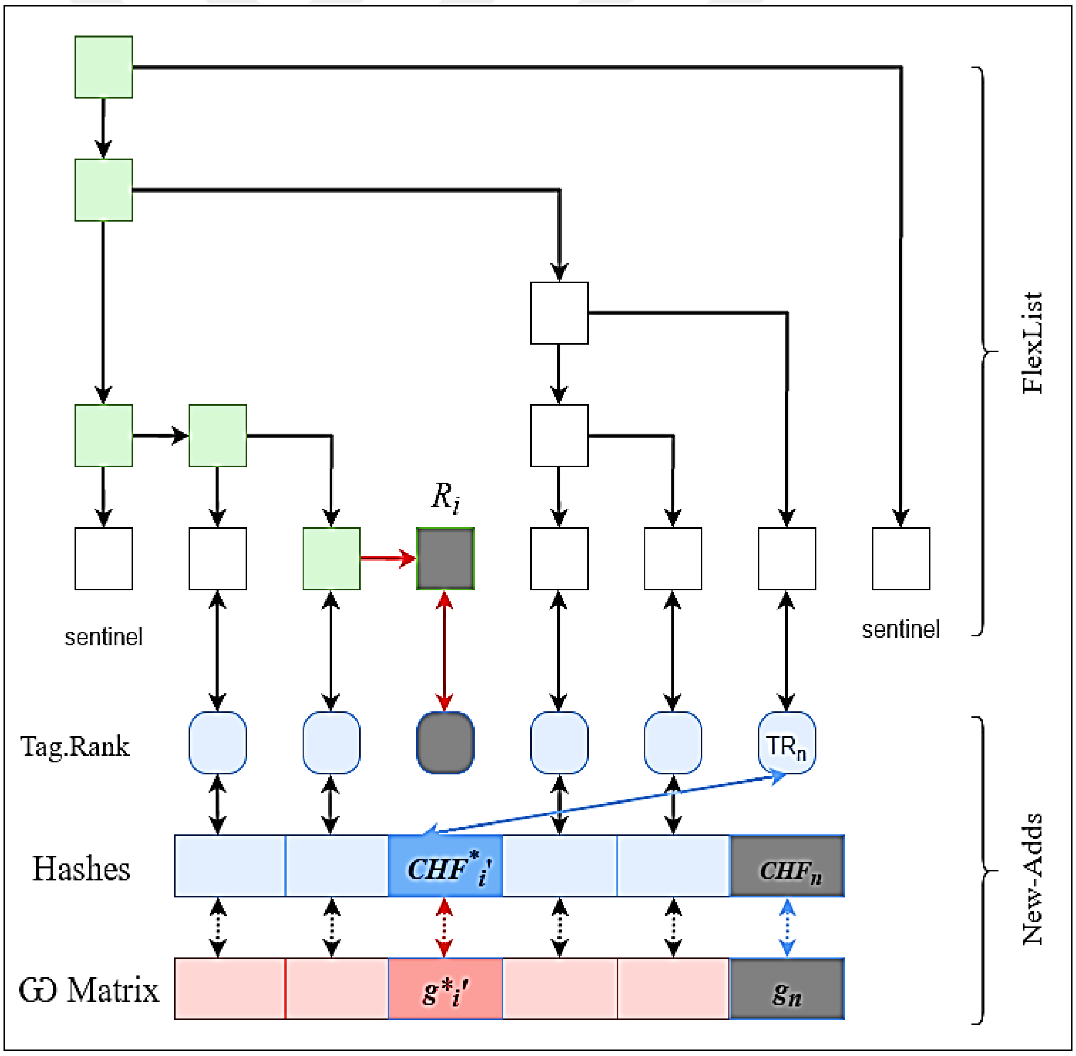


Figure 19 Deletion Operation

4.4 Integrity

We clarify a nowadays issue about the data integrity. The issue is covered by some steps, especially such as *challenges*, *proofs and verifications*. To explain how the Client gets a verification of the integrity to proceed the file onto the Server, we have utilized the Challenge Prove-Verify (CPV) algorithm. Figure 20 demonstrates the CPV algorithm.

Overall, in different issues for decreasing the general expenses of communication, we guide the Server to generate the *index* and the *coefficient* via its side. The Client delivers the permutation and function pseudo-random ($\prod, \mathcal{F}$) to the Server. The client generates the $k2, k3$ numbers randomly (they are extended from $k1 \leftarrow \{0,1\}^k$) while transmitting them to Server for *challenge*. The Server orders a system call (*prove function*) for making and returning the prove ($\wp$).

Moreover, the Client orders a system call (*verify function*) for confirming if $\wp$ was true, then the outcome will be an *Accept* else if it corrupts the file then the outcome will be a *Reject.* Additionally, Table (2) summarizes the notation that we have used.

Table 2 Characters consumed in the CPV algorithms

Symbol	**Description**
$\boldsymbol{C}$	Counts the No. of the node
$\boldsymbol{CG}$	Challenge generates
$\boldsymbol{j}$	Index parameter
$\prod$	Pseudo-random permutation
$\boldsymbol{N}$	No. of the node in i^{th}
$\boldsymbol{K2}$, $\boldsymbol{k3}$	Random numbers extended from $k1 \leftarrow \{0,1\}^k$
$\boldsymbol{\beta}$	The coefficient of the rank's nodes
$\boldsymbol{\alpha}$	Forges coefficient of the rank's nodes
$\boldsymbol{\wp}$	Outcome of *Prove function*
$\boldsymbol{FS}$	File Server set of $\{R, TR, CHF\}$
$\boldsymbol{DPG}$	Data Possession Game
$\boldsymbol{Accept}$	Win Data Possession Game
$\boldsymbol{Reject}$	Lose Data Possession Game

```
Setup CPV algorithm

i. $\boldsymbol{Challenge}\ (\boldsymbol{C} \rightarrow \boldsymbol{CG})$

    1. $R_n \rightarrow C$    //counts the No. of the node in $i^{th}$
    2. $k2\ , k3$    //generates random numbers extended from $k1 \leftarrow \{0,1\}^k$
    3. $i_j = \prod_{k2}(j)$    //gets index $i_j$ for $1 \le j \le C$
    4. $\beta_j = \mathcal{F}_{k3}(j)$    //gets coefficient $\beta_j$ for $1 \le j \le C$
    5. $CG = \{(i_1, \ldots, i_C), (\beta_1, \ldots, \beta_C)\}$   //outcomes

ii. $\boldsymbol{Prove}\ ((\boldsymbol{CG}, \boldsymbol{FS}) \rightarrow \wp)$

    1. $CG\ , FS$    //input challenge and file Server ($FS\ is\ \{R, TR, CHF\}$)
    2. $FS_{i_j}$ ?    //searches $\{R'_{i_j}, TR'_{i_j} and\ CHF_{i_j}\}$ when $i_j \in (i_1, \ldots, i_C)$
    3. $R_C = \beta_1 . R'_{i_1} + \ldots + \beta_C . R'_{i_C}\ , CHF_c = CHF_{i_1} \cdot \ldots \cdot CHF_{i_C}$
       //computes file Server via challenge within the pointers
    4. $\wp = \{R_C\ , CHF_C\ , (TR_{i_1}, \ldots, TR_{i_C})\}$    //outcomes

iii. $\boldsymbol{Verify}\ (\boldsymbol{sk},\ \boldsymbol{CG},\ \wp,\ GD)$

    1. $sk = k1$    //input secret key
    2. $CG = \{(i_1, \ldots, i_C), (\beta_1, \ldots, \beta_C)\}$    //input challenge function
    3. $\wp = \{R_C\ , CHF_C\ , (TR_{i_1}, \ldots, TR_{i_C})\}$    //input prove function
    4. $g_{i_j}$    //input GD matrix and look for $(g_{i_1}, \ldots, g_{i_C})$ via $i_j$
    5. $TR_C\ , CHF_C$    //computes the Tag.Rank and the Hashes
       //since $TR_C = \{TR_{i_1}'^{\beta_1}, \ldots, TR_{i_C}'^{\beta_C}\} = g^{R_C} \bmod N$
       // $CHF_C = \mathcal{H}_{k_1}(TR_{i_1}' \,||\, \mathcal{F}(g_{i_1}) \,||\, i_1) \cdot \ldots \cdot \mathcal{H}_{k_1}(TR_{i_C}' \,||\, \mathcal{F}(g_{i_C}) \,||\, i_C)$
    6. $TR\ ?= TR_C\ \&\&\ CHF\ ?= CHF_C$
       //computes if equal then outcome Accept else outcome Reject
```

Figure 20 CPV algorithm

4.5 Security Analysis

We have built up the security of adapted scheme to be homogeneous with “*Data Possession Game”* (DPG) [7] of the original DPDP scheme. The following scenarios (1, 2) are rolling between Contender and Opponent with a view to resistance and facing the probable attacks.

4.5.1 Scenario1

By holding all the Tag.Rank and Hashes congruent with CPV algorithm:

1. Contender: Generates pk, sk and Ѡ matrix by calling the *KeyGen function* ($KeyGen\ (1^k)\ \rightarrow\ \{pk, sk\}$) then saving the sk and the *Ѡ matrix* secret, and uploads the pk to Server.
2. Opponent: chooses i^{th} nodes R_i (since $0\ \leq i\ \leq n$), they predicts the inquiries with adaptation, then transmits to Contender.
3. Contender: computes TR_i and CHF_i by calling *Tag.Rank* function ($Tag.Rank(pk, sk, R_i, g_i, i)\ \rightarrow\ \{TR_i\ , CHF_i\}$), then transmits back.
4. Opponent: keeps on inquiry about the remainder Tag.Ranks from the Contender, then storages all the parameters (TR_i and congruent CHF_i).
    - ❖ Otherwise: Opponent either upgrade $R_i\ to\ R_i^*$ or modify only the allocates of i to i^* , then upgrades the TR_i^* and CHF_i^* by transmitting them to Contender.
5. Contender: Generates $Challenge\ (C\ \rightarrow\ CG)$ by calling CPV algorithm and asking Opponent for providing $Prove\ ((CG, FS)\ \rightarrow\ Ꝓ)$ to prove the possession of rank's nodes in i^{th} ($R_{i_1}, \ldots, R_{i_C}$), which determines by CPV algorithm.
6. Opponent: computes $Prove\ ((CG, FS)\ \rightarrow\ Ꝓ)$ to prove the possession Ꝓ for Tag.Ranks, which determines by CPV algorithm and return Ꝓ.
7. Contender: checks whether Opponent is winning the DPG by calling CPV algorithm when getting ($Verify\ (sk, CG,$ Ꝓ, Ѡ$)\ =\ Accept$) on not when getting ($Verify\ (sk, CG,$ Ꝓ, Ѡ$)\ =\ Reject$).

4.5.2 Scenario2

Assume the Opponent wins the DPG by putting a hand on all the Tag.Ranks and Hashes but missing several of the information (rank's nodes). We consider these steps to show how Contender reacts with Opponent in DPG fairly by utilizing cryptographic hash function with create and upgrade the Tag.Ranks.

1. Contender: delivered a cryptographic hash function, N $(N = pq)$ as the modulus and patterns a high-order g as the base.

   Then gets $CG = \{(i_1, \ldots, i_C), (\beta_1, \ldots, \beta_C)\}$ by the CVP algorithm.
2. Opponent: when forges the rank's nodes$(\alpha_{i_j}, \ldots, \alpha_{i_k})$, while miss the original rank's nodes $R_{i_j}, \ldots, R_{i_k}$ $(since\ \{i_j, \ldots, i_n\} \subseteq \{i_1, \ldots, i_C\})$.

   Then compute that such as:

   $R_j = \beta_1 R_{i_1} + \ldots + \beta_j \alpha_{i_j} + \ldots + \beta_n \alpha_{i_n} + \ldots + \beta_c R_{i_c}$

   And $CHF_j = CHF_{i_1} \cdot \ldots \cdot CHF_{i_C}$

   And returns $\wp = \{R_C, CHF_C, (TR_{i_1}, \ldots, TR_{i_C})\}$
3. Contender: makes a system call for *Verify function* $(Verify\ (sk, CG, \wp, \omega))$ to comfier the prove such as:

   $TR_j = g^R \bmod N = g^{\beta_1 R_{i_1} + \ldots + \beta_j \alpha_{i_j} + \ldots + \beta_n \alpha_{i_n} + \ldots + \beta_c R_{i_c}} \bmod N$,

   $TR_c = TR_{i_1}{}^{\beta_1} \cdot \ldots \cdot TR_{i_c}{}^{\beta_c}$,

   $CHF_c = \mathcal{H}_{k_1}(TR_{i_1}{}' \,||\, \mathcal{F}(g_{i_1}) \,||\, i_1) \cdot \ldots \cdot \mathcal{H}_{k_1}(TR_{i_C}{}' \,||\, \mathcal{F}(g_{i_C}) \,||\, i_c)$
4. Opponent: checks the outcome if *accept* (only when $(TR_j = TR_c)$ && $(CHF_j = CHF_c)$)
   - Otherwise: as the Opponent are conserved *TR & CHF* truthfully, we consider $(CHF_j = CHF_c)$ and $(TR_j \mathrel{?=} TR_c)$.

   Since:

   If $(TR_j = TR_c)$ then

   $TR_{i_1}{}^{\beta_1} \cdot \ldots \cdot TR_{i_c}{}^{\beta_c} = g^{\beta_1 R_{i_1} + \ldots + \beta_j \alpha_{i_j} + \ldots + \beta_n \alpha_{i_n} + \ldots + \beta_c R_{i_c}} \bmod N$,

   $TR_{i_z} = g^{R_{i_z}} \bmod N$ (When $i_z \in (i_1, \ldots, i_C)$)

We get:

$$g^{\beta_1 R_{i_1} + \dots + \beta_c R_{i_c}} = g^{\beta_1 R_{i_1} + \dots + \beta_j \alpha_{i_j} + \dots + \beta_n \alpha_{i_n} + \dots + \beta_c R_{i_c}} \, mod \; N$$

5. Contender: may have $\beta \neq \alpha$ while $g^{\beta} = g^{\alpha} \, mod \; N$, when adaptively altering the coefficients $(\beta_1, \dots, \beta_C)$.

Subsequently, consider $\alpha = A \,\&\, \beta = B$ so $(A - B = k\varphi(N))$ and $(A - B)$ can be used to factor N, according to [17, 7]. Overall, Opponent cannot win the DPG except these both cases, "huge integer factorization problem can be solved by the Opponent" and "collision in the collision-resistant can be found in the CHF" [21, 22].

Hence, the Tag.Ranks and Hashes, which created by adapted scheme, efficient for ensuring that the data integrity of adapted scheme is secure.

CHAPTER 5

EVALUATION AND COMPARISON

After using the data structure of the Flex-List and adding the New-Adds parameters (Tag.Rank, Hashes, and ꞇƆ matrix), we need to evaluated the performance of our approach to see the effect of the New-Adds.

C++ programming language have been used to be corresponding with the Flex-list implementation in [18]. The Flex-list needs an assistance from "$Cashlib$" library, for cryptography, and the "$Boost\ Asio$" library, for network programming, to work [19] [20].

We managed the local experiments by using $64bits$ Ubuntu 14.04 ($Trusty\ Tahr$); "Intel (R) core (TM) i3-2348m CPU @ 2.30 GHz" processor (with activating only one core of the 4 processors); $4GB$ of memory (RAM); and $3MB$ CacheSize L3 level Cache.

For creating the security parameters (Tag.Rank and Hashes) and expecting the outcome security of $80bits$, we applied $1024bits$ RSA modulus [21], 80bits random numbers, and SHA-2 Cryptographic Hash Function [22].

5.1 Auxiliary Storage

The auxiliary storage confirms dynamically the Client and the Server. The Client has the private key and the ꞇƆ matrix. Whereas, the Server has the Tag.Rank and the Hashes parameters which they are getting a fixed values (whatever the file size it is being, they get $128B$ for Tag.Rank and $20B$ for Hashes). Hence, the additional storage at the Server will be computed by $TR + CHF * R/2$, as shown in the following example:

We have considered that N of the RSA-Group is 1024 bits, and the raw file is $4GB$, the raw file capable to be divided into blocks of $4KB$, have $1{,}000{,}000$ nodes, each block has a $128B$ Tag.Rank and a $20B$ Hashes. Skip-List has approximately R nodes that equal to the number of blocks (as mentioned in Section 3.1), while the Flex-List has approximately $R/2$ nodes (as mentioned in Section 3.3).

In order to prove that only $74MB$ of memory filled with $TR + CHF$ at the server side, we have made a several calculations such as the following; calculates the nodes by $R/2$ $(1{,}000{,}000/2)$ so we get $500{,}000$ nodes; calculates the amount of bytes for a single node by added up the $TR + CHF$ $(20 + 128)$ so we get $148B$; calculates the total bytes of nodes $(500{,}000 * 148B)$ so we get $(74MB)$. The 74MB is equal to 1.8% of the raw file.

The ꞬƆ matrix has $500{,}000$ items when each item takes about 2B (support, $2^{16} = 65536$ times update). Consequently, the Client will get $1MB$ as an auxiliary storage. The storage value of the ꞬƆ matrix can be calculated as the following $(500{,}000 * 2B)$ so we get $1MB$. The $1MB$ is equal to 0.025% of the raw file.

5.2 Comparisons

At the Server, the adapted scheme is differentiated by 1.8% from the Flex-DPDP scheme, in [14], when it compared in term of the additional storage by using a 4GB raw file, and 0.025% of the raw file at the Client.

However, the important reason behind these increases (The New-Adds parameters) is to give an assistance to decrease the complexity of computational (i.e., the time taken by the client to verify the returned proof by the server) and communication (i.e., the size of the returned proof by the server to the client). Likewise, this capacity of the auxiliary storage is acceptable with any framework. In the other hand, presenting the challenge and the upgrade on the adapted system is to provide a comparison among the adapted scheme and the other DPDP schemes.

5.2.1 Challenge

We created a challenge scenario, as a probabilistic system that run through the Client-side and transmitted towards the Server/cloud, as the following steps:

1. The Client chooses the nodes, which are essential to challenge, and creates the indexes and coefficients.
2. The Server looks for the Hashes parameter to detect the challenge nodes, Tag.Ranks and Hashes, then computes a prove $\mathcal{p}$.
3. The Client utilizes the Ꮗ matrix and the private key to confirm the proof via *Verify function*.
   - ❖ The complexity of the (1, 2 and 3) steps is constant.
4. The Client recalls the *Challenge function* for getting the inquiry of 40 Bytes to $k2$ and $k3$ ($k2$ =20B, $k3 = 20B$).
5. The Client determines the inquiry of 40B with the size of $\mathcal{p}$ by recalling the *Prove function* which be determined by the number of the checked rank's nodes.
6. The Client will get fixed value, around $(60KB - 64KB)$, every time of calling the *Prove function*.

We compared these result with the static PDP scheme [3] "when checking 460 blocks each time can discovery this pollution at a probability of 99%, when 1% of the file blocks are polluted". The Big-O notation of the adapted scheme for Challenge and proof is equal to $O(1)$.

When comparing with the dynamic DPDP scheme which has accessing the paths of checked blocks with proofing size equal to $449KB$, the Big-O notation of the adapted scheme for Challenge and proof is equal to $O(1)$.

However, the complexity result of both the Challenge and the proof in the recent schemes (Flex-DPDP [14] and DPDP [9]) is $O(logn)$.

5.2.2 Upgrade

To illustrate the upgrade operation, the *Modification Operation* will take a place at the Client side (as mentioned in section 4.4.2).

Firstly, the Client needs to update the $\{i,\ R_i^*,\ TR_i^*,\ CHF_i^*\}$ parameters for upgrading operation. The size of the parameters is equal to $4152B$. In order to verify the upgrade operation, the client will recall the CPV algorithm for $\{i', g^s, CHF_k\}$ inquiry parameters. So far, the Client gets the outcome, which is equal to $152B$, and then gets the calculation, which is totally equal to $4304B$. Comprehensively, the computational and communication complexity at the Client side is a constant (i.e., the Big-O notation is $O(1)$).

Secondly, the Server will take the segmentations of the $\{R_i\ ,\ TR_i\ ,\ CHF_i'\ ,\ i\,'\}$ parameters. The size of the parameters is equal to $4152B$ as well. The size of the $\{TR_s\ , CHF\}$ parameters that returned, from the upgrade verification by calling the CPV algorithm, is equal to 128B and 20B respectively. So far, the Server gets the outcome, which is equal to $148B$, and then gets the calculation, which is totally equal to $4300B$. Comprehensively, the Server communication complexity is a constant (i.e., the Big-O notation is $O(1)$). While, the computational complexity is equal to $logn$ (i.e., the Big-O notation is $O(logn)$) because the needing to search the whole Flex-List to detect the specific upgraded node.

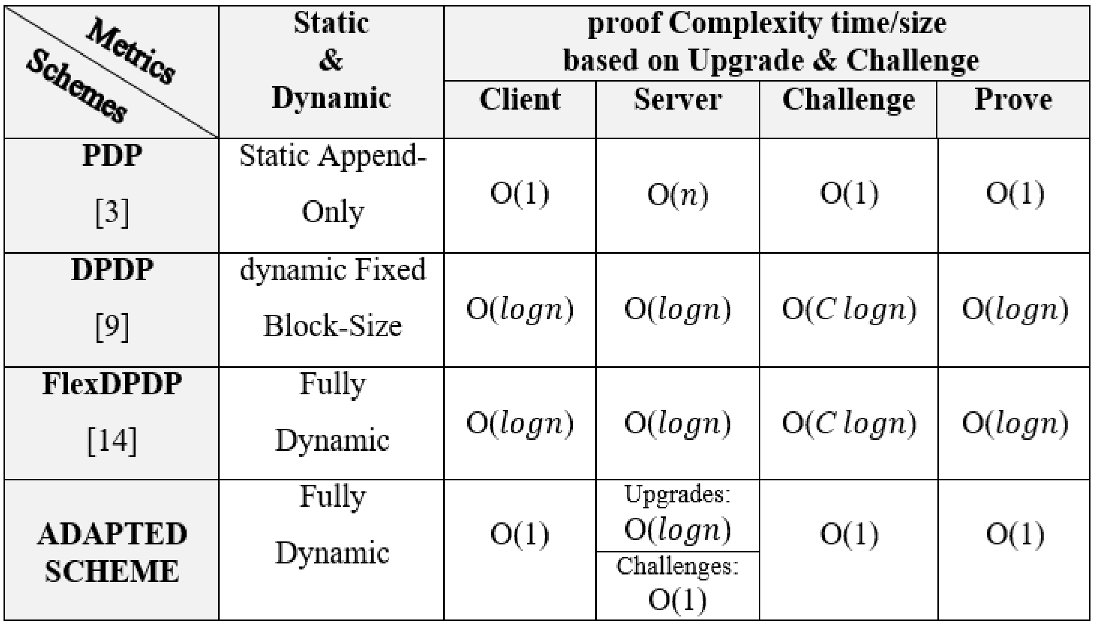
To ensure the integrity of the file, we compared the DPDP scheme, which utilizes the tag of the RBASL, and the Flex-DPDP scheme, which utilizes the tag of the Flex-List, with the adapted scheme.

The communication complexity of the file is equal to $logn$ in both (DPDP and Flex-DPDP) schemes because of the urgent to access the whole path of the specific node that will be returned from the upgrade.

5.3 Results

Table 3, illustrates the results of applying the adapted scheme and gathering the challenges and the upgrades results in Section (5.1). Obviously, the performance of the adapted scheme is acceptable according to the enhancements in the complexity.

Table 3 Comparison between PDP schemes (C the numbers of challenge block)



Metrics / Schemes	Static & Dynamic	proof Complexity time/size based on Upgrade & Challenge			
		Client	Server	Challenge	Prove
PDP [3]	Static Append-Only	O(1)	O(n)	O(1)	O(1)
DPDP [9]	dynamic Fixed Block-Size	O($logn$)	O($logn$)	O($C\ logn$)	O($logn$)
FlexDPDP [14]	Fully Dynamic	O($logn$)	O($logn$)	O($C\ logn$)	O($logn$)
ADAPTED SCHEME	Fully Dynamic	O(1)	Upgrades: O($logn$) / Challenges: O(1)	O(1)	O(1)

In Table 4, we demonstrate the result of comparison between the adapted scheme and the FlexDPDP scheme after ten iterations on 106 size of the Flex-List file. Hence, we manage these steps to explain the Table 4:

- Leads the inquiries (Multi-queries) over 1s and we get about 100 times more queries than the recent scheme.
- Manages the single proof and verifies have taken within the adapted scheme around 2ms, which it is reducing the time for the authentication and the verification.
- Manages the multiple proofs and verifies have taken within the adapted scheme around 170ms, which it is reducing the time for the authentications and the verifications at the Client.

- Manages the multiple proofs and verifies have taken within the adapted scheme around 25ms, which it is reducing the time for the authentications and the verifications at the Server.

Table 4 Result of comparison between the adapted and the FlexDPDP schemes

SCHEMES	FlexDPDP	Adapted
Multi-queries	300/1000ms	400/1000ms
Single Proof & Verify	5ms	2ms
At Client Multiple Proof & Verify	649.11ms	170ms
At Server Multiple Proof & Verify	38.60ms	25ms

Particularly, the adapted scheme is appropriate with the situations that deals with time of challenges rather than upgrades.

CHAPTER 6

CONCLUSION

We have taken some steps for alleviating fears when the data storage over outsourcing, and guarantees the integrity of the files into cloud computing dynamically, and to alleviate the computational and communication complexity during the upgrade operations (Insertion, Modification and Deletion operations) and challenges (by used CPV algorithm). Particularly, the adapted scheme is appropriate with the situations that deals with time of challenges rather than upgrades. The performance of the adapted scheme is acceptable according to the enhancements in the complexity at the Client and the Server sides.

In spite of the fact that the New-Adds expended the auxiliary storage by the Client-side, the Client stores approximately 0.025% of the 4GB file Size. Generality, that amount is agreeable during many situations. The adapted scheme is a fully dynamic model. Regard to the remarkable enhancement in the adapted scheme, cutting-complexity of communication from $O(log\, n)$ to $O(1)$ at both the Client and the Server sides. Cutting-complexity of computations (in Challenges) from $O(log\, n)$ to $O(1)$ at both sides in the adapted scheme.

6.1 Future Work

We plan on extending the adapted scheme by implementing the parallel build of the Flex-List with the New-Adds of the adapted scheme. In order to achieve the distribution of the DPDP on multi-CSP without using the third-party and get the replication of the data.

REFERENCES

[1] **Mell, P. and Grance, T., (2009).** *"Effectively and securely using the cloud computing paradigm"*. NIST, Information Technology Laboratory, pp.304-311.

[2] **Landis, C., and Blacharski, D. (2013).** *"Cloud Computing Made Easy"*. Vitual Global, Incorporated.

[3] **Ateniese G, Burns R, Curtmola R, Herring J, Kissner L, Peterson Z, and Song D., (2007).** *"Provable data possession at untrusted stores"*. In Proceedings of the 14th ACM conference on Computer and communications security, Oct 28, pp. 598-609.

[4] **Ateniese, G., Di Pietro, R., Mancini, L.V. and Tsudik, G., (2008).** *"Scalable and efficient provable data possession"*. In Proceedings of the 4th ACM international conference on Security and privacy in communication networks, September, p. 9.

[5] **Ateniese, G., Kamara, S. and Katz, J., (2009).** *"Proofs of storage from homomorphic identification protocols"*. In International Conference on the Theory and Application of Cryptology and Information Security, Springer Berlin Heidelberg, December, pp. 319-333.

[6] **Juels, A. and Kaliski Jr, B.S., (2007).** *"PORs: Proofs of retrievability for large files"*. In Proceedings of the 14th ACM conference on Computer and communications security, October, pp. 584-597.

[7] **Erway, C.C., Küpçü, A., Papamanthou, C. and Tamassia, R., (2015).** *"Dynamic provable data possession"*. ACM Transactions on Information and System Security (TISSEC), 17(4), p.15.

[8] **Deswarte, Y., Quisquater, J.J. and Saïdane, A., (2004).** *"Remote integrity checking"*. In Integrity and internal control in information systems, VI Springer US., pp. 1-11.

[9] **Gazzoni Filho, D.L. and Barreto, P.S.L.M., (2006).** *"Demonstrating data possession and uncheatable data transfer"*. IACR Cryptology ePrint Archive, p.150.

[10] **Sebé, F., Domingo-Ferrer, J., Martinez-Balleste, A., Deswarte, Y. and Quisquater, J.J., (2008).** *"Efficient remote data possession checking in critical information infrastructures"*. IEEE Transactions on Knowledge and Data Engineering, 20(8), pp.1034-1038.

[11] **Hao, Z., Zhong, S. and Yu, N., (2011)**. *"A privacy-preserving remote data integrity checking protocol with data dynamics and public verifiability"*. IEEE transactions on Knowledge and Data Engineering, 23(9), pp.1432-1437.

[12] **Liu, F., Gu, D. and Lu, H., (2011)**. *"An improved dynamic provable data possession model"*. In IEEE International Conference on Cloud Computing and Intelligence Systems, September, pp. 290-295.

[13] **Esiner, E., Küpçü, A. and Özkasap, Ö., (2014)**. *"Analysis and optimization on FlexDPDP: A practical solution for dynamic provable data possession"*. In International Conference on Intelligent Cloud Computing, February, Springer International Publishing, pp. 65-83.

[14] **Esiner, E., Kachkeev, A., Küpçü, A. and Özkasap, Ö., (2013)**. *"Flexlist: optimized skip list for secure cloud storage"*. Technical Report, Koç University.

[15] **Pugh, W., (1990)**. *"Skip lists: a probabilistic alternative to balanced trees"*. Communications of the ACM, 33(6), pp.668-676.

[16] **Goodrich, M.T., Tamassia, R. and Schwerin, A., (2001)**. *"Implementation of an authenticated dictionary with skip lists and commutative hashing"*. In DARPA Information Survivability Conference &amp; Exposition II. DISCEX'01. Proceedings, IEEE, Vol. 2, pp. 68-82.

[17] **Miller, G.L., (1975)**. *"Riemann's hypothesis and tests for primality"*. In Proceedings of seventh annual ACM symposium on Theory of computing, May, pp. 234-239.

[18] **Cryptography, Security, and Privacy Research Group, (FlexDPDP).** https://crypto.ku.edu.tr/downloads.

[19] **Brownie, cashlib, cryptographic library.** http://github.com/brownie/cashlib.

[20] **Boost – asio, library.** http://www.boost.org/doc/libs.

[21] **Papamanthou, C., Tamassia, R. and Triandopoulos, N., (2008)**. *"Authenticated hash tables"*. In Proceedings of the 15th ACM conference on Computer and communications security, October, pp. 437-448.

[22] **Dobraunig, C., Eichlseder, M. and Mendel, F., (2014)**. *"Analysis of SHA-512/224 and SHA-512/256"*. In International Conference on the Theory and Application of Cryptology and Information Security, Springer Berlin Heidelberg, December, pp. 612-630.