Homomorphic Encryption

Simay Göçen
Department of Computer Engineering
Faculty of Engineering , Dokuz Eylül University
İzmir/Türkiye
simay.gocen@ogr.deu.edu.tr

***Abstract*— Homomorphic encryption has emerged as a groundbreaking solution in secure computation, enabling operations on encrypted data without decryption. This report provides a comprehensive overview of homomorphic encryption, delving into its foundational concepts and significance. Additionally, it surveys relevant literature to highlight the advancements and applications within this domain. The primary focus lies in a comparative analysis of two prominent homomorphic encryption schemes, BFV and CKKS, examining their code implementation, performance metrics, and efficiency. Through this comparative study, insights into their respective strengths and weaknesses are revealed, shedding light on the practical considerations for implementation. The findings contribute to a deeper understanding of the practical implications and performance characteristics of homomorphic encryption schemes.**

Keywords—encryption,homomorphic,homomorphic encryption,cryptography,SEAL,BFV Scheme,CKKS Scheme

I. INTRODUCTION

A cutting-edge cryptographic technique that can help with today's privacy and data security issues is homomorphic encryption. This unique kind of encryption lets you work with the data while maintaining its confidentiality and enabling computations. One cryptographic method that allows for the execution of mathematical operations on encrypted data is homomorphic encryption. This maintains the confidentiality of the data by enabling you to compute on encrypted data. Ordinarily, operations cannot be completed without first decrypting the data; however, homomorphic encryption solves this problem. When public key encryption was initially developed in the late 1970s, Diffie and Hellman proposed the concept of homomorphic encryption. Nevertheless, this concept was never really put into practice, and Craig Gentry created an encryption system in 2009 before it was finished. This marked the advent of completely homomorphic encryption and was regarded as a potent cryptographic instrument.

Advantages:

• **Data Privacy**: Even when being processed, encrypted data is kept private.

• Data security in cloud-based computations is guaranteed by secure cloud computing.

• **Data sharing**: Enables the safe exchange of sensitive data.

• Data analytics: Offers the capability to examine data that has been encrypted.

Disadvantages:

• **Processing Load:** Since calculations are encrypted, the processing load increases and requires more computing resources.

• **Performance:** Homomorphic encryption can be demanding on the processor and slow in some cases.

• **Complexity:** Can be complex to use in practice and may require special expertise..

Types of homomorphic encryption :

• Algorithms for **Partially Homomorphic Encryption** make it easier to analyze circuits composed of a single type of gate, like addition or multiplication gates

• For two distinct types of gates, only a restricted number of circuits may be analyzed using **Somewhat Homomorphic Encryption** algorithms.

• **Leveled Fully Homomorphic Encryption** makes it easier to evaluate arbitrary circuits with bounded (pre-specified) depth that consist of many gate types.

• **Fully Homomorphic Encryption (FHE),** the most powerful concept in homomorphic encryption, allows any circuit composed of multiple gate types to be evaluated with unlimited depth.

Users can access particular cloud data segments thanks to homomorphic encryption, which eliminates the requirement to decrypt the complete data block. This methodology guarantees the safe preservation of confidential information while permitting precise analytical processes. When users access data in untrusted locations, such as public clouds, this becomes even more important. Data is consistently encrypted when homomorphic encryption is used, which reduces the possibility of data compromise. Essentially, homomorphic encryption improves cloud security and can be used in a variety of industries, including government, blockchain, AI & ML, and healthcare.

Applications requiring homomorphic encryption can be developed using the library offered by Microsoft SEAL (Simple Encrypted Arithmetic Library). It speeds up research and development and makes homomorphic encryption usable in projects even for developers who are not familiar with it. Microsoft SEAL is an effective tool for industrial and research purposes.

II. Related Works

II.I Privacy-preserving federated learning based on multi-key homomorphic encryption [1]

Federated learning is a distributed machine learning concept. It offers the ability to train a machine learning model using data from decentralized devices. A remote device receives the local data from the central server in order to train the global model utilizing it. After that, it sends the model update back to the server after compiling the data. The server accumulates and disperses model modifications from remote devices to generate a new global model. The server and remote devices communicate via iterative model updates and the computed global model during training.

This article emphasizes how security and privacy concerns have become crucial in mobile services and networks with the introduction of machine learning and the Internet of Things. Sensitive data can be compromised during data transfers to a central unit; federated learning minimizes this requirement by sharing only model changes as opposed to local data. Privacy leaks are still an issue, though. This study presents an enhanced multi-key homomorphic encryption protocol, called xMK-CKKS, and employs it to create a novel privacy-preserving federated learning method. This technique requires all participating devices to work together to encrypt model updates before sharing them with a server for analysis. This technique is robust to collaboration between less than n devices (n-1) participating devices and servers, and prevents privacy leakage from publicly shared model updates in federated learning. According to evaluation results, this technique preserves model accuracy while performing better than previous advances in terms of communication and computing cost.

The recommended xMK-CKKS based federated learning technique protects data secrecy via multi-key homomorphic encryption. Through the optimization of the MK-CKKS homomorphic encryption algorithm, this method reduces the danger of privacy leakage in the context of federated learning. The xMK-CKKS scheme provides a combined public key and decryption share for secure and simple encryption and decryption. As such, it is more appropriate for privacy preservation in federated learning scenarios. Furthermore, the xMK-CKKS-based federated learning system protects the confidentiality of model updates by multi-key homomorphic encryption and is impervious to collaboration attacks between involved devices and the server. The proposed strategy provides strong privacy protection for federated learning..

We evaluated the federated learning strategy based on xMK-CKKS in terms of accuracy, compute cost, energy usage, and communication cost. Additionally, we compared it to federated learning using MK-CKKS and Paillier homomorphic encryption. Specifically, we conducted a large-scale data scenario experiment in elder care using ten Jetson Nano IoT devices and one server. We have carefully evaluated and compared the plans. The experiment shows the efficiency, efficacy, and adaptability of our method in IoT domains as well as in terms of communication costs, computing costs, accuracy, and energy consumption. As a result, safe federated learning may be implemented on Internet of Things devices.

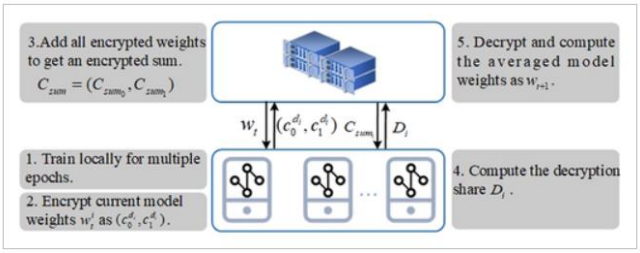


Figure: Federated learning that protects privacy using xMK-CKKS multi-key homomorphic encryption

II.II A Survey on Fully Homomorphic Encryption: An Engineering Perspective [2]

This article examines the stronger and more recent Fully Homomorphic Encryption (FHE) schemes in detail as well as reviewing the older and more modern Somewhat Homomorphic Encryption (SHE) methods. These systems' underlying concepts are explained, and an engineering viewpoint is used to assess the security and performance of the schemes.In other words, the article considers homomorphic encryption algorithms for data processing and storage that are encrypted, and it gives information about these techniques' functionality and security from an engineering standpoint.

By introducing the first tenable technique that supports any computation on cryptographic data, Gentry's PhD thesis from 2009 revolutionized the area of cryptography. This has long been regarded as the pinnacle of cryptography since it is applicable to a wide range of industries, including data mining and cloud computing. The initial plan was implemented inefficiently and was based on ideas. However, more effective schemes have been devised as a result of the core idea—using a SHE scheme that can evaluate its own decryption circuit—being applied repeatedly to different encryption systems (LWE, NTRU, and AGCD). Other significant strategies, including replacing the switches and modules, were suggested, and these systems improved in efficiency.

This study examines current SHE and FHE encryption strategies and tactics that are based on Gentry's research. It has been demonstrated that re-encryption processes make universal FHE schemes impractical in terms of performance, based on experimental findings given for different systems. However, methods like fixed

perspectives and module swapping make schemes created for particular applications applicable.

Future research will focus on a few topics, including security. Although the majority of FHE schemes available today provide security levels that are adequate for some applications, this study's discussion of IND-CCA1 renders them unsuitable for general-purpose applications.

To sum up, the identification and creation of the first FHE schemes marks a significant advancement in both thought and problem-solving methodology. It is intended that this study will serve as a valuable resource for comprehending the fundamental ideas behind SHE/FHE schemes and advancing the subject.

II.III. New Fully Homomorphic Encryption Scheme Based on Multistage Partial Homomorphic Encryption Applied in Cloud Computing [3]

According to this article, security issues continue to impede the widespread adoption of cloud computing, despite its status as one of the most significant developments in the computing industry. To protect data on cloud servers, numerous encryption algorithms are employed. Cloud storage allows for the encrypted storing of data. Nevertheless, a recently developed method known as "homomorphic encryption" allows specific operations to be performed on encrypted data.

The article provides an overview of security issues related to cloud computing and asserts that completely homomorphic encryption is not suited for secure cloud computing because of its flaws, which include enormous key sizes and subpar computational efficiency. With these problems in mind, a "hybrid homomorphic encryption scheme" was created to increase data security. It blends a homomorphic encryption technique based on the GM encryption algorithm—which is perfect for addition—with a homomorphic encryption scheme based on the RSA algorithm, which is wonderful for multiplication.

By utilizing this hybrid encryption technique, it is possible to effectively circumvent the drawbacks of homomorphic encryption systems and take advantage of their resistance to privacy breaches. With the use of a two-layer encryption technology, this hybrid encryption algorithm boosts security of data saved in the cloud while increasing speed (2.9 times) and reducing computation time (66%). As a result, hybrid encryption facilitates authentication and offers great security.

They analyze the performance of this implementation of Hybrid Homomorphic Encryption. The tables below show the encryption and decryption times for three different plain text sizes for the EHES algorithm, which supports a single operation multiplicative process, and the recommended approach, which supports two operations additive-multiplicative processes three times faster.

The study describes the combination of GM and RSA algorithms, which increases data privacy and broadens the techniques' application regions. The following is a summary of the research: First, text data is encrypted using GM. This encrypted data is used as the system's input data and is subsequently re-encrypted using RSA. This data is subjected to calculations, and decryption is accomplished by repeating the same procedure. The goal is to decrease calculation time and boost the level of security offered individually by merging these current methods, which is a promising field for further study.

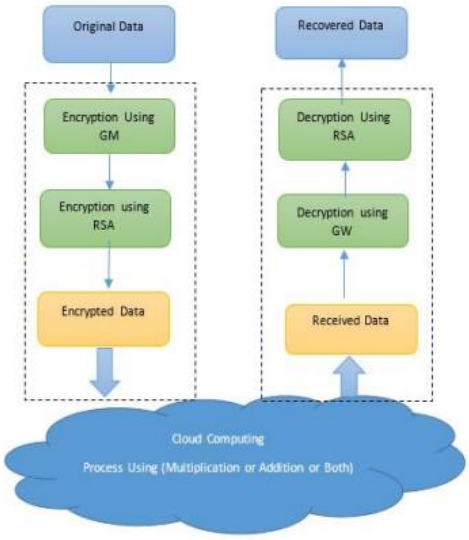


Figure : Hybrid Homomorphic Encryption Scheme Proposed Method

II.IV Research on Homomorphic Encryption for Arithmetic of Approximate Numbers [4]

Approximate Number Arithmetic for Homomorphic Encryption (HEAAN), an enhanced variant of homomorphic encryption (HE), is described in this article. Plaintext encryption, computation, and decryption are supported by HEAAN, and the outcomes of these operations match calculations in plaintext. Addition and multiplication are involved in these computations. Data security, privacy protection, and security breach prevention are all ensured by HEAAN. HEAAN provides precision of 32 bits.

Addition and multiplication operations on encrypted integer data are supported by traditional Full Homomorphic Encryption (FHE) systems, producing outcomes that are similar to plaintext calculations. However, floating point numbers are not supported by this type of FHE; it can only accept integers. The scope of implementing real-time calculation includes both real and floating point numbers. With the HEAAN Encryption technique, the limitations of traditional FHE systems are circumvented. HEAAN's components include scaling, homomorphic addition and multiplication, encryption, and decryption. HEAAN retains the first few significant digits of the computed result, regardless of its magnitude. This is done using a Python library called Pi-HEAAN.

The homomorphic addition and multiplication operations scale with the encryption method called "Homomorphic Encryption for Approximate Numbers," or HEAAN. This method can also be used to encrypt floating point numbers and other real numbers on the number line. This method reduces the ciphertext to a smaller modulus and rounds the plaintext.

The need for safe data processing grows as the technical effect does as well. For this reason, the HEAAN system is in use. Operations on encrypted data that are equivalent to those on plaintext can be carried out thanks to the HEAAN system. The computation is carried out using encrypted data; the real parameters are not accessible. After the value has been encrypted, analysis is done. As a result, security breaches are avoided and data integrity and security are increased.

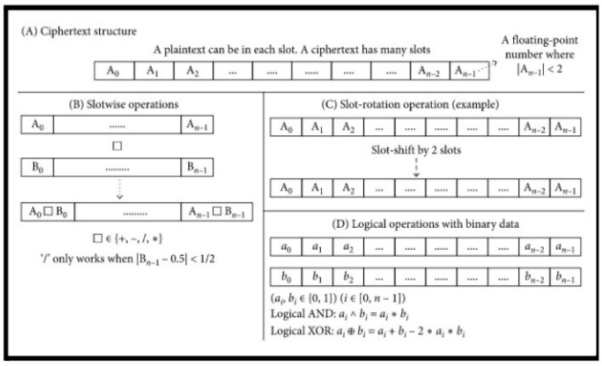


Figure:HEAAN operations and structure

Times(s)	Addition	Multiplication
Encryption Time	0.370	0.370
Computation Time	0.023	1.000
Decryption Time	0.084	0.084
Total Time	0.477	1.454

Figure: Time Analysis

II.V. On the Feasibility of Homomorphic Encryption for Internet of Things [5]

This article looks at homomorphic encryption's (HE) performance, energy consumption, and usability in Internet of Things devices. Although HE enables operations on encrypted data, it is currently challenging to utilize on Internet of Things devices efficiently. According to the article, enhanced hardware capabilities and optimized libraries can lead to greater efficiency in HE systems. Additionally, it uses the SEAL and HElib libraries on the Raspberry Pi 4 hardware platform to examine the performance and energy consumption of various HE schemes. The findings assess how well HE schemes balance energy usage, performance, and security and offer recommendations for enhancing HE for Internet of Things applications. Based on the needs of IoT applications, the findings of this study can aid in the development of resource-constrained, secure HE apps.

They employ a hardware energy tester to measure energy consumption. An Atorch USB energy tester, positioned in between the board and the power source, provides power to the Raspberry Pi 4 board for measurement. Additionally, they log the algorithm's execution time and power consumption in watts. Lastly, they use the product of power consumption and execution time to determine energy usage in joules.

They examine Microsoft SEAL and HElib's performance in this experiment. They are further assessed on SEAL with encryption, decryption, multiplication, and addition operations for BFV and CKKS systems. HElib with BGV and CKKS schemes are the schemes that are analyzed.After that, they noted the execution time along with the degrees (1024, 2048, 4096, 8192, and 16384). Next, each scheme's CPU consumption was measured. Additionally, a comparison was made between the memory utilization of SEAL and HElib for each method. Lastly, they examined how much energy each homomorphic library used. A graphical plotting and analysis are done on the comparisons with homomorphic encryption, schemes, CPU, memory, and energy consumption.

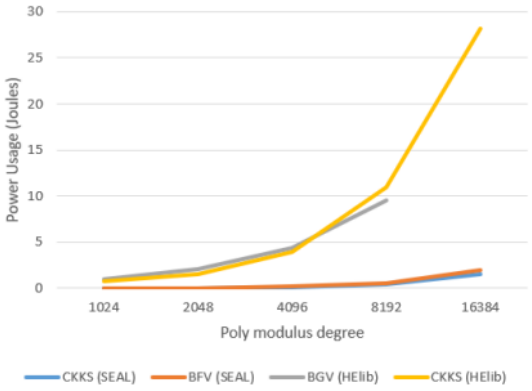


Figure : SEAL & HElib's power usage on a Raspberry Pi 4

III. FULLY HOMOMORPHIC ENCRYPTION SCEHEMES ANALYSIS

Fully Homomorphic Encryption (FHE), the most powerful concept in homomorphic encryption, allows any circuit composed of multiple gate types to be evaluated with unlimited depth.

Applications requiring homomorphic encryption can be developed using the library offered by MicrosoftSEAL (Simple Encrypted Arithmetic Library). It speeds up research and development and makes homomorphic encryption usable in projects even for developers who are not familiar with it. MicrosoftSEAL is an effective tool for industrial and research purposes.

Two distinct homomorphic encryption methods are included with Microsoft SEAL. Modular arithmetic is made possible on encrypted integers via the BFV system. While encrypted real or complex numbers can be added to and multiplied using the CKKS technique, the results are simply approximations. For gathering encrypted real numbers, assessing machine learning models in encrypted data, or figuring out transaction distances between encrypted places, CKKS is the greatest choice. The only available scheme for applications that require accurate quantities is the BFV scheme.

III.I. Brakerski-Fan-Vercauteren (BFV) Scheme:

Plaintext Space and Polynomials:

- The formula for the plaintext field is Rt=Zt[x]/(x^n+1). This is a representation of polynomials of order *n*, where *t* takes the coefficients.
- When working with Rt, a Ring structure is employed, where the product of plaintext polynomials x^n results in x^(-1).

Encryption and Decryption Procedures:

- Integer or rational values have to be converted into Rt polynomials in plaintext before they can been crypted.
- A private key is needed for the process of encryption, and encrypted polynomials can undergo homomorphic addition and multiplication.
- The goal of the decoding procedure is to restore the original values of the encrypted polynomials.

Ciphertext Space and Relinearization:

- Polynomial sequences on Rq are ciphertexts. These arrays have dimensions that grow until relinearization is carried out in homomorphic multiplication operations since they comprise at least two polynomials.
- Relinearization is used to lower the cost by shrinking the results of homomorphic multiplication.

The Modular Arithmetic Relationship and the CRT Theorem:

- The Chinese Remainder Theorem, or CRT for short, is a modular arithmetic theorem that permits the use of several modules. Different modular rings are frequently worked with in the BFV design.
- In modular arithmetic, rings like Rt and Rq are exceptional examples.
- Modular Arithmetic: In this type of arithmetic, a number is calculated over a specified module. Modular arithmetic is typically performed on Rt and Rq in the BFV scheme.

Modulus and Relinearization:

- A number called modulus is used to calculate how certain operations on a ring will turn out. t and q in the BFV diagram typically correspond to modulus values in modular rings.
- Relinearization is used to lower the computational cost and decrease the enormous ciphertexts that are produced by homomorphic multiplication procedures. Relinearization is achieved by decreasing the size of the ciphertext by eliminating part of the multipliers.

III.II. CKKS (Cheon-Kim-Kim-Song) Scheme:

One homomorphic encryption system that works well for numerical computations is called CKKS. This approach is intended primarily for applications involving homomorphic encryption and real integers.

Plaintext and Polynomials:

- Real numbers are present in the plaintext field of the CKKS scheme.
- A polynomial is used to represent the elements of plaintext. Specifically, every coefficient of a polynomial in plaintext represents a real integer.

Area of Ciphertext:

- Polynomial sequences on Rq are ciphertexts. Relinearization is used to regulate these arrays as they increase via homomorphic multiplication processes.

Relinearization and Modular Arithmetic:

- Modular arithmetic is employed in the CKKS scheme, but its capacity to handle real numbers sets it apart from other schemes.
- Large ciphertexts produced by homomorphic multiplication operations can be shrunk and costs can be decreased by using relinearization.

Custom methods and Optimizations:

- The mathematical methods and optimizations found in CKKS are exclusive. It primarily concentrates on homomorphic addition and multiplication operations as well as numerical precision control.

When doing numerical analysis, database queries, and other real number operations under homomorphic encryption, the CKKS method is an especially good option.

In Summary;

BFV:

- Polynomials in plaintext space are specified on Rt.
- The polynomial arrays on Rq are the ciphertexts.
- Particularly for homomorphic encryption applications over integers or small-sized data types, the BFV technique was created.
- Semantic security and integer arithmetic are its main topics.
- BFV has stricter error control and numerical precision requirements than the CKKS method.
- Because it uses integer arithmetic, there is minimal precision loss.
- In particular, the BFV system is tuned for integer arithmetic.
- In applications that are appropriate, it can offer superior performance.

CKKS:

- Real numbers make up the plaintext field, and polynomials are used to represent these real values.
- Although ciphertexts are more directly representations of real numbers, they are still polynomial arrays over Rq.
- Particularly for homomorphic encryption applications over real numbers and numerical computations, the CKKS technique was created.
- Offers extremely accurate computations and concentrates on mathematical optimizations
- Because CKKS uses real numbers, it provides a more flexible method of achieving numerical precision.
- Homomorphic addition and multiplication operations are the main focus of precision control during computation.
- CKKS is best suited for numerical computations and real numbers.
- It achieves high precision by concentrating on mathematical optimizations alone.

In the subject of homomorphic encryption, both systems have a wide range of application possibilities; nevertheless, the scenario in which they are employed and the method of application will determine which is better.

For instance, the CKKS scheme is more appropriate for numerical computations and operates with real numbers, whereas the BFV system operates with integers.

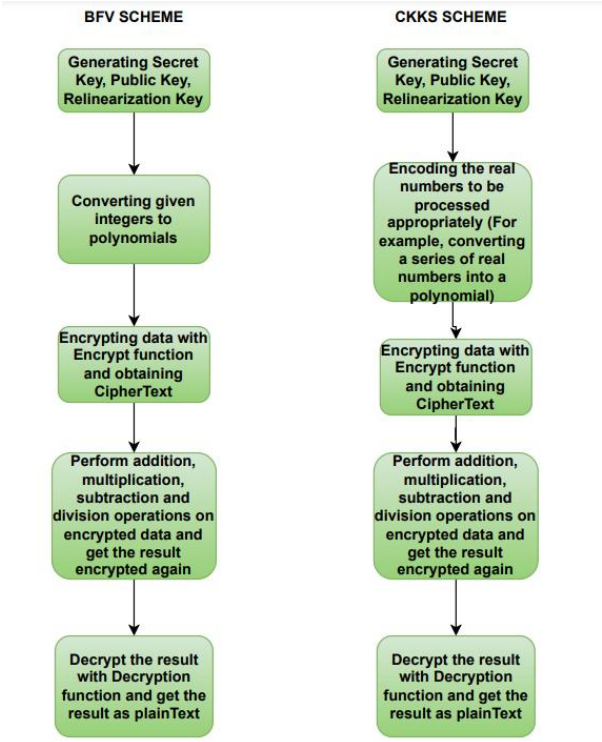
III.III My Project:

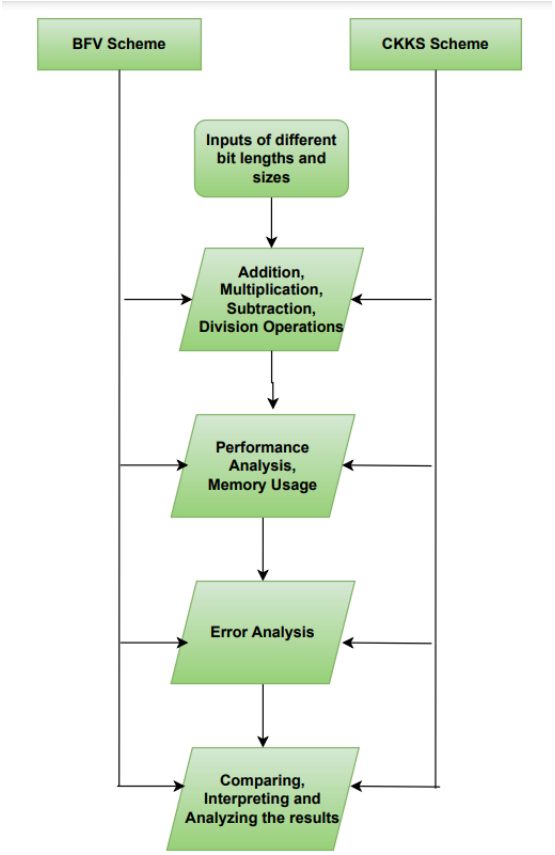
In my project, I will basically code the algorithms of BFV and CKKS schemes in Python language. I will try to show addition, multiplication, subtraction operations for both schemes. I will compare the ciphertext sizes created in both schemes. I will enter inputs of different bit sizes and show how the outputs return according to these inputs. I will prove the differences I just mentioned in code.

I will analyze and compare both schemes from these aspects;

- Time Analysis
- Memory Usage
- Sensitivity Control
- Efficiency Comparisons
- Security Levels
- Literature Comparisons

Flowcharts of this Project:





IV. IMPLEMENTATION OF THE PROJECT

Basic mathematical concepts underlying the techniques used in the homomorphic encryption processIt is important. Modular arithmetic,abstract solutions for implementing schemes such as BFV and CKKS.It is very important to be familiar with areas such as mathematics, especially ring theory. With thistogether, CRT (Chinese Remaider Theorem), NTT (Number Theoretic Transform) and FTT (FastIt is also necessary to know and apply operations such as Fourier Transform. Take a look at these issuesThinking that it would be useful to discuss these topics, I wrote brief explanations about these issues below.

Ring Theory- Quotient Ring – Modüler Arithmetic

Ring theory is a field in abstract mathematics that specifically studies the properties of rings. a ring, with addition and multiplication operations, closed, unit element, inverse elements for addition and is a set with inverses of non-zero elements for multiplication.

Quotient ring is a ring consisting of a special subset of a ring called an ideal. It is a structure created under a specific process. An ideal is a special set of elements within a given ring. It represents the subset containing a combination of Quotient ring is the representation of an ideal on a ring.
It is a derived structure created with all its elements. This structure is "divided" by a certain ideal or can be thought of as "modded".

For example, we can construct a modular arithmetic quotient ring over integers. Here, 6 If we choose an ideal such as multiples, we will be dividing by 6 on integers. This In this case, all numbers divisible by 6 form an equivalence class, and these classes. The structure it creates represents modular arithmetic according to 6. This is an example of a quotient ring because A modulated structure was created based on a certain ideal (multiples of 6).

Quotient ring may come into play here; It provides a mathematical basis for homomorphic encryption.
For example, functions used for homomorphic encryption can be used to collect or It can perform operations such as multiplying. These transactions are encrypted based on the quotient ring structure performed on the data. This means performing mathematical operations on encrypted data.
and it is a way to get the results right.

Quotient ring, a basis for performing mathematical operations for homomorphic encryption
provides. In this way, transactions can be made on encrypted data and data kept in encrypted form
processing is possible.

FFT-NTT-CRT

FFT (Fast Fourier Transform), CRT (Chinese Remainder Theorem) and NTT (Number
Theoretic Transform) are important mathematical tools in homomorphic encryption processes.

FFT is a fast calculation algorithm used to separate a signal into its frequency components.
It is used in the encryption process, especially in the CKKS (Cheon-Kim-Kim-Song) encryption scheme.
CKKS is a scheme used for homomorphic encryption and performs mathematical operations.
It uses FFT to achieve this. This scheme is particularly useful for multiplication operations between numbers.
It often uses the FFT algorithm to perform.

CRT demonstrates modular division of a number and the use of the remainder theorem in modular arithmetic.
It is a theorem that provides. In the context of homomorphic encryption, we generally use a larger module. It is used to divide the data into small modules and spread the operations over these parts. Especially BFV , It is used in the (Brakerski-Fan-Vercauteren) encryption scheme. This diagram is done by breaking the module into parts and It is used for homomorphic encryption by performing operations on these subparts.

NTT is an algorithm that enables the conversion of numbers over a domain. homomorphicin the context of encryption, especially in some encryption schemes (e.g. BFV) consisting of polynomialsIt is used to transform data and perform operations. NTT converts data from polynomial coefficientsIt enables mathematical operations to be performed by converting the data into roots.These algorithms are used in performing mathematical operations in the homomorphic encryption process.is used.

While FFT and NTT are used to operate on polynomials, CRT is largerto divide modules into smaller parts and perform operations on modular arithmeticis used. These algorithms combine mathematical calculations in certain steps of homomorphic encryption protocols.plays important roles in the execution of transactions

Coding of BFV Scheme:

BFV scheme is a technique used for homomorphic encryption.

The encryption process follows the following steps:

• The message is multiplied by the scaling factor.
• Random vectors are generated and error vectors are calculated.
• The ciphertext is created. This is done by combining error vectors and random vectors with certain operations. Generally, in BFV-like schemes, a ciphertext resulting from encryption may consist of two components.

These components contain the coefficients of polynomials and are based on different calculations.

• c0: Usually represents the main component of the encrypted message. It is the main building block of the encrypted version of the original message.
• c1: This component forms another part of the encrypted message. This component may change as a result of different processes.

In some encryption schemes, this component may represent an additional operation or another contribution to the encryption process. Both of these components represent different aspects of the encrypted message and are used during the decryption process. Which component represents what may depend on the configuration and algorithm of the encryption scheme.

The decryption process includes the following steps:

• An intermediate message is created from the ciphertext. This step is achieved by combining the ciphertext and the secret key with certain operations.
• The plaintext is obtained by dividing the intermediate message by the scaling factor. This results in the original message being rescaled.
• The resulting message is rounded and retrieved according to the plaintext module.

Operations:

- **Add**: Adds two ciphertexts. Related Returns the sum of two ciphertexts in the context.
- **Multiply**: Multiplies two ciphertexts and relinearizes them. Returns the multiplication and relinearized version of two ciphertexts in the relevant context.
- **Subtract**: Subtracts the other from one ciphertext. Returns the result by subtracting the other from one ciphertext in the relevant context.
- **Relinearize**: It reduces a three-dimensional ciphertext to two dimensions. In the relevant context, it relinearizes a three-dimensional ciphertext and makes it two-dimensional.

Add Function :

```
def add(self, ciph1, ciph2):
    """Adds two ciphertexts.

    Adds two ciphertexts within the context.

    Args:
        ciph1 (Ciphertext): First ciphertext.
        ciph2 (Ciphertext): Second ciphertext.

    Returns:
        A Ciphertext which is the sum of the two ciphertexts.
    """
    assert isinstance(ciph1, Ciphertext)
    assert isinstance(ciph2, Ciphertext)

    new_ciph_c0 = ciph1.c0.add(ciph2.c0, self.coeff_modulus)
    new_ciph_c1 = ciph1.c1.add(ciph2.c1, self.coeff_modulus)
    return Ciphertext(new_ciph_c0, new_ciph_c1)
```

Multiply Function:

```
def multiply(self, ciph1, ciph2, relin_key):
    """Multiplies two ciphertexts.

    Multiplies two ciphertexts within the context, and relinearizes.

    Args:
        ciph1 (Ciphertext): First ciphertext.
        ciph2 (Ciphertext): Second ciphertext.
        relin_key (RelinKey): Relinearization keys.

    Returns:
        A Ciphertext which is the product of the two ciphertexts.
    """
    assert isinstance(ciph1, Ciphertext)
    assert isinstance(ciph2, Ciphertext)

    c0 = ciph1.c0.multiply_fft(ciph2.c0)
    c0 = c0.scalar_multiply(1 / self.scaling_factor)
    c0 = c0.round().mod(self.coeff_modulus)

    c1 = ciph1.c0.multiply_fft(ciph2.c1).add(ciph1.c1.multiply_fft(ciph2.c0
    c1 = c1.scalar_multiply(1 / self.scaling_factor)
    c1 = c1.round().mod(self.coeff_modulus)

    c2 = ciph1.c1.multiply_fft(ciph2.c1)
    c2 = c2.scalar_multiply(1 / self.scaling_factor)
    c2 = c2.round().mod(self.coeff_modulus)

    return self.relinearize(relin_key, c0, c1, c2)
```

Subtraction Function:

```
def subtract(self, ciph1, ciph2):
    """Subtracts ciph2 from ciph1.

    Subtracts one ciphertext from another within the context.

    Args:
        ciph1 (Ciphertext): The minuend ciphertext.
        ciph2 (Ciphertext): The subtrahend ciphertext.

    Returns:
        A Ciphertext which is the result of subtracting ciph2 from ciph1.
    """
    assert isinstance(ciph1, Ciphertext)
    assert isinstance(ciph2, Ciphertext)

    new_ciph_c0 = ciph1.c0.subtract(ciph2.c0, self.coeff_modulus)
    new_ciph_c1 = ciph1.c1.subtract(ciph2.c1, self.coeff_modulus)
    return Ciphertext(new_ciph_c0, new_ciph_c1)
```

BFV Scheme Examples and Tests

In order to run examples of the BFV scheme, we first need to enter the polynomial degree, plaintext mode value and ciphertext mode value.

degree: This represents the polynomial degree. It refers to the degree of polynomial to be used in the BFV scheme. Encryption and decryption processes are related to this degree. For example, it specifies the degree of polynomial in which an encrypted message will be represented.

plain_modulus: This is the plaintext modulus used during encryption. In the BFV scheme, this module is usually a small prime where messages are processed before being encrypted. This value affects the size and configuration of the encrypted text.

ciph_modulus: This is the ciphertext module used during encryption. This module affects the size of the encrypted text and the security of encryption operations. This number is usually a large prime number and is important for the size and security of the encrypted text. This determines the size of the encrypted message.

After these parameters are created, the necessary objects of the classes are created.

BFV scheme can be obtained by polynomially encoding integer vectors. I also converted the same vectors into polynomial plaintexts by polynomial encoding and wrote a test function in which I ran the same add, multiply and subtact functions on these samples and recorded the results and times of these functions in a table.

First, I entered degree, polynomial modulus and ciphertext modulus values.

```
        self.degree = int(arg)
        self.plain_modulus = 17
        self.ciph_modulus = 8000000000000
        self.params = BFVParameters(poly_degree=self.degree,
                                    plain_modulus=self.plain_modulus,
                                    ciph_modulus=self.ciph_modulus)
```

The test function of an example “add” operation that I wrote in the test section. Other subtract and multiply functions are in exactly the same format.

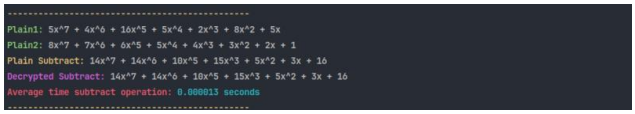
```
    1 usage
    def run_test_add(self, message1, message2):
        poly1 = Polynomial(self.degree, message1)
        poly2 = Polynomial(self.degree, message2)
        plain1 = Plaintext(poly1)
        print(Fore.GREEN+Style.BRIGHT+"Plain1:", Style.BRIGHT+str(plain1))
        plain2 = Plaintext(poly2)
        print(Fore.GREEN+Style.BRIGHT+"Plain2:", Style.BRIGHT+str(plain2))
        plain_add = Plaintext(poly1.add(poly2, self.plain_modulus))
        ciph1 = self.encryptor.encrypt(plain1)
        ciph2 = self.encryptor.encrypt(plain2)
        start_time = time.perf_counter()
        ciph_add = self.evaluator.add(ciph1, ciph2)
        total_time = time.perf_counter() - start_time
        decrypted_add = self.decryptor.decrypt(ciph_add)
        print(Fore.YELLOW + Style.BRIGHT + "Plain Add:", Style.BRIGHT + str(plain_add))
        print(Fore.MAGENTA + Style.BRIGHT + "Decrypted Add:", Style.BRIGHT + str(decrypted_add))
        self.assertEqual(str(plain_add), str(decrypted_add))
        return total_time
```

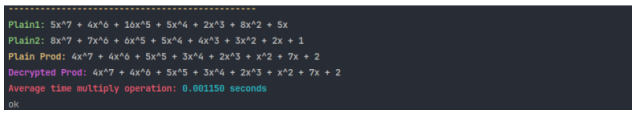
```
def test_evaluator_time(self):
    self.params.print_parameters()

    total_time = 0
    vec1 = [0, 5, 8, 2, 5, 10, 4, 5]
    print(Fore.BLUE + Style.BRIGHT + "Vector 1: ", Style.BRIGHT + str(vec1))
    vec2 = [1, 2, 3, 4, 5, 6, 7, 8]
    print(Fore.BLUE + Style.BRIGHT + "Vector 2: ", Style.BRIGHT + str(vec2))
    total_time += self.run_test_add(vec1, vec2)
    print(Fore.RED + Style.BRIGHT + "Average time add operation: %s%f seconds%s" % (
        Fore.CYAN + Style.BRIGHT, total_time, Fore.RESET))
    print(Fore.YELLOW + Style.BRIGHT + "-----------------------------------------------")
    total_time=0
    total_time += self.run_test_subtract(vec1, vec2)
    print(Fore.RED + Style.BRIGHT + "Average time subtract operation: %s%f seconds%s" % (
        Fore.CYAN + Style.BRIGHT, total_time, Fore.RESET))
    print(Fore.YELLOW + Style.BRIGHT + "-----------------------------------------------")
    total_time=0
    total_time += self.run_test_multiply(vec1, vec2)
    print(Fore.RED + Style.BRIGHT + "Average time multiply operation: %s%f seconds%s" % (
        Fore.CYAN + Style.BRIGHT, total_time, Fore.RESET))
```

Outputs of these tests;

```
simay@simay-ABRA-A5-V16-4:~/PycharmProjects/HomomorphicEncryption$ python3 tests/bfv/bfv_performance.py TestEvaluator 8
test_evaluator_time (__main__.TestEvaluator) ... Encryption parameters
        polynomial degree: 8
        plaintext modulus: 17
        ciphertext modulus size: 42 bits
Vector 1:  [0, 5, 8, 2, 5, 10, 4, 5]
Vector 2:  [1, 2, 3, 4, 5, 6, 7, 8]
Plain1: 5x^7 + 4x^6 + 10x^5 + 5x^4 + 2x^3 + 8x^2 + 5x
Plain2: 8x^7 + 7x^6 + 6x^5 + 5x^4 + 4x^3 + 3x^2 + 2x + 1
Plain Add: 13x^7 + 11x^6 + 5x^5 + 10x^4 + 6x^3 + 11x^2 + 7x + 1
Decrypted Add: 13x^7 + 11x^6 + 5x^5 + 10x^4 + 6x^3 + 11x^2 + 7x + 1
Average time add operation: 0.000014 seconds
```





I recorded the time values of the previous stage and the time values of the new stage in the table.

I analyzed how the results changed by entering different "degree", "plaintext" and "ciph modulus" values into the test functions I wrote. For example;

- Same vector , same plaintext modulus , different ciph modulus
- Same vector, modulus same ciph modulus , different plaintext
- Different length vector, same plaintext, same modulus

BFV Scheme Examples

(seconds)	ADD	ADD	MULTIPLY	MULTIPLY	SUBTRACT	SUBTRACT
DIFFERENT CIPHERTEXT MODULUS (42 BITS VS 208 BITS)	0.000010	0.000011	0.002291	0.001988	0.000022	0.000016
DIFFERENT PLAINTEXT MODULUS (16 VS 256)	0.000026	0.000022	0.001900	0.004569	0.000011	0.000039
DIFFERENT VECTOR LENGTH (8 VS 16)	0.000008	0.000023	0.000892	0.005062	0.000010	0.000031

The cryptotext module determines the length of encrypted data. The table shows the results for the 42-bit and 208-bit cryptotext modules. Larger cryptotext modules require more calculations, leading to slower performance.

The plaintext module determines the length of the plaintext. The table shows the results for the 16-bit and 256-bit plaintext modules. Larger plaintext modules require more computation, leading to slower performance.

Vector length determines the length of vectors on which the arithmetic operation will be performed. The table shows the results for 8-bit and 16-bit vector lengths. Longer vectors require more calculations, leading to slower performance.

The arithmetic operation determines the type of calculation to be performed. The table shows the performance of addition, multiplication, subtraction and division operations. Addition and subtraction are faster than multiplication and division.

Overall, the table shows that the performance of BFV slows down as the module size, plaintext module, and vector length increase. The type of arithmetic operation does not significantly affect performance.

These results can be concluded that BFV is suitable for use in secure calculation of sensitive data but may have slower performance for larger module sizes, plaintext modules or vector lengths.

To evaluate the suitability of BFV for a particular application, the calculation time needs to be compared with the requirements of the application.

Coding of CKKS Scheme

The CKKS (Cheon-Kim-Kim-Song) encryption scheme is a scheme used for homomorphic encryption and is specifically designed for processing digital data. In the CKKS scheme, real numbers can be treated homomorphically and signed integers are represented by these real numbers. Integers are first converted to real numbers in the CKKS scheme. This conversion is done to represent the numbers in the plaintext space in the encrypted space. This process is normally accomplished through an encoding step and usually involves some form of scaling and rounding. In summary, in the CKKS scheme, integers are encoded in the following steps:

1. Scaling: Integers are scaled to a certain range (for example, between -1 and 1). This scaling process is done to increase the accuracy and efficiency of homomorphic calculations.

2. Rounding: Since scaled integers are represented by real numbers, rounding can be applied. This helps control precision when converting scaled numbers to real numbers.

3. Conversion to Real Numbers: In this step, scaled and rounded integers are converted to real numbers. These real numbers are represented in the encrypted space where homomorphic operations can be performed.

These operations ensure that the integers in the plaintext space are represented by real numbers in the encrypted space. In this way, while operations are performed under homomorphic encryption, operations can be performed on integers through encrypted real numbers. After these operations, the results are decoded back and the real numbers obtained in the encrypted space are converted back into integers and obtained in the clear text space.

CKKS Scheme Examples and Tests

First, the necessary attributes are defined and the necessary objects are defined:

```
def main():

    poly_degree = 8
    ciph_modulus = 1 << 600
    big_modulus = 1 << 1200
    scaling_factor = 1 << 30
    params = CKKSParameters(poly_degree=poly_degree,
                            ciph_modulus=ciph_modulus,
                            big_modulus=big_modulus,
                            scaling_factor=scaling_factor)
    key_generator = CKKSKeyGenerator(params)
    public_key = key_generator.public_key
    secret_key = key_generator.secret_key
    relin_key = key_generator.relin_key
    encoder = CKKSEncoder(params)
    encryptor = CKKSEncryptor(params, public_key, secret_key)
    decryptor = CKKSDecryptor(params, secret_key)
    evaluator = CKKSEvaluator(params)
```

Then, vectors created from real numbers are defined and these vectors must first be encoded according to the ckks scheme. ckks_encoder is used for this: the encode function takes a list of complex numbers called values and a scaling_factor.

1. The num_values variable specifies the number of elements in the values list.

2. Plain_len uses twice num_values to calculate the length of the polynomial to be encoded. Because complex numbers consist of two components (real and imaginary part).

3. To_scale scales given complex numbers using an embedding process. Here complex numbers are handled as a result of a special operation.

4. Multiplication is done with scaling_factor and the real and imaginary parts are handled separately. These values are placed in the message list to form the coefficients of the polynomial.

5. The message list, which is the result of these operations, is placed into a Polynomial object to create a Plaintext object. This created Plaintext object can be used for encryption.

Then we need to encrypt these encoded plaintexts. The deckks_encryptor scheme is used for this. The encryption steps seem to be processed as follows:

1. The encrypt function takes a plain text for the encryption process.

2. The values p0 and p1 appear to be polynomials used as part of the encryption key.

3. Random_vec is initialized by generating a random polynomial. This polynomial looks like a random vector to be used in the encryption process.

4. error1 and error2 are initialized again by creating random polynomials. These polynomials are used to represent errors that may occur during the encryption process.

5. While creating c0 and c1, random_vec vectors are multiplied by p0 and p1. These multiplication operations are one of the basic operations to be used in the encryption process.

6. The error1 and error2 polynomials are added to the generated multiplication results. This step represents errors that may occur during the encryption process.

7. The plain value is added to c0. This allows the cleartext to be appended to the encrypted data.

8. Finally, operations are performed with the modulus value (self.coeff_modulus). These steps ensure that certain limits and rules are maintained during the encryption process.9.

The created polynomials c0 and c1 are placed into a Ciphertext object together with the plain's scaling factor and module to obtain encrypted data. These steps seem to form the building blocks of the homomorphic encryption process under the CKKS scheme. This scheme is a method used for homomorphic calculations, especially while preserving numerical precision and used in conjunction with analysis operations.

Example add operation steps (Other operations are in same manner) :

```
message1 = [0.5, 0.3 + 0.2j, 0.78, 0.88j]
message2 = [0.2, 0.11, 0.4 + 0.67j, 0.9 + 0.99j]

plain1 = encoder.encode(message1, scaling_factor)
print(Fore.GREEN + Style.BRIGHT + "Polynomial plaintext of message1 : ", Style.BRIGHT + str(plain1))
plain2 = encoder.encode(message2, scaling_factor)
print(Fore.GREEN + Style.BRIGHT + "Polynomial plaintext of message2 : ", Style.BRIGHT + str(plain2))
decoded_plain1 =encoder.decode(plain1)
decoded_plain2 = encoder.decode(plain2)
print(Fore.BLUE+Style.BRIGHT+"Reel Numbers plaintext decoded of message1:"_, Style.BRIGHT+str(decoded_plain1))
print(Fore.BLUE+Style.BRIGHT+"Reel Numbers plaintext decoded of message2:"_, Style.BRIGHT+str(decoded_plain2))
result = [x + y for x, y in zip(message1, message2)]
print(Fore.LIGHTYELLOW_EX + Style.BRIGHT + "Expected Result is:"_ Style.BRIGHT + str(result))
ciph1 = encryptor.encrypt(plain1)
print(Fore.MAGENTA + Style.BRIGHT + "CipherText of message1:", Style.BRIGHT + str(ciph1))
ciph2 = encryptor.encrypt(plain2)
print(Fore.MAGENTA + Style.BRIGHT + "CipherText of message1:", Style.BRIGHT + str(ciph2))
start_time = time.perf_counter()
ciph_add = evaluator.add(ciph1, ciph2)
total_time = time.perf_counter() - start_time
print(Fore.RED + Style.BRIGHT + "CipherText of message1+message2: ", Style.BRIGHT + str(ciph_add))
decrypted_add= decryptor.decrypt(ciph_add)
print(Fore.LIGHTGREEN_EX + Style.BRIGHT + "CiphertText to Polynomial PlainText: "_Style.BRIGHT + str(decrypted_add))
decoded_add = encoder.decode(decrypted_add)
print(Fore.LIGHTYELLOW_EX + Style.BRIGHT + "Reel Numbers PlainText Result :", Style.BRIGHT + str(decoded_add))
print(Fore.RED + Style.BRIGHT + "Total Time of Add:", total_time)
```

Output of Add Operation:

Output of Subtraction Operation:

Output of Multiply Operation:

I analyzed how the results changed by entering different "degree", "scaling_factor" and "ciph modulus" values into the test functions I wrote. For Exampe;

- Same vector , same scaling factor , different ciph modulus
- Same vector , same ciph modulus , different scaling factor
- Different vector , same ciph modulus , same scaling factor

Table Output:

CKKS Scheme Examples

(seconds)	ADD	ADD	MULTIPLY	MULTIPLY	SUBTRACT	SUBTRACT
DIFFERENT CIPHERTEXT MODULUS (601 BITS VS 2401 BITS)	0.000019633	0.00005682799	0.0508304	0.056431	0.000022209	0.00004419699
DIFFERENT SCALING FACTOR (31 BITS VS 201 BITS)	0.000036571	0.000017728	0.053671	0.050562	0.000021714	0.000021041
DIFFERENT VECTOR LENGTH (8 VS 16)	0.000020582	0.000042862	0.058830	0.099897	0.000023554	0.000033900

The ciphertext module determines the length of the encrypted data. The table shows the results for the 601-bit and 2401-bit ciphertext modules. Larger ciphertext modules require more calculations, leading to slower performance.

The plaintext module determines the length of the plaintext. The table shows the results for the 31-bit and 201-bit plaintext modules. Larger plaintext modules require more computation, leading to slower performance.

Vector length determines the length of vectors on which the arithmetic operation will be performed. The table shows the results for 8-bit and 16-bit vector lengths. Longer vectors require more calculations, leading to slower performance.

The arithmetic operation determines the type of calculation to be performed. The table shows the performance of addition, multiplication, subtraction and division operations. Addition and subtraction are faster than multiplication and division.

Overall, the table shows that the performance of CKKS slows down as the module size, plaintext module, and vector length increase. The type of arithmetic operation does not significantly affect performance.

These results can be concluded that CKKS is suitable for use in secure calculation of sensitive data but may have slower performance for larger module sizes, plaintext modules or vector lengths.

To evaluate the suitability of CKKS for a particular application, the calculation time needs to be compared with the requirements of the application.

In addition to the results given in the table, other factors that can affect the performance of CKKS include:

Cryptographic parameters: The CKKS scheme has a number of cryptographic parameters that affect encryption strength and computational efficiency. These parameters should be set according to the requirements of the application.

Hardware: CKKS calculations are typically performed more efficiently on dedicated hardware such as GPUs.
CKKS is a powerful and efficient scheme for homomorphic encryption. However, its performance must be carefully evaluated to meet the requirements of the application.

GENERAL COMPARISONS AND ANALYSIS OF BFV AND CKKS SCHEMES:

BFV Scheme VS CKKS Scheme

	BFV	CKKS
MEMORY USAGE	Lower memory requirement	Higher memory requirement
SENSITIVITY CONTROL	Lower sensitivity control	Higher sensitivity control
SECURITY LEVEL	High security level	Low security level
EFFICIENY COMPARISON	Higher processing speed	Slower processing speed
LITERATURE COMPARISON	More common use	Newly Developed Scheme

V. CONCLUSION

In conclusion, this report navigates the complex landscape of homomorphic encryption, elucidating its fundamental principles and illustrating its transformative potential. The exploration of BFV and CKKS homomorphic encryption schemes underscores their distinct attributes in terms of computational efficiency and applicability. The comparative analysis exposes the trade-offs between performance metrics, highlighting BFV's advantages in certain scenarios and CKKS's proficiency in others. The insights gleaned from this comparative study offer valuable guidance for practitioners and researchers seeking to leverage homomorphic encryption in real-world applications. As this technology continues to evolve, further research and innovation are crucial to harness its

full capabilities while addressing practical challenges. Homomorphic encryption stands poised as a cornerstone in secure computation, promising unprecedented privacy-preserving solutions across diverse domains.

VI. RERERENCES

[1] Jing Ma, Si-Ahmed Naas, Stephan Sigg, Xixiang Lyu(2022), Privacypreserving federated learning based on multi-key homomorphic encryption,Internatonal Journal of ıntellıgent systems,(5880-5901) (https://1551683842c92c442f30948395dc2e4ae6a00da3.vetisonline.com/doi/full/10.1002/int.22818)(https://doi.org/10.1002/int.22818)

[2] Paulo Martins and Leonel Sousa, Artur Mariano(2017), a survey on fully homomorphic encryption: an engineering perspective, acm computing surveysvolume 50issue 6article no.: 83pp 1–33 (https://224b1d915ba5a6eb7905a118ac41b5953093a960.vetisonline.com/doi/10.1145/3124441)

[3] Zainab Hikmat Mahmood; Mahmood Khalel Ibrahem,(2018), new fully homomorphic encryption scheme based on multistage partial homomorphic encryption applied in cloud computing, 1st annual international conference on information and sciences (aicis), ieee, doi: 10.1109/aicis.2018.00043,(https://ieeexplore.ieee.org/document/8640952)

[4] Dharani D; Anitha Kumari K.; Rohit M (2023), Research on Homomorphic Encryption for Arithmetic of Approximate Numbers, ieee, 2023 International Conference on Intelligent Systems for Communication, IoT and Security (ICISCoIS)), DOI: 10.1109/ICISCoIS56541.2023.10100464 (https://ieeexplore.ieee.org/document/10100464)

[5] H Manohar Reddy; Sajimon P C; Sriram Sankaran(2022), on the feasibility of homomorphic encryption for internet of things, ieee 8th world forum on internet of things (wf-iot), doi: 10.1109/wfiot54382.2022.10152214,(https://ieeexplore.ieee.org/document/10152 214)

[6] Shereen Mohamed Fawaz , Nahla Belal , Adel ElRefaey , Mohamed Waleed Fakhr , A Comparative Study of Homomorphic Encryption SchemesUsing Microsoft SEAL,2021,Journal of Physics: Conference Series , doi:10.1088/1742-6596/2128/1/012021

[7] Ung Hee Cheon, Andrey Kim, Miran Kim, Yongsoo Song, Homomorphic Encryption for Arithmetic of Approximate Numbers,Seoul National University, Republic of Korea,2016

[8] Yancho B. Wiryen , Noumsi Woguia Auguste Vigny, Joseph Mvogo Ngono, Louis Aime Fono, A Comparative Study of BFV and CKKs Schemes to Secure IoT Data Using TenSeal and Pyfhel Homomorphic Encryption Libraries, November 2023, International Journal of Smart Security Technologies 10(1):1-17, DOI:10.4018/IJSST.333852

[9] Lei Jiang, Lei Ju, FHEBench: Benchmarking Fully Homomorphic Encryption Schemes,2022, https://doi.org/10.48550/arXiv.2203.00728

[10] F. Bergamaschi, “HElib: an Implementation of homomorphic encryption,” https://github.com/homenc/HElib, 2019.

[11] https://www.inferati.com/blog/fhe-schemes-bfv

[12] https://github.com/sarojaerabelli/py-fhe

[13] https://github.com/acmert/bfv-python

[14] https://github.com/wgxli/simple-fhe