Secure Query on Encrypted Data by Using Fully Homomorphic Encryption

A Dissertation Submitted

for Fulfilment of the Requirements

for the Degree of

MASTER OF TECHNOLOGY

in

Cryptology & Security

By Mrinmoy Bera
Roll - CrS2312

to

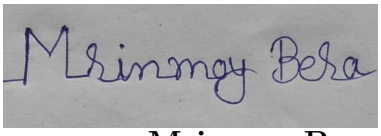
CRPTOGRAPHY & SECURITY RESEARCH UNIT
INDIAN STATISTICAL INSTITUTE
KOLKATA - 700 108, INDIA

DECLARATION

I, **Mrinmoy Bera (Roll No: CrS2312)**, hereby declare that, this report entitled **"Secure Query on Encrypted Data by Using Fully Homomorphic Encryption (FHE)"** submitted to Indian Statistical Institute, Kolkata towards the fulfilment of the requirements for the degree of **Master of Technology in CSRU**, is an original work carried out by me under the supervision of **Captain Manish Khanna, Lt. Cdr. Keval Krishan and Dr. Mriganka Mandal** and has not formed the basis for the award of any degree or diploma, in this or any other institution or university. I have sincerely tried to uphold academic ethics and honesty. Whenever a piece of external information or statement or result is used then, that has been duly acknowledged and cited.

Kolkata - 700 108

July 2025



Mrinmoy Bera

Certificate

This is to certify that the work contained in this project report entitled **"Secure Query on Encrypted Data By Using FHE"** submitted by **Mrinmoy Bera (Roll No. CrS2312)** to the Indian Statistical Institute, Kolkata towards the fulfilment of the requirements for the degree of **Master of Technology in CSRU** has been carried out by him under my supervision and that it has not been submitted elsewhere for the award of any degree.

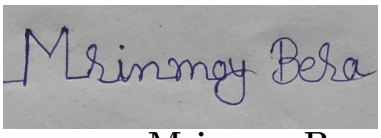
Dr. Mriganka Mandal
Assistant Professor
Cryptology and Security Research Unit
Indian Statistical Institute
Kolkata-700108,INDIA

ACKNOWLEDGEMENT

I thank everyone who has assisted me in seeing this project through to its completion. I would like to first express my profound gratitude and deepest regards to my Guides, WESEE and ISI Kolkata, and sincerely wish to acknowledge their vision, guidance, valuable feedback and constant support throughout this project. I am indebted to my friends for their steadfast encouragement and time. I am lastly grateful to the Indian Statistical Institute, Kolkata for providing the necessary resources and facilities to complete this project to the best of my ability.

Kolkata - 700 108

July 2025



Mrinmoy Bera

Abstract

Cloud service providers typically store user data in an encrypted form (**data at rest**). However, when a user performs a query, the server first **decrypts** the data, processes the query on plaintext, and then sends the result back to the user (**data in transit**). This process exposes a critical vulnerability—if the cloud server is ever **compromised**, the decrypted data becomes accessible to the attacker. To address this security gap, we design a **secure query protocol** that eliminates the need to decrypt data on the server side.

Fully Homomorphic Encryption (FHE) offers a groundbreaking solution by enabling **arbitrary computations** directly on **encrypted data**. This means that even complex queries can be performed without ever revealing the underlying plaintext, and the resulting ciphertext can be **decrypted** by the user to obtain the correct result.

This thesis presents a comprehensive **framework** for executing **secure queries** on encrypted datasets using FHE. We introduce **protocols** for encrypting and decrypting both individual **integers** and structured **CSV files**, ensuring end-to-end **data confidentiality** during storage and processing. Additionally, we implement various operations on encrypted data, including **arithmetic operations** (addition, multiplication), **comparison operations** (equal, not equal, greater than, less than, etc.), and data operations such as **minimum**, **maximum**, and **sorting**.

Our approach enables secure, **database-style querying** over encrypted data in **untrusted cloud environments**, making it highly relevant for **privacy-preserving cloud computing**. We provide detailed **benchmarks** to evaluate the performance and accuracy of our operations, offering insights into the **practicality** and **limitations** of current FHE schemes. The results demonstrate the **feasibility** of secure cloud-based data analytics through homomorphic encryption.

KeyWords : Cloud Computing, Data Base, Data at rest, FHE, sql-query, minimum, maximum, sorting.

Contents

Notations

1. $\mathcal{R}_{q,N} = \mathbb{Z}_q[X]/(X^N + 1)$ denotes a polynomial ring modulo $X^N + 1$ with coefficients in $\mathbb{Z}_q$.
2. $\mathcal{U}(\mathcal{R}_{q,N})$ denotes the uniform distribution over $\mathcal{R}_{q,N}$.
3. $\mathcal{D}$ is the secret distribution.
4. χ denotes the error(noise) distribution.
5. Δ denotes the scaling factor.
6. **Comp_mat** denotes the comparison matrix.
7. $\approx$ denotes approximation.
8. `uint8` denotes an 8-bit unsigned integer.
9. **ms** stands for milliseconds.
10. `Min` refers to the minimum operation.
11. `min` stands for minute.
12. **H_wt(i)** denotes Hamming Weight of j-th column.
13. **LT** denotes Less than function on plaintext.
14. `Less` denotes less than function on Cipher text.
15. **GT** denotes Greater than function on plaintext.
16. `Greater` denotes the greater than function on Cipher text.

Chapter 1

Introduction

The paper [**1 Estore**] introduced a user-friendly encrypted storage scheme named **Estore** which is based on Hadoop distributed file system. They designed and implemented six sets of protocols to meet users' requirements for security and use. EStore manages users and their keys through **registration** and **authentication**, and developed a searchable encryption module and encryption/decryption module to support ciphertext retrieval and secure data outsourcing.[**2**] In this paper, using **hybrid encryption** for store data-in-rest by using **AES** with **Twofish** . [**3**] This paper proposed a non-circuit based homomorphic encryption but in this paper not getting any bootstrapping technique so not perform multiple operation on ciphertext. [**4**] This paper presents a practical and secure **Fully homomorphic order-preserving encryption (FHOPE)** scheme, which allows cloud server to perform complex SQL queries that contain diferent operators (such as addition, multiplication, order comparison, and equality checks) over encrypted data without repeated encryption. But this algorithm is **not CPA-secure**.

In our proposed approach, **Boolean** and **Integer** values are encrypted using the **TFHE** (Fast Fully Homomorphic Encryption over the Torus) algorithm. For floating-point numbers, we apply a scaling technique to convert them into integers before encryption with TFHE. We then implement user-defined functions (UDFs) to compute operations such as `min`, `max`, `sort` (both ascending and descending), and `filter` on encrypted columns. Finally, these encrypted queries are executed on data stored in PostgreSQL, enabling secure and privacy-preserving data analysis.

1.1 Fully Homomorphic Encryption

A Fully Homomorphic Encryption (FHE) scheme is defined by a tuple of four algorithms:

$$(\texttt{KeyGen}, \texttt{Enc}, \texttt{Dec}, \texttt{Eval})$$

Key Generation

- **Public-key FHE:** $\texttt{KeyGen}(1^\lambda) \to (pk, sk, \texttt{evk})$
  where pk is the public key, sk is the secret key, and `evk` is the evaluation key.

- **Symmetric-key FHE:** $\texttt{KeyGen}(1^\lambda) \to (k, \texttt{evk})$
  where k is the symmetric secret key.

Encryption

$$\texttt{Enc}(key, m) \to c$$

- For public-key encryption: $key = pk$
- For symmetric-key encryption: $key = k$

where m is the plaintext and c is the ciphertext.

Decryption

$$\texttt{Dec}(key, c) \to m$$

- For public-key encryption: $key = sk$
- For symmetric-key encryption: $key = k$

Evaluation

$$\texttt{Eval}(f, c_1, c_2, \ldots, c_n, \texttt{evk}) \to c_f$$

Here, f is a function (e.g., Boolean or arithmetic circuit) and c_f is a ciphertext such that:

$$\texttt{Dec}(key, c_f) = f(m_1, m_2, \ldots, m_n)$$

1.2 Cloud Computing and the Role of FHE in Data Security

Cloud computing platforms are extensively used for hosting scalable and efficient data management systems, such as relational databases and distributed storage frameworks. Examples include:

- **SQL** and **PostgreSQL**: For structured data storage and complex query execution.
- **Hadoop Distributed File System (HDFS)**: For distributed storage and processing of large-scale unstructured data.

These systems offer advantages such as scalability, cost-effectiveness, and high availability. However, outsourcing data to third-party cloud infrastructure introduces significant **security and privacy concerns**, including:

- Loss of control over sensitive data,
- Risk of unauthorized access by cloud providers or external attackers,

- Challenges in meeting regulatory requirements (e.g., GDPR, HIPAA).

While traditional encryption methods secure data *at rest* and *in transit*, they fail to protect data *during computation*. This is where **Fully Homomorphic Encryption (FHE)** plays a critical role.

FHE enables computation directly on encrypted data. Specifically, it allows cloud servers to perform operations such as:

- Encrypted query execution (e.g., `SELECT`, `WHERE`, `MIN`/`MAX`),
- Aggregation and sorting over ciphertexts,
- Secure evaluation of user-defined functions.

Since FHE does not require decryption at any point, it ensures **end-to-end data confidentiality**, even in untrusted cloud environments. By integrating FHE with systems like PostgreSQL or Hadoop, it is possible to achieve **privacy-preserving cloud computing**, where data remains secure throughout its entire lifecycle. .

1.3 Problem Statement

In modern cloud-based database systems, **PostgreSQL** is extensively used for managing and querying structured data due to its extensibility, performance, and support for user-defined functions. However, outsourcing data to third-party cloud infrastructure introduces significant **privacy and security concerns**, particularly when sensitive data is involved.

This project addresses the problem of executing SQL-like queries over encrypted data stored in PostgreSQL **without revealing any sensitive information** to the cloud server.

1.4 Our Approach

1. **Key Generation:** The client generates a pair of cryptographic keys — a *client key* and a *server key*. The client sends the *server key* to the server for evaluation purposes.
2. **Data Encryption:** The client encrypts structured data (e.g., salary, age) using the *client key*.
3. **Data Upload:** The client uploads the encrypted data to a secure database.
4. **User Query:** A user sends a query request to the server (e.g., "What is the average salary?").
5. **Custom Query Formation:** The server generates a custom query that can operate on the encrypted data based on the user's request.
6. **Database Query:** The server sends the custom query to the database to retrieve relevant encrypted data.

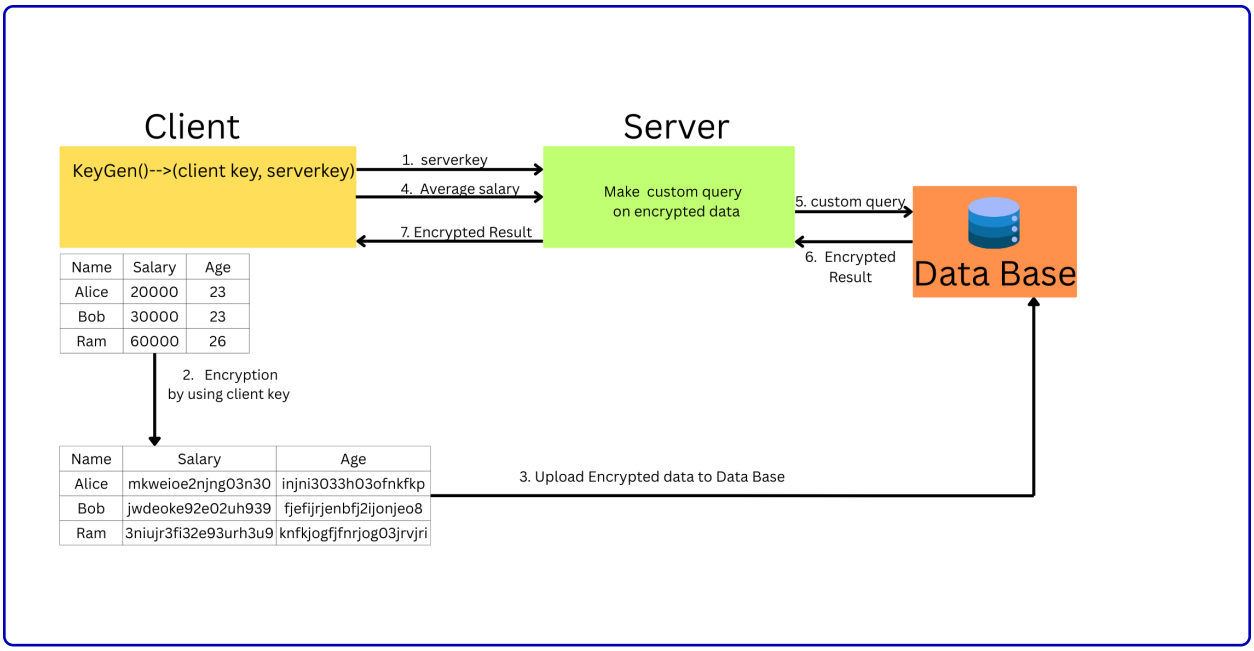


Figure 1.1: Encrypted Query Workflow with Stylish Border

7. **Encrypted Result Retrieval:** The server receives the encrypted result set from the database.

8. **Result Decryption:** The server forwards the encrypted result to the client. The client decrypts the result using the *client key* to obtain the final output.

1.5 Organization of Dissertation

This dissertation is organized into five main chapters, each addressing a specific aspect of performing secure queries over encrypted data using Fully Homomorphic Encryption (FHE).

- **Chapter 2: Encryption and Decryption Protocol**
  We describe the encryption and decryption algorithms used for various data types, including integers, floating-point numbers, and structured data. The chapter also discusses the underlying hard lattice problems that provide the foundation for security in FHE.

- **Chapter 3: Arithmetic Operation on Encrypted Data**
  This chapter explains how basic arithmetic operations such as addition, subtraction, and multiplication can be performed on encrypted data without decryption, preserving data confidentiality during computation.

- **Chapter 4: Operation on Encrypted Data**
  We present algorithms to perform more complex operations such as comparisons, finding minimum/maximum values, sorting, and executing SQL-like queries (`AVG`, `WHERE`,

`ORDER BY`, etc.) on encrypted data. Special attention is given to both integer and floating-point data types, leveraging CKKS for approximate arithmetic.

- **Chapter 5: Benchmarks**
  The final chapter presents experimental results and performance benchmarks for various query operations on encrypted datasets. The results highlight the trade-offs between performance, precision, and scalability for both integer and floating-point encrypted queries.

Each chapter builds upon the previous one to gradually develop a comprehensive framework for secure query processing on encrypted databases using FHE.

Chapter 2

Encryption and Decryption Protocol

For encrypting Boolean and integer values, we are using the **TFHE** scheme. The security of the TFHE scheme is based on a family of hard problems, namely the Learning With Errors (LWE) problem. These problems are commonly used as foundations for building Fully Homomorphic Encryption (FHE) schemes. The LWE problem has two flavours: a search problem and a decision problem.

2.1 Hard Lattice Problems

Definition 1 (Learning With Errors (LWE)) *Let $n \in \mathbb{N}$ be a positive integer, which we call the LWE dimension. Let $q \in \mathbb{N}$ be a positive integer, which we call the ciphertext modulus. Let $\mathbf{s} = (s_0, \cdots, s_{n-1}) \in \mathbb{Z}_q^n$ be a vector, which we call the secret, where for each $0 \leq i < n$, s_i is sampled from some distribution $\mathcal{D}$. We call $\mathcal{D}$ the secret distribution. Further, let χ be another distribution which we call the error distribution. We define a sample from the learning with errors distribution $LWE_{q,n,\chi,\mathcal{D}}$ to be of the form*

$$(\mathbf{a}, b = \sum_{i=0}^{n-1} a_i \cdot s_i + e) \in \mathbb{Z}_q^n \times \mathbb{Z}_q,$$

where $\mathbf{a} = (a_0, \ldots, a_{n-1}) \leftarrow \mathcal{U}(\mathbb{Z}_q)^n$, meaning that each a_i is sampled uniformly and independently from $\mathbb{Z}_q$, and the error or noise $e \in \mathbb{Z}_q$ is sampled from χ.

While the LWE problem is a strong candidate for a hard problem, it results in a significant expansion factor between the unencrypted and encrypted data. When multiple values are to be encrypted, one can decrease this expansion factor by using a generalized version of the LWE problem which uses more complex mathematical structures than just integers.

Below, we define the General Learning With Errors (GLWE) problem (also known as the Module Learning With Errors problem) on the ring $\mathfrak{R}_{q,N}$ defined as

$$\mathfrak{R}_{q,N} = \mathbb{Z}_q[X]/\langle X^N + 1\rangle$$

with N a power of two. Elements of this ring can be written as polynomials of at most degree $N-1$ having integer coefficients. Addition and multiplication follow standard polynomial

operations modulo q. However, after multiplication, the result is reduced to a degree at most $N-1$ by taking the remainder modulo the polynomial X^N+1.

Using the relation $X^N=-1$, any term with an exponent greater than N can be reduced accordingly.

Definition 2 (General Learning With Errors (GLWE)) *Let $N \in \mathbb{N}$ be a power of two, which we call the polynomial size. Let $k \in \mathbb{N}$ be a positive integer which we call the GLWE dimension. Let $q \in \mathbb{N}$ be a positive integer which we call the ciphertext modulus. Let $\mathbf{S} = (S_0, \cdots, S_{k-1}) \in \mathcal{R}_{q,N}^k$ be a vector, which we call the secret, where for each $0 \leq i < k$, $S_i = \sum_{j=0}^{N-1} s_{i,j} X^j$ is sampled from some distribution $\mathcal{D}$, We call $\mathcal{D}$ is the secret distribution and let χ be an error(noise) distribution. We define a sample from the general learning with errors distribution $GLWE_{q,N,k,\chi,\mathcal{D}}$ to be of the form*

$$(\mathbf{A}, B = \sum_{i=0}^{n-1} A_i \cdot S_i + E) \in \mathcal{R}_{q,N}^{k+1}$$

where $\mathbf{A} = (A_0, \ldots, A_{n-1}) \leftarrow \mathcal{U}(\mathcal{R}_{q,N})^k$, meaning that each A_i is sampled uniformly and independently from $\mathcal{R}_{q,N}$, and the error or noise $E \in \mathcal{R}_{q,N}$ is such that all the coefficients are sampled from χ.

2.2 Encryption Algorithm for Integer

1. **Polynomial Decomposition:** The integer message is first decomposed and represented as a polynomial in the ring $\mathcal{R}_{p,N}$.
   Example: let's choose $p = 5, \quad N = 4$

   $m = 214 \xrightarrow{Decomposition} x^3 + 3 \cdot x^2 + 2 \cdot x + 4 \in \mathcal{R}_{p,N}$

2. **Plaintext Encoding:** Let $q \in \mathbb{N}$ be the cipher text modulus. Let further $p \in \mathbb{N}$ be the plaintext modulus. Let $M \in \mathcal{R}_{p,N}$ be a message. We defined encoding of M as :

   $$\tilde{M} = Encode(M, p, q) = \lfloor \Delta \cdot M \rceil \in \mathcal{R}_{q,N}$$

   with $\Delta = \frac{q}{p} \in \mathbb{Q}$ referred to as the scaling factor. Here we take M to be a polynomial with coefficients in $[0, p)$, multiply by Δ, then round each coefficient to an integer and finally reduce modulo q.
   Example: let's choose $q = 53$ then

   $x^3 + 3 \cdot x^2 + 2 \cdot x + 4 \in \mathcal{R}_{p,N} \xrightarrow{Encoding(\Delta = 10.6)} 11 \cdot x^3 + 32 \cdot x^2 + 21 \cdot x + 42 \in \mathcal{R}_{q,N}$

3. **Encryption:** Finally, the encoded polynomial is encrypted using the homomorphic encryption mechanism of the TFHE scheme.

Algorithm 1 Encryption: $\texttt{CT} \leftarrow \texttt{Encrypt}(\tilde{M}, \mathbf{S}; \chi_\sigma)$

Require: $\tilde{M} \in \mathcal{R}_{q,N}$ ▷ Encoded message
Require: $\mathbf{S} = (S_0, S_1, \ldots, S_{k-1})$ ▷ GLWE secret key
Require: χ_σ ▷ Noise distribution
Ensure: $\texttt{CT} \in \text{GLWE}_{\mathbf{S}}(\tilde{M})$ ▷ Ciphertext

$E \leftarrow 0$
for $j = 0$ to $N - 1$ **do**
  $\epsilon_j \leftarrow \chi_\sigma$
  $E \leftarrow E + \epsilon_j \cdot X^j$
end for
for $i = 0$ to $k - 1$ **do**
  $A_i \leftarrow \mathcal{U}(\mathcal{R}_{q,N})$ ▷ Uniform sampling
end for
$B \leftarrow \sum_{i=0}^{k-1} A_i \cdot S_i + E + \tilde{M}$
$\texttt{CT} \leftarrow (A_0, A_1, \ldots, A_{k-1}, B)$
return CT

Example: Let us choose $q = 53, \quad p = 5, \quad \Delta = \frac{q}{p} = 10.6$. Let us take $N = 4, \quad k = 2$. Let $A = (A_0, A_1) = (x^3 + 3x^2 + 4x + 5, 7x^3 + 2x^2 + x + 4), \quad E = x^3 + x + 1 \quad and \quad S = (S_0, S_1) = (x^2 + x, x^3 + 1)$. Then $A_0 \cdot S_0 = 7x^3 + 9x^2 + 4x - 4 and A_1 \cdot S_1 = 11x^3 - 5x^2 - x + 3$. Therefore $B = A_0 \cdot S_0 + A_1 \cdot S_1 + \tilde{M} + E = (30x^3 + 36x^2 + 25x + 42)$. So our cipher text is following-

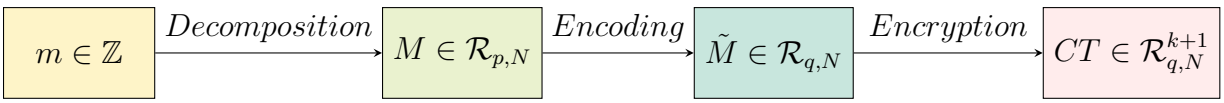
$$CT = (x^3 + 3x^2 + 4x + 5, 7x^3 + 2x^2 + x + 4, 30x^3 + 36x^2 + 25x + 42)$$

$$\tilde{M} = 11 \cdot x^3 + 32 \cdot x^2 + 21 \cdot x + 42 \in \mathcal{R}_{q,N}$$

Encryption

$$CT = (x^3 + 3x^2 + 4x + 5, 7x^3 + 2x^2 + x + 4, 30x^3 + 36x^2 + 25x + 42) \in \mathcal{R}_{q,N}^{k+1}$$

The following diagram illustrates how an integer is encrypted into a ciphertext by using TFHE algorithm -



Note: For an integer $m \geq p^N$, we represent it using its p^N-expression as a tuple:

$$m \to (a_0, a_1, \ldots, a_d)$$

Each component a_i is then encrypted individually to obtain the ciphertext tuple:

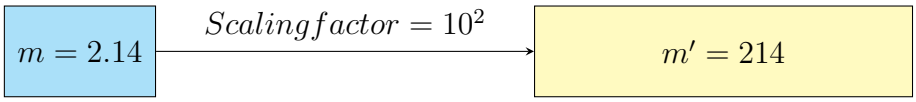
$$(a_0, a_1, \ldots, a_d) \xrightarrow{\text{Encrypt}} (\text{CT}_0, \text{CT}_1, \ldots, \text{CT}_d)$$

2.3 Encryption Algorithm for Floating Value

To encrypt a floating-point value, we follow these two steps:

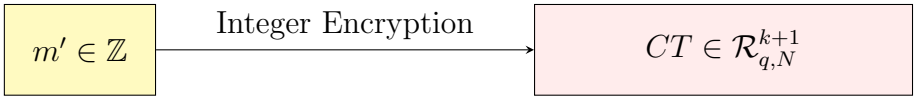
Scaling to Integer

Multiply the floating-point value by a fixed scaling factor to convert it into an integer.**Example:** let us take $m = 2.14$ as plaintext and we take scaling factor is 10^2 then after scaling we get $m' = 214$.



Integer Encryption

Apply the previously defined encryption technique for integer values to encrypt the resulting integer.



2.4 Encryption Protocol for Structure Data

To encrypt **structured data**, we process each **numerical** column individually. For each column, the elements are encrypted one by one, and the encrypted values are then stored under the same column name in a new CSV file. This process is repeated for all numerical columns in the structured dataset.The following is the structured data encryption protocol:

Algorithm 2 Structured Data Encryption

Require: Structured data file `data.csv` and secret_key

Load the structured data from `data.csv`
for each numerical column *col* in the data **do**
  if *col* is of integer type **then**
    for e in *col* **do**
      $enc\_e \leftarrow \text{Int\_Enc}(e)$
    end for
  else if *col* is of float type **then**
    for each element e in *col* **do**
      $enc\_e \leftarrow \text{Float\_Enc}(e)$
    end for
  end if
end for

Example:

Name	Salary	Age
Alice	50000	24.3
Bob	60000	27.3
Robin	75000	24.9

Encryption →

Name	Salary	Age
Alice	nfi39eu2eb3igf0	wobe4r6hn3
Bob	jehijruve23rfftb	gijog6g5jn
Robin	guhr04h5h944b9	nit45jn4ot

2.5 Decryption Algorithm for Integer

Decryption algorithm done by three steeps and the steeps are following -

1. Decryption: We Decrypt the cipher text $CT \in \mathcal{R}_{q,N}^{k+1}$ by using the secret key and get $\tilde{M} \in \mathcal{R}_{q,N}$. We discuss following is Decryption algorithm-

Algorithm 3 $\bar{M} \leftarrow \text{Decrypt}(\text{CT}, \mathbf{S})$

Input: $\mathbf{S} = (S_0, \ldots, S_{k-1})$: the secret key
$\text{CT} = (A_0, \ldots, A_{k-1}, B) \in \mathcal{R}_{q,N}^{k+1}$: the ciphertext
Output: $\bar{M} \in \mathcal{R}_{q,N}$: a noisy version of $\tilde{M}$, called the phase of CT

1: $\bar{M} \leftarrow B - \sum_{i=0}^{k-1} A_i \cdot S_i$
2: **return** $\bar{M}$

Example: Let us take $q = 53, N = 4, k = 2$. And also take secret key $(S_0, S_1) = (x^2+x, x^3+1)$ *and* $CT = (A_0, A_1, B) = (x^3+3x^2+4x+5, 7x^3+2x^2+x+4, 30x^3+37x^2+25x+42)$. So $A_0 \cdot S_0 = 7x^3 + 9x^2 + 4x - 4 \quad A_1 \cdot S_1 = 11x^3 - 5x^2 - x + 3$. Therefore

$$\bar{M} = B - A_0 \cdot S_0 - A_1 \cdot S_1 = 12x^3 + 33x^2 + 22x + 41$$

$$CT = (x^3 + 3x^2 + 4x + 5, 7x^3 + 2x^2 + x + 4, 30x^3 + 36x^2 + 25x + 42) \in \mathcal{R}_{q,N}^{k+1}$$

Decryption ↓

$$\bar{M} = 12 \cdot x^3 + 32 \cdot x^2 + 22 \cdot x + 41 \in \mathcal{R}_{q,N}$$

2. Decoding: To decode $\bar{M} \in R_{q,N}$, We compute the following function:

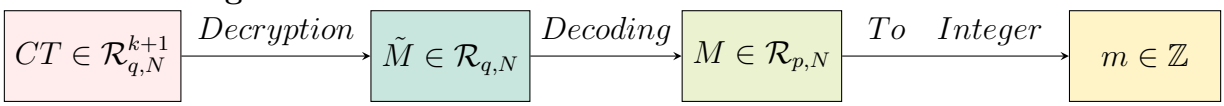
$$M = Decode(\tilde{M}, p, q) = \lfloor \frac{\bar{M}}{\Delta} \rceil$$

Example: Let us take $\bar{M} = 12x^3 + 32x^2 + 22x + 41 \in R_{q,N}$ and p =5, q = 53 then $\Delta = 10.6$. Therefore $M = \lfloor \frac{\bar{M}}{\Delta} \rceil = x^3 + 3x^2 + 2x + 4$

$$12 \cdot x^3 + 32 \cdot x^2 + 22 \cdot x + 41 \in \mathcal{R}_{q,N} \xrightarrow{Decoding(\Delta = 10.6)} x^3 + 3 \cdot x^2 + 2 \cdot x + 4 \in \mathcal{R}_{p,N}$$

3. Convert to Integer: For converting $M \in \mathcal{R}_{p,N}$ to integer by calculating the polynomial at $x = p$.
   Example: Let $M = x^3+3\cdot x^2+2\cdot x+4$ and $x = p = 5$ then $m = 5^3+3\cdot 5^2+2\cdot 5+4 = 214$.

The following diagram shows how the entire decryption process is carried out, from **ciphertext** to **integer**:



2.6 Decryption Algorithm for Floating Value

We perform decryption of floating-point values in the following two steps:

1. Integer Decryption: We decrypt the ciphertext corresponding to the floating-point number using the integer decryption algorithm, which yields the scaled integer value.

2. Scaling to Float: If the original value was scaled by a factor of Δ during encryption, we recover the actual floating-point value by multiplying the decrypted result by $\frac{1}{\Delta}$.

2.7 Decryption Protocol for Structured Data

To decrypt **structured data**, we process each **numerical** column individually. For each column, the encrypted elements are decrypted one by one, and the decrypted values are stored under the same column name in a new CSV file. This process is repeated for all numerical columns in the structured dataset. The following is the structured data decryption protocol:

Algorithm 4 Structured Data Decryption

Require: Encrypted structured data file `encrypted_data.csv` and `secret_key`

Load the encrypted structured data from `encrypted_data.csv`
for each numerical column *col* in the data **do**
    if *col* is of integer type **then**
        for each element *enc_e* in *col* **do**
            $e \leftarrow$ Int_Dec(*enc_e*)
        end for
    else if *col* is of float type **then**
        for each element *enc_e* in *col* **do**
            $e \leftarrow$ Float_Dec(*enc_e*)
        end for
    end if
end for
return `Decrypted.csv` file

Note: During the decryption of floating-point values, we use a scaling technique with a scaling factor denoted by Δ. After performing addition or subtraction on encrypted data, the original scaling factor Δ is still valid during decryption. However, after a multiplication operation, the effective scaling factor becomes Δ^2 due to the multiplication of the encoded values.

2.8 Chapter Conclusion

In this chapter, we explore how basic encryption and decryption algorithms can be applied to structured data. We demonstrate the process of encrypting data in a structured format and retrieving it securely through decryption. This serves as a foundation for the next chapter, where we implement arithmetic operations such as addition, subtraction, and multiplication directly on encrypted data using homomorphic encryption techniques.

Chapter 3

Arithmetic Operation on Encrypted Data

In this chapter, we discuss how arithmetic operations such as addition, subtraction, and multiplication can be performed on encrypted data using homomorphic encryption. We also provide examples to demonstrate the correctness and feasibility of these operations. This chapter lays for more complex secure computations by showing how basic arithmetic can be achieved without decrypting the data, thereby preserving privacy throughout the computation process.

3.1 Addition on Encrypted Data

To add two ciphertexts, we perform polynomial addition since each ciphertext $\mathbf{CT} \in \mathcal{R}_{q,N}^{k+1}$. The ciphertext addition algorithm is as follows:

Algorithm 5 Ciphertext Addition in $\mathcal{R}_{q,N}^{k+1}$

Require: Ciphertexts $\mathbf{CT}_1 = (c_{1,0}, c_{1,1}, \ldots, c_{1,k})$, $\mathbf{CT}_2 = (c_{2,0}, c_{2,1}, \ldots, c_{2,k})$ in $\mathcal{R}_{q,N}^{k+1}$
Ensure: $\mathbf{CT}_{\text{add}} = \mathbf{CT}_1 + \mathbf{CT}_2$ in $\mathcal{R}_{q,N}^{k+1}$
1: **for** $i = 0$ to k **do**
2: $\quad c_{\text{add},i} \leftarrow (c_{1,i} + c_{2,i}) \mod (q, (X^n + 1))$
3: **end for**
4: $\mathbf{CT}_{\text{add}} \leftarrow (c_{\text{add},0}, c_{\text{add},1}, \ldots, c_{\text{add},k})$
5: **return** $\mathbf{CT}_{\text{add}}$

Example: Let us take $q = 53, N = 4, k = 2$. Let us take ciphertexts as

$$CT_1 = (x^3 + 3x^2 + 4x + 5, \quad 7x^3 + 2x^2 + x + 4, \quad 30x^3 + 36x^2 + 25x + 42)$$

$$CT_2 = (2x^3 + 4x^2 + 3x + 5, \quad 6x^3 + x^2 + 2x + 7, \quad 32x^3 + 35x^2 + 26x + 42)$$

Now we calculate:

$$CT_{\text{add}} = (x^3 + 3x^2 + 4x + 5 + 2x^3 + 4x^2 + 3x + 5) \mod (q, X^4 + 1)$$

$$= (3x^3 + 7x^2 + 7x + 10) \mod 53$$

$$7x^3 + 2x^2 + x + 4 + 6x^3 + x^2 + 2x + 7 = 13x^3 + 3x^2 + 3x + 11 \mod 53$$

$$30x^3 + 36x^2 + 25x + 42 + 32x^3 + 35x^2 + 26x + 42 = 62x^3 + 71x^2 + 51x + 84 \mod 53$$

Now reduce modulo 53: $62 \mod 53 = 9$, $71 \mod 53 = 18$, $51 \mod 53 = 51$, $84 \mod 53 = 31$

$$\Rightarrow \boxed{CT_{\text{add}} = (3x^3 + 7x^2 + 7x + 10, \quad 13x^3 + 3x^2 + 3x + 11, \quad 9x^3 + 18x^2 + 51x + 31)}$$

3.2 Subtraction on Encrypted Data

To subtract two ciphertexts, we perform polynomial subtraction since each ciphertext $\mathbf{CT} \in \mathcal{R}_{q,N}^{k+1}$. The ciphertext subtraction algorithm is as follows:

Algorithm 6 Ciphertext Subtraction in $\mathcal{R}_{q,N}^{k+1}$

Require: Ciphertexts $\mathbf{CT}_1 = (c_{1,0}, c_{1,1}, \ldots, c_{1,k})$, $\mathbf{CT}_2 = (c_{2,0}, c_{2,1}, \ldots, c_{2,k})$ in $\mathcal{R}_{q,N}^{k+1}$
Ensure: $\mathbf{CT}_{\text{sub}} = \mathbf{CT}_1 - \mathbf{CT}_2$ in $\mathcal{R}_{q,N}^{k+1}$
1: **for** $i = 0$ to k **do**
2: $\quad c_{\text{sub},i} \leftarrow (c_{1,i} - c_{2,i}) \mod (q, (X^n + 1))$
3: **end for**
4: $\mathbf{CT}_{\text{sub}} \leftarrow (c_{\text{sub},0}, c_{\text{sub},1}, \ldots, c_{\text{sub},k})$
5: **return** $\mathbf{CT}_{\text{sub}}$

Example: Let us take $q = 53, N = 4, k = 2$. Let us take ciphertexts as

$$CT_1 = (x^3 + 3x^2 + 4x + 5, \quad 7x^3 + 2x^2 + x + 4, \quad 30x^3 + 36x^2 + 25x + 42)$$

$$CT_2 = (2x^3 + 4x^2 + 3x + 5, \quad 6x^3 + x^2 + 2x + 7, \quad 32x^3 + 35x^2 + 26x + 42)$$

Now we calculate:

$$CT_{\text{sub}} = (x^3 + 3x^2 + 4x + 5 - (2x^3 + 4x^2 + 3x + 5)) \mod (q, X^4 + 1)$$

$$= (-x^3 - x^2 + x) \mod 53 = (52x^3 + 52x^2 + x)$$

$$7x^3 + 2x^2 + x + 4 - (6x^3 + x^2 + 2x + 7) = x^3 + x^2 - x - 3 \mod 53 = x^3 + x^2 + 52x + 50$$

$$30x^3 + 36x^2 + 25x + 42 - (32x^3 + 35x^2 + 26x + 42) = -2x^3 + x^2 - x \mod 53 = 51x^3 + x^2 + 52x$$

$$\Rightarrow \boxed{CT_{\text{sub}} = (52x^3 + 52x^2 + x, \quad x^3 + x^2 + 52x + 50, \quad 51x^3 + x^2 + 52x)}$$

3.3 Multiplication on Encrypted Data

To multiply two ciphertexts, we perform polynomial multiplication since each ciphertext $\mathbf{CT} \in \mathcal{R}_{q,N}^{k+1}$ and after multiplication we reduce it in the ciphertext space $\mathcal{R}_{q,N}^{k+1}$. The ciphertext multiplication algorithm is as follows:

Algorithm 7 Ciphertext Multiplication in $\mathcal{R}_{q,N}^{k+1}$

Require: Ciphertexts $\mathbf{CT}_1 = (c_{1,0}, c_{1,1}, \ldots, c_{1,k})$, $\mathbf{CT}_2 = (c_{2,0}, c_{2,1}, \ldots, c_{2,k})$ in $\mathcal{R}_{q,N}^{k+1}$
Ensure: $\mathbf{CT}_{\text{mul}} = \mathbf{CT}_1 * \mathbf{CT}_2$ in $\mathcal{R}_{q,N}^{k+1}$
1: **for** $i = 0$ to k **do**
2: $\quad c_{\text{mul},i} \leftarrow (c_{1,i} * c_{2,i}) \mod (q, (X^n + 1))$
3: **end for**
4: $\mathbf{CT}_{\text{mul}} \leftarrow (c_{\text{mul},0}, c_{\text{mul},1}, \ldots, c_{\text{mul},k})$
5: **return** $\mathbf{CT}_{\text{mul}}$

Example: Let us take $q = 53, N = 4, k = 2$. Let us take ciphertexts as

$$CT_1 = (x^3 + 3x^2 + 4x + 5, \quad 7x^3 + 2x^2 + x + 4, \quad 30x^3 + 36x^2 + 25x + 42)$$

$$CT_2 = (2x^3 + 4x^2 + 3x + 5, \quad 6x^3 + x^2 + 2x + 7, \quad 32x^3 + 35x^2 + 26x + 42)$$

Now we compute component-wise multiplication modulo $(X^4 + 1, q = 53)$. Denote this by:

$$CT_{\text{mult}} = (CT_1[0] \cdot CT_2[0], \quad CT_1[1] \cdot CT_2[1], \quad CT_1[2] \cdot CT_2[2]) \mod (X^4 + 1, 53)$$

Step 1: Multiply the first polynomials

$$(x^3 + 3x^2 + 4x + 5)(2x^3 + 4x^2 + 3x + 5) =$$

$$= 2x^6+4x^5+3x^4+5x^3+6x^5+12x^4+9x^3+15x^2+8x^4+16x^3+12x^2+20x+10x^3+20x^2+15x+25$$

Now combine like terms:

$$= 2x^6+(4x^5+6x^5)+(3x^4+12x^4+8x^4)+(5x^3+9x^3+16x^3+10x^3)+(15x^2+12x^2+20x^2)+(20x+15x)+25$$

$$= 2x^6 + 10x^5 + 23x^4 + 40x^3 + 47x^2 + 35x + 25$$

Now reduce modulo $X^4 + 1$: Replace $x^4 = -1$, $x^5 = -x$, $x^6 = -x^2$, so:

$$2x^6 = -2x^2, \quad 10x^5 = -10x, \quad 23x^4 = -23$$

So final reduced form:

$$-2x^2 - 10x - 23 + 40x^3 + 47x^2 + 35x + 25 = 40x^3 + (47x^2 - 2x^2) + (35x - 10x) + (25 - 23)$$

$$= 40x^3 + 45x^2 + 25x + 2$$

All coefficients mod 53:

$$\Rightarrow 40x^3 + 45x^2 + 25x + 2$$

Repeat similarly for the other two components:

Step 2: Multiply second polynomials

$$(7x^3 + 2x^2 + x + 4)(6x^3 + x^2 + 2x + 7)$$

[We can expand and reduce similarly as above, and then finally reduce modulo $X^4 + 1$ and 53.]

Step 3: Multiply third polynomials

$$(30x^3 + 36x^2 + 25x + 42)(32x^3 + 35x^2 + 26x + 42)$$

[Again, follow same expansion and reduction.]

Final Answer:

$$CT_{\text{mult}} = (40x^3 + 45x^2 + 25x + 2, \quad \text{[2nd poly result]}, \quad \text{[3rd poly result]})$$

3.4 Chapter Conclusion

In this chapter, we demonstrated the implementation of basic arithmetic operations—addition, subtraction, and multiplication—on encrypted data. These operations form the foundation for building more complex functionalities. In the next chapter, we will discuss how we implemented operations such as less than, greater than, equal to, minimum, maximum, and sorting over encrypted data.

Chapter 4

Operation on Encrypted Data

For encrypted integer value we are using Zama's TFHE-based library. Zama's TFHE-based homomorphic encryption library supports comparison operations between ciphertexts, including `Equal`, `NotEqual`, `Greater`, `GreaterOrEqual`, `Less`, and `LessOrEqual`. Leveraging these primitives, we implemented advanced data processing tasks over encrypted datasets.

4.1 Comparison Operations on Encrypted Data

Let us take $ct_1 = FHE\_Enc(m_1) \quad and \quad ct_2 = FHE\_Enc(m_2)$. Zama provides the following comparison operators directly on encrypted integers:

- `Equal`(ct_1, ct_2): returns a ciphertext encoding 1 if $m_1 = m_2$, else 0.
- `NotEqual`(ct_1, ct_2): returns a ciphertext encoding 1 if $m_1 \neq m_2$, else 0.
- `Greater`(ct_1, ct_2): returns a ciphertext encoding 1 if $m_1 > m_2$, else 0.
- `GreaterOrEqual`(ct_1, ct_2): returns a ciphertext encoding 1 if $m_1 \geq m_2$, else 0.
- `Less`(ct_1, ct_2): returns a ciphertext encoding 1 if $m_1 < m_2$, else 0.
- `LessOrEqual`(ct_1, ct_2): returns a ciphertext encoding 1 if $m_1 \leq m_2$, else 0.

4.2 Minimum and Maximum Function

By using comparison functions in **section 4.1** and Arithmetic operations discussed in **Chapter 3**, We now implement minimum and maximum functions between two ciphertext. Let us take $CT_1 \quad and \quad CT_2$ be two cipher text. Now following way we can implemented minimum function

$$Min(CT_1, CT_2) = CT_1 * Less(CT_1, CT_2) + CT_2 * (1 - Less(CT_1, CT_2))$$

$$\implies Min(CT_1, CT_2) = CT_2 + (CT_1 - CT_2) * Less(CT_1, CT_2)$$

Now following way we can implemented maximum function between two ciphertext

$$Max(CT_1, CT_2) = CT_1 * Greater(CT_1, CT_2) + CT_2 * (1 - Greater(CT_1, CT_2))$$

$$\implies Max(CT_1, CT_2) = CT_2 + (CT_1 - CT_2) * Greater(CT_1, CT_2)$$

4.3 Sorting an Encrypted Array

By using the comparison functionality and arithmetic operations discussed in **Chapter 3**, we now implement a sorting algorithm on an encrypted array. For a given array

$$\text{Arr} = [m_0, m_1, \ldots, m_{k-1}]$$

- **Step 1:** We compute a *comparison matrix* **Comp_mat** defined as follows:

$$\textbf{Comp\_mat}_{i,j} = \begin{cases} \text{LT}(m_i, m_j) & \text{if } i < j \\ 0 & \text{if } i = j \\ 1 - \text{LT}(m_j, m_i) & \text{if } i > j \end{cases}$$

  Where **LT** is the function on plaintext.

- **Step 2:** For finding ascending order of an array, We compute the **Hamming weight** (i.e., the number of ones) of each column in the comparison matrix **Comp_mat**. The Hamming weight of the j-th column is denoted as $\textbf{H\_wt}(j)$.

- **Step 3:** Let **Arr_sort** be the sorted version of the array **Arr**. We compute each element of the sorted array as follows:

$$\textbf{Arr\_sort}[i] = \sum_{j=0}^{k-1} \text{Equal}(i, \textbf{H\_wt}(j)) \cdot m_j, \quad 0 \leq i \leq (k-1)$$

Example: Let us take the array:

$$\textbf{Arr} = [3, 9, 7, 1, 10]$$

The comparison matrix **Comp_mat** is:

$$\textbf{Comp\_mat} = \begin{bmatrix} 0 & 1 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 1 \\ 1 & 1 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

Now, we compute the **Hamming weight** of each column, which gives the number of ones in each column:

$$\begin{aligned}\mathbf{H\_wt}(0) &= 1\\ \mathbf{H\_wt}(1) &= 3\\ \mathbf{H\_wt}(2) &= 2\\ \mathbf{H\_wt}(3) &= 0\\ \mathbf{H\_wt}(4) &= 4\end{aligned}$$

Now we compute the sorted array **Arr_sort** using the formula:

$$\mathbf{Arr\_sort}[i] = \sum_{j=0}^{k-1} \mathrm{Equal}(i, \mathrm{H\_wt}(j)) \cdot m_j$$

Evaluating each position:

$$\begin{aligned}\mathbf{Arr\_sort}[0] &= \mathrm{Equal}(0, \mathbf{H\_wt}(3)) \cdot 1 = 1\\ \mathbf{Arr\_sort}[1] &= \mathrm{Equal}(1, \mathbf{H\_wt}(0)) \cdot 3 = 3\\ \mathbf{Arr\_sort}[2] &= \mathrm{Equal}(2, \mathbf{H\_wt}(2)) \cdot 7 = 7\\ \mathbf{Arr\_sort}[3] &= \mathrm{Equal}(3, \mathbf{H\_wt}(1)) \cdot 9 = 9\\ \mathbf{Arr\_sort}[4] &= \mathrm{Equal}(4, \mathbf{H\_wt}(4)) \cdot 10 = 10\end{aligned}$$

Therefore, the final sorted array is:

$$\mathbf{Arr\_sort} = [1, 3, 7, 9, 10]$$

Now we sorting array in **Descending** order so for that we first computing the **Hamming weight** of each row, which gives the number of ones in each row:

$$\mathbf{H\_wt}(0) = 3, \mathbf{H\_wt}(1) \quad = 1, \mathbf{H\_wt}(2) = 2, \mathbf{H\_wt}(3) \quad = 4, \mathbf{H\_wt}(4) = 0,$$

Evaluating each position:

$$\begin{aligned}\mathbf{Arr\_sort}[0] &= \mathrm{Equal}(0, \mathbf{H\_wt}(4)) \cdot 10 = 10\\ \mathbf{Arr\_sort}[1] &= \mathrm{Equal}(1, \mathbf{H\_wt}(1)) \cdot 9 = 9\\ \mathbf{Arr\_sort}[2] &= \mathrm{Equal}(2, \mathbf{H\_wt}(2)) \cdot 7 = 7\\ \mathbf{Arr\_sort}[3] &= \mathrm{Equal}(3, \mathbf{H\_wt}(0)) \cdot 3 = 3\\ \mathbf{Arr\_sort}[4] &= \mathrm{Equal}(4, \mathbf{H\_wt}(3)) \cdot 1 = 1\end{aligned}$$

Therefore, the final sorted array is:

$$\mathbf{Arr\_sort} = [10, 9, 7, 3, 1]$$

Remark: Since the comparison matrix **Comp_mat** is defined by $\frac{k(k-1)}{2}$ elements, it can be memory inefficient for large k. Instead, we can compute the Hamming weights of its columns by iteratively computing one comparison $\mathrm{LT}(m_i, m_j)$ with $i < j$ at a time.

To achieve this, we take an array of size k initialized with zeros, which will eventually store the Hamming weights of each columns. For each pair (i, j), we perform the following updates:

- If $i < j$ then Add the result of $\mathrm{LT}(m_i, m_j)$ to the j-th element of the array.
- If $i > j$ then Add the result of $1 - \mathrm{LT}(m_i, m_j)$ to the j-th element of the array.

In this approach, only k elements of the Hamming weight array need to be stored in RAM, instead of the full $\frac{k(k-1)}{2}$ comparison matrix.

4.3.1 Sorting algorithm for encrypted array

For encrypted array we use following algorithm for finding sorted encrypted array in ascending order:

Algorithm 8 Sort_Asc(CT_Arr)

Require: **CT_Arr** $= [CT_0, CT_1, \ldots, CT_{k-1}]$
Ensure: **Sort_CT_Arr**
for $j = 0$ to $k - 1$ **do** #Computing Hamming Weight
  H_wt[j] = 0
  for $i = 0$ to $k - 1$ **do**
    if $i < j$ **then**
      H_wt[j] $=$ **H_wt[j]** $+ \mathit{Less}(CT_i, CT_j)$
    end if
    if $i > j$ **then**
      H_wt[j] $=$ **H_wt[j]** $+ (1 - \mathit{Less}(CT_j, CT_i))$
    end if
  end for
end for
for $i = 0$ to $k - 1$ **do** #Computing Sorted array
  H_wt[j] = 0
  for $j = 0$ to $k - 1$ **do**
    Sort_CT_Arr$[i]$ $=$ **Sort_CT_Arr**$[i]$ $+$ $\mathit{Equal}($**FHE_Enc**$(i),$ **H_wt**$[j]) *$ **CT_Arr[j]**
  end for
end for
return Sort_CT_Arr

Note: To compute the descending order of an encrypted array, we calculate the Hamming weight of each row in **Comp_mat**, and the remaining steps follows from the algorithm used for finding the ascending order of the encrypted array.

4.4 Finding Minimum and Maximum

To find the minimum and maximum of an array, we iteratively apply the two-element minimum and maximum algorithms respectively. But for that It will take **O(n)** time.

The tournament method for finding the **maximum** of an array consists of $\lceil \log n \rceil$ iterations. In each round, the array is divided into pairs. If the array has an odd number of elements, one is passed unchanged to the next round. The maximum of each pair is propagated to the next round, and the process continues until only one element remains. This is the maximum of the input array.

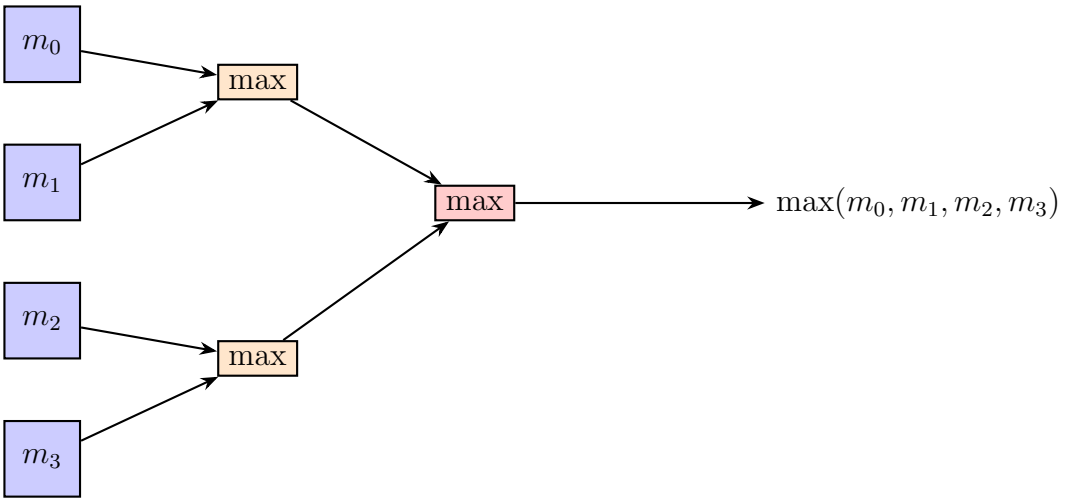


Figure 4.1: The tournament method of finding the maximum of an array.

To reduced the depth of the array maximum algorithm we can combine two method first we apply tournament method next we apply sorting algorithm for encrypted array. Let us take $Arr = [m_0, m_1, \ldots, m_{k-1}]$ be an input array after we apply $\mathbf{T}$ iteration of tournament method we get the $Arr' = [m'_0, m'_1, \ldots, m_{k'-1}$ of length $k' = \lceil \frac{k}{2^T} \rceil$ and Arr' contains the maximum element of the array Arr. Next we are apply sorting algorithm of encrypted array on Arr' and take the 0-th index element if we apply descending order or if we apply ascending order then we take the $(k'-1)$-th index of the sorted array for getting maximum element of the array.

Another way we can do that we compute the **Comp_mat** for Arr'. To find out the array maximum element that exploits the fact that the **Comp_mat** column related to the maximum contains only 1 except for the main diagonal entry. That is

$$\prod_{i=0, i\neq j}^{i=(k'-1)} \mathbf{Comp\_mat}_{i,j} = 1 \iff m'_j = max(Arr')$$

$$\implies max(Arr') = \sum_{j=0}^{k'-1} m_j \cdot \prod_{i=0, i\neq j}^{i=(k'-1)} \mathbf{Comp\_mat}_{i,j}$$

Note: Similarly, we can find the **minimum** element of an array by exploiting the fact that the row in the **Comp_mat** corresponding to the minimum element contains only 1s, except for the main diagonal entry. Therefore, the minimum can be found as follows:

$$\prod_{\substack{j=0 \\ j\neq i}}^{k'-1} \mathbf{Comp\_mat}_{i,j} = 1 \iff m'_i = \min(\mathbf{Arr'})$$

$$\implies \min(\mathbf{Arr}') = \sum_{i=0}^{k'-1} m_i \cdot \prod_{\substack{j=0 \\ j \neq i}}^{k'-1} \mathbf{Comp\_mat}_{i,j}$$

In this section, we implemented operations such as `Min`, `Max`, and sorting of an array using the `TFHE-rs` library, with plaintext inputs in the form of integers. This choice was made because encrypting floating-point numbers using `TFHE-rs` results in significantly large ciphertext sizes, making computations inefficient. To address this, in the next section, we use the CKKS scheme along with specific techniques to perform operations on encrypted floating-point data.

4.5 Comparison Algorithm for Encrypted Real-Valued Data

In the above setup, we perform comparison operations on encrypted data, including sorting, and computing the minimum and maximum over encrypted integer values. For floating-point numbers, we use the **CKKS** scheme to encrypt the data. Next, we define the comparison function as

$$Comp(m_0, m_1) = \begin{cases} 1 & \text{if } m_0 > m_1 \\ \frac{1}{2} & \text{if } m_0 = m_1 \\ 0 & \text{if } m_0 < m_1 \end{cases}$$

We defined the sign function by

$$Sign(x) = \begin{cases} 1 & \text{if } x > 0 \\ 0 & \text{if } x = 0 \\ -1 & \text{if } x < 0 \end{cases}$$

Therefore, the functions $Sign$ and $Comp$ are equivalent to each other by

$$Comp(m_0, m_1) = \frac{(Sign(m_0 - m_1) + 1)}{2}$$

Definition 3 *A polynomial f is (α, ϵ)-close to $Sign(x)$ over $[-1, 1]$ if it satisfies*

$$\|f(x) - sign(x)\|_{\infty,[-1,-\epsilon]\cup[\epsilon,1]} \leq 2^{-\alpha}$$

where $\alpha > 0$, $0 < \epsilon < 1$.

We get a result from [paper - 15] is the following -

Theorem 1 *If $d \geq \frac{1}{log(c_n)} \cdot log(1/\epsilon) + \frac{1}{log(n+1)} \cdot log(\alpha - 1) + O(1)$, then $f_n^d(x)$ is an (α, ϵ)-close polynomial to $Sign(x)$ over [-1, 1]. Where $f_n(x) = \sum_{i=0}^{n} \frac{1}{4^i} \cdot \binom{2i}{i} \cdot x(1 - x^2)^i$ and $f_n^d = f_n \circ f_n \circ \cdots \circ f_n$(d-times).*

4.5.1 Comparison Algorithm

Now we have introduced the comparison algorithm for encrypted real value. By **Theorem-1** we get $f_n^d(x)$ is approximation of $sign(x)$. We denote $f_n^d(x) \approx sign(x)$.

$$Since \quad Comp(m_0, m_1) = \frac{sign(m_0 - m_1) + 1}{2} \quad and \quad f_n^d(x) \approx sign(x).$$

$$\implies Comp(m_0, m_1) \approx \frac{f_n^d(m_0 - m_1) + 1}{2}.$$

Now following is the comparison algorithm

Algorithm 9 $Comp(m_0, m_1, n, d)$

Require: $m_0, m_1 \in [0, 1]$ and $n, d \in \mathbb{N}$
Ensure: An approximate value of 1 if $m_0 > m_1$, 0 if $m_0 < m_1$ and 1/2 otherwise.
1: $x \leftarrow (m_0 - m_1)$
2: **for** $i = 1$ to d **do**
3: $\quad x \leftarrow f_n(x)$
4: **end for**
5: **return** $\frac{(x+1)}{2}$

Note: In **Algorithm 9**, we observe that the *comparison* function between two plaintext values is approximated using a polynomial. Therefore, for comparing encrypted data, we can apply a similar approach. Since our encryption scheme is fully homomorphic, it allows us to implement the approximate polynomial form of the comparison function directly on encrypted values.

4.5.2 Minimum/Maximum

By using the *Sign* function, we are computing the Minimum and Maximum functions. Min/-Max functions corresponds to the absolute function as

$$Min(m_0, m_1) = \frac{m_0 + m_1}{2} - \frac{|m_0 - m_1|}{2}.$$

and

$$Max(m_0, m_1) = \frac{m_0 + m_1}{2} + \frac{|m_0 - m_1|}{2}.$$

Since $|x| = x \cdot Sign(x)$ and $f_n^d(x) \approx Sign(x)$, then $x \cdot f_n^d(x) \approx |x|$.
Therefore,

$$Min(m_0, m_1) \approx \frac{m_0 + m_1}{2} - \frac{(m_0 - m_1) \cdot f_n^d(m_0 - m_1)}{2}.$$

and

$$Max(m_0, m_1) \approx \frac{m_0 + m_1}{2} + \frac{(m_0 - m_1) \cdot f_n^d(m_0 - m_1)}{2}.$$

Now following is the algorithm of Maximum between two numbers -

Algorithm 10 `float_max`(m_0, m_1, n, d)

Require: $m_0, m_1 \in [0, 1]$ and $n, d \in \mathbb{N}$
Ensure: An approximate value of $Max(m_0, m_1)$
1: $x \leftarrow (m_0 - m_1)$, $y \leftarrow \frac{m_0+m_1}{2}$
2: **for** $i = 1$ to d **do**
3: $\quad x \leftarrow f_n(x)$
4: **end for**
5: $y \leftarrow y + \frac{m_0-m_1}{2} \cdot x$
6: **return** y

Now following is the algorithm of Minimum between two numbers -

Algorithm 11 `float_min`(m_0, m_1, n, d)

Require: $m_0, m_1 \in [0, 1]$ and $n, d \in \mathbb{N}$
Ensure: An approximate value of $Min(m_0, m_1)$
1: $x \leftarrow (m_0 - m_1)$, $y \leftarrow \frac{m_0+m_1}{2}$
2: **for** $i = 1$ to d **do**
3: $\quad x \leftarrow f_n(x)$
4: **end for**
5: $y \leftarrow y - \frac{m_0-m_1}{2} \cdot x$
6: **return** y

Note-1: In **Algorithm 10** and **Algorithm 11**, we observe that the *Min* and *Max* functions between two plaintext values is approximated using a polynomial. Therefore, for finding *Min/Max* of encrypted data, we can apply a similar approach. Since our encryption scheme is fully homomorphic, it allows us to implement the approximate polynomial form of the *Min/Max* function directly on encrypted values.

Note-2: By using **Algorithm 8** we can Implemented Ascending and Descending order for encrypted array of floating data. Next by using **section-4.4** we can implemented minimum and maximum for encrypted array of floating points.

4.6 Chapter Conclusion

In this chapter, we implemented `Min`, `Max`, and sorting operations on encrypted arrays for both integer and floating-point plaintexts. In the next chapter, we will see how these operations can be used to perform secure queries on a database.

Chapter 5

Query on encrypted data

In this chapter, we demonstrate how to perform basic database queries—such as `Min`, `Max`, `Average`, `Sorting`, and `WHERE` conditions—directly on encrypted data. By using previously implemented algorithm for finding min, max and sorting of an encrypted array.

5.1 Encrypted `AVG` Query

To compute the average over encrypted values:

- Sum all ciphertext values using homomorphic addition.
- Count the number of items (can be encrypted).
- At the client side, the client decrypts the sum value and divides it by the count and get the average value.

The following is the average algorithm on encrypted data -

Algorithm 12 $AVG(Enc\_Arr)$

Require: Enc_Arr is encrypted array.
Ensure: enc_sum is encrypted sum and $count$ is number of element in Enc_Arr

```
enc_sum = Enc_Arr[0]
count = 1
for val in Enc_Arr[1 :] do
    enc_sum = Add_Enc(enc_sum, val)
    count = count + 1
end for
return (enc_sum, count)
```

To perform an average query on encrypted data, we modify the client's average query into a custom query suitable for encrypted data. We then use the above algorithm to compute $(enc\_sum, count)$ and send these to the client. The client decrypts enc_sum using their secret key and divides it by $count$ to obtain the average value of the specified column.

5.2 Encrypted WHERE Queries

To perform a WHERE query on encrypted data, we input the encrypted array and an encrypted reference value. A comparison operation (e.g., less than, greater than, or equal) is then applied, returning an encrypted boolean array indicating the result of the comparison. The following algorithm we are using for performing where like query -

Algorithm 13 WHERE($Enc\_Arr, Enc\_Ref, operation$)

Require: Enc_Arr is encrypted array, Enc_Ref is encrypted reference value, $operation$ is comparison operation.

Ensure: enc_result is an encrypted boolean array indicating the result of the comparison.

```
enc_result =[ ]
if operation == "lt" then
    for val in Enc_Arr do
        result = Less(val, enc_ref)
        enc_result.append(result)
    end for
else if operation == "gt" then
    for val in Enc_Arr do
        result = Greater(val, enc_ref)
        enc_result.append(result)
    end for
else if operation == "ge" then
    for val in Enc_Arr do
        result = GreaterOrEqual(val, enc_ref)
        enc_result.append(result)
    end for
else if operation == "le" then
    for val in Enc_Arr do
        result = LessOrEqual(val, enc_ref)
        enc_result.append(result)
    end for
end if
return enc_result
```

The client initiates a WHERE query by sending an encrypted reference value along with the query request. The server then transforms this into a custom query suitable for encrypted data processing. It performs the comparison operation on the encrypted dataset and obtains an encrypted boolean array as the result. This encrypted result is sent back to the client. Finally, the client decrypts the boolean array using its secret key to retrieve the actual comparison outcomes.

5.3 Encrypted `ORDER BY` Clause

To perform the `ORDER BY` clause on encrypted data, we use **Section 4.5**. The client first initiates a sorting query on a specific column and sends the request to the server. The server then constructs a custom query suitable for encrypted data and uses the corresponding function to compute the sorted encrypted array. Finally, it sends the result back to the client, who decrypts it to obtain the sorted array for the specified column. Following way server calculate sorting of a column according to user query -

Algorithm 14 `ORDER_BY`(*Enc_Arr*, *order_type*)

Require: *Enc_Arr* is encrypted array and *order_type* is either asc or desc.
Ensure: Sorted encrypted array

```
if order_type == "asc" then
    result = Sort_Asc(Enc_Arr)        # Algorithm 8
else if order_type == "desc" then
    result = Sort_Desc(Enc_Arr)
end if
return result
```

5.4 Encrypted `MIN` and `MAX` Queries

To compute the `Min` or `Max` of an encrypted array, we utilize different strategies depending on the data type of the encrypted values. For **encrypted integer arrays**, we adopt the comparison-based technique described in **Section 4.4**. On the other hand, for **encrypted floating-point arrays**, we rely on the CKKS-based method discussed in **Section 4.5.2**. The following unified algorithm outlines how the system dynamically selects the appropriate procedure based on the input data type:

Algorithm 15 `Max`(*Enc_Arr*, *data_type*)

Require: *Enc_Arr* is encrypted array and *data_type* is either `"int"` or `"float"`.
Ensure: Maximum value in encrypted form

```
if data_type == "int" then
    result = int_max(Enc_Arr)                    # Uses logic from Section 4.4
else if data_type == "float" then
    result = float_max(Enc_Arr)                  # Uses logic from Section 4.5.2
end if
return result
```

This abstraction ensures modularity in our design, allowing the system to support multiple encrypted numeric types seamlessly.

Chapter 6

Benchmarks

6.1 Benchmark Setup and Environment

In this section, we present the performance benchmarks for various homomorphic operations such as `AVG`, `MIN`, `MAX` and `SORT` executed on encrypted data.

All experiments were conducted on an **AWS** `hpc7a.96xlarge` instance, equipped with a **96-core AMD EPYC 9R14 CPU** running at **2.60 GHz** and **740 GB RAM**.

The evaluation considers varying numbers of encrypted rows (ranging from 1000 to 5000) and data type size as FHE Units (8, 16, 32, 64, and 128). Each benchmark records the query execution time in seconds, measuring how efficiently homomorphic encryption operations can be performed at scale.

6.2 For Integer Columns

There are various types of integers such as `uint8`, `uint16`, `uint32`, `uint64`, and `uint128`. The time complexity for performing operations on encrypted data varies depending on the integer type. In the following section, we present the time complexities of different queries (e.g., average, min/max, and sorting) on encrypted data for various integer types.

6.2.1 Avg-Query

The following graph shows the time complexity of the average query across different row sizes and integer data types.

Table 6.1: AVG Query Execution Time (in seconds)

Row Number	FHE Unit 8	FHE Unit 16	FHE Unit 32	FHE Unit 64	FHE Unit 128
1000	0.423	0.440	0.604	0.781	1.250
2000	0.467	0.484	0.665	0.859	1.376
3000	0.491	0.510	0.701	0.904	1.455
4000	0.508	0.528	0.726	0.937	1.550
5000	0.522	0.543	0.745	0.962	1.542

As the row size increases, the query execution time also increases. Additionally, larger integer types (e.g., `uint128`) generally result in higher computational overhead due to increased encrypted value size.

6.2.2 `Min/Max`-Query

The following graph shows the time complexity of the min/max query across different row sizes and integer data types.

Row Number	FHE Unit 8	FHE Unit 16	FHE Unit 32	FHE Unit 64	FHE Unit 128
1000	0.582	0.698	0.870	1.050	1.402
2000	0.640	0.768	0.958	1.155	1.542
3000	0.674	0.809	1.008	1.217	1.625
4000	0.698	0.838	1.045	1.261	1.684
5000	0.717	0.860	1.073	1.295	1.729

Table 6.2: Min/Max Query Execution Time (in seconds)

As the number of rows increases, the time required to compute the minimum or maximum value for a encrypted column also increases. Larger integer types (e.g., `uint128`) introduce additional computational cost due to the complexity of homomorphic comparison operations on longer encrypted bit representations.

6.2.3 `ORDER By`-Query

The following graph shows the time complexity of the `ORDER BY` query across different row sizes and integer data types.

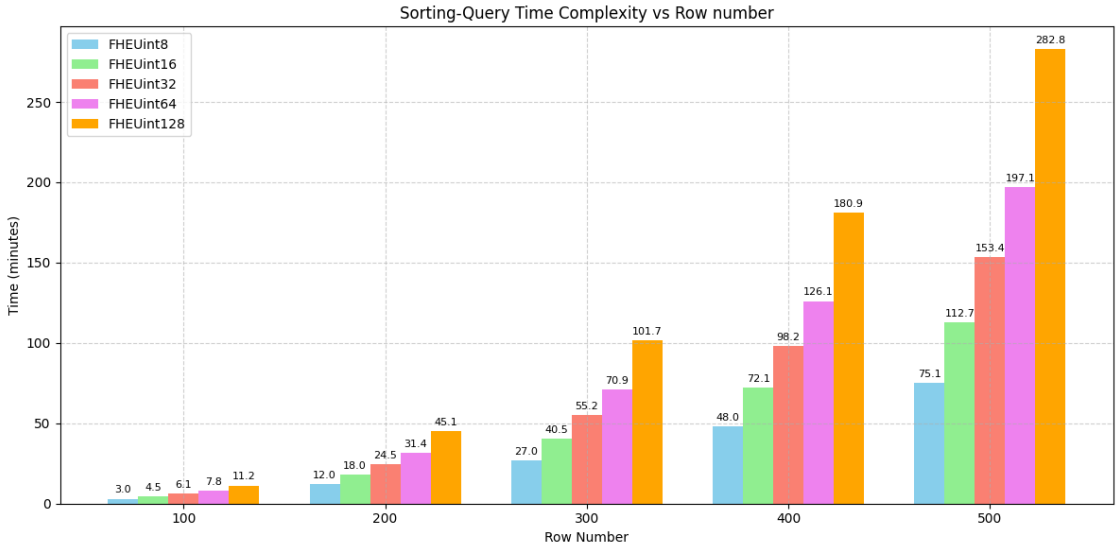


Figure 6.1: Time Complexity of `Orderby`-Query

As the number of rows increases, the sorting time grows accordingly due to the increased number of encrypted comparisons. Larger integer types (e.g., `uint128`) lead to higher computation times, as sorting encrypted values requires bitwise comparisons on longer ciphertexts. This evaluation emphasizes the trade-off between data precision and performance in encrypted sorting, making it crucial to choose the appropriate integer type based on both security and efficiency requirements.

6.3 For Floating Columns

In this section, we calculate the running time of various queries on encrypted floating-point data. We recall that we are using the **CKKS** algorithm for encrypting floating-point numbers between 0 and 1. It is important to note that CKKS is an *approximate* encryption scheme, and thus we assume an approximation error of the form $2^{-\alpha}$, where α is a tunable precision parameter.

6.3.1 `Min/Max`-Query on Floating Points

To evaluate the performance of Min/Max operations on encrypted floating-point arrays, we perform experiments with increasing data size and varying values of the approximation parameter α. As the value of α increases, the ciphertext becomes more precise but also more computationally expensive to process. For experiment purpose we are using **HEAAN** Library.

Row Numbers	$\alpha = 8$	$\alpha = 12$	$\alpha = 16$	$\alpha = 20$
100	4.53 min	7.14 min	11.59 min	21.87 min
200	5.21 min	8.21 min	13.33 min	25.17 min
300	5.61 min	8.84 min	14.35 min	27.09 min
400	5.89 min	9.29 min	15.08 min	28.46 min
500	6.11 min	9.63 min	15.64 min	29.52 min

Table 6.3: Time complexity of Min/Max-Query on encrypted floating-point data using CKKS

From the table above, we observe that the time required to compute the Min/Max increases both with the number of encrypted rows and with the precision level α. This trade-off between precision and performance is inherent to approximate homomorphic encryption schemes like CKKS and must be carefully tuned based on application requirements.

Note: For the sorting algorithm on encrypted data, this will take a lot of time.

Chapter 7

Conclusion and Future Work

7.1 Conclusion

In this thesis, we have developed and analyzed a framework for performing secure queries over encrypted data using Fully Homomorphic Encryption (FHE). Beginning with an overview of the fundamentals of FHE and its relevance in securing cloud-based computations, we described in detail the encryption and decryption protocols for integers, floating-point numbers, and structured data.

We designed and implemented homomorphic arithmetic operations such as addition, subtraction, and multiplication. Building upon this, we implemented core database query functionalities—including `Min`, `Max`, `AVG`, `ORDER BY`, and `WHERE` clauses—over encrypted data. For integer data types, comparison and sorting operations were designed based on comparator circuits. For floating-point data, we used the CKKS scheme, with special attention to handling approximate errors within acceptable bounds (typically of the form $2^{-\alpha}$).

Our benchmarking results demonstrate that while encrypted queries incur significantly higher computation times compared to plaintext queries, the system remains practical for datasets of moderate size. In particular, we observed a clear trade-off between computational cost and precision in the case of floating-point operations.

This work establishes a secure and functional base for privacy-preserving data analytics.

7.2 Future Work

Despite the progress achieved, several avenues remain open for future exploration and improvement:

- **Optimizing Performance:** Utilize batching, SIMD techniques, and parallelization to improve execution speed.
- **Noise Management:** Develop more efficient methods for reducing ciphertext noise growth and rescaling.
- **Ciphertext Compression:** Investigate techniques to reduce ciphertext size, especially for large datasets.

- **Complex SQL Support:** Extend support to advanced SQL operations like joins, nested queries, and groupings.

These directions not only promise performance enhancements but also broaden the applicability of secure encrypted querying systems in real-world, large-scale scenarios.

Bibliography

1. EStore A User-Friendly Encrypted Storage Scheme for Distributed File Systems. uxiang Chen, Guishan Dong, Chunxiang Xu, Yao Hao and Yue Zha. https://www.researchgate.net/publication/374804091_EStore_A_User-Friendly_Encrypted_Storage_Scheme_for_Distributed_File_Systems

2. Hybrid Implementation of Twofish AES ElGamal and RSA Cryptosystems. https://www.researchgate.net/publication/346943016_Hybrid_Implementation_of_Twofish_AES_ElGamal_and_RSA_Cryptosystems

3. Securing Big Data Processing With Homomorphic Encryption. https://www.researchgate.net/publication/340644952_Securing_Big_Data_Processing_With_Homomorphic_Encryption

4. HOPE: Homomorphic Order-Preserving Encryption for Outsourced Databases—A Stateless Approach. https://arxiv.org/pdf/2411.17009

5. Craig Gentry, *A Fully Homomorphic Encryption Scheme*, 2009. https://crypto.stanford.edu/craig/craig-thesis.pdf

6. Zvika Brakerski, Vinod Vaikuntanathan, *Fully Homomorphic Encryption from Ring-LWE*, 2011. https://eprint.iacr.org/2011/277.pdf

7. HOPE: Homomorphic Order-Preserving Encryption for Outsourced Databases—A Stateless Approach. https://arxiv.org/html/2411.17009v1

8. Attribute based honey encryption algorithm for securing big data: Hadoop distributed file system perspective. https://peerj.com/articles/cs-259/

9. An approach for enhancing data confidentiality in Hadoop. https://www.researchgate.net/publication/344071068_An_approach_for_enhancing_data_confidentiality_in_Hadoop

10. Approximate Homomorphic Encryption with Reduced Approximation Error. https://eprint.iacr.org/2020/1118.pdf

11. Microsoft SEAL Library Documentation.
https://github.com/microsoft/SEAL

12. HElib Library Documentation.
https://github.com/homenc/HElib

13. Somewhat Practical Fully Homomorphic Encryption.
Junfeng Fan and Frederik Vercauteren.
https://eprint.iacr.org/2012/144.pdf

14. A New CRT-based Fully Homomorphic Encryption.
Anil Kumar Pradhan, Vaultree Ltd.
https://eprint.iacr.org/2024/1105

15. Efficient Homomorphic Comparison Methods with Optimal Complexity.
Jung Hee Cheon, Dongwoo Kim, and Duhyeong Kim.
https://eprint.iacr.org/2019/1234

16. A Complete Beginner Guide to the Number Theoretic Transform (NTT).
Ardianto Satriawan, Rella Mareta, Hanho Lee.
https://eprint.iacr.org/2024/585

17. Faster homomorphic comparison operations for BGV and BFV.
Ilia Iliashenko1 and Vincent Zucca.
https://eprint.iacr.org/2021/315.pdf

18. TFHE-rs Library.
https://docs.zama.ai/tfhe-rs

19. TFHE-rs Hand book.
https://github.com/zama-ai/tfhe-rs-handbook/blob/main/tfhe-rs-handbook.pdf