International Journal of Scientific Research in Computer Science, Engineering and Information Technology
ISSN : 2456-3307 (www.ijsrcseit.com)
doi : https://doi.org/10.32628/CSEIT217252

Modified Fully Homomorphic Encryption based on Parallel Processing in Cloud Computing

Parth Tandel*, Abhinav Shubhrant, Mayank Sohani
Mukesh Patel School of Technology Management and Engineering, Shirpur, Maharashtra, India

Article Info
Volume 7, Issue 2
Page Number: 250-262

Publication Issue :
March-April-2021

Article History
Accepted : 05 April 2021
Published : 11 April 2021

ABSTRACT

Cloud Computing is widely regarded as the most radically altering trend in information technology. However, great benefits come with great challenges, especially in the area of data security and privacy protection. Since standard cloud computing uses plaintext, certain encryption algorithms were implemented in the cloud for security reasons, and ‘encrypted' data was then stored in the cloud. Homomorphic Encryption (HE), a modern kind of encryption strategy, is born as a result of this change. Primarily, the paper will focus on implementing a successful Homomorphic Encryption (HE) scheme for polynomials. Furthermore, the objective of the paper is to propose, produce and implement a method to convert the already implemented sequentially processing Homomorphic Encryption into parallel processing Homomorphic Encryption (HE) using a Parallel Processing concept (Partitioning, Assigning, Scheduling, etc) and thereby producing a better performing Homomorphic Encryption (HE) called Fully Homomorphic Encryption (FHE). Fully Homomorphic Encryption (FHE) is an encryption technique that can perform specific analytical operations, functions and methods on normal or encrypted data and can still perform traditional encryption results as performed on plaintext. The three major reasons for implementing Fully Homomorphic Encryption (FHE) are advantages like no involvement of third parties, trade-off elimination between privacy and security and quantum safety.

Keywords : Cloud Computing, Encryption, Homomorphic Encryption, Fully Homomorphic Encryption, Parallel Computing, Parallel Processing, Partitioning.

I. INTRODUCTION

Cloud Computing is a technology which lets smooth, voluntary (on-request) network access to shareable and configurable computing resources and data that can be provisioned, manipulated and delivered with the least amount of administration or cloud service provider’s involvement. Before getting into the security concerns, let us take a look at the three major services and benefits of Cloud Computing [1].

Software as a Service (SaaS) - It is called SaaS that distributes a software operated by third party companies, thus allowing users to access the software over the network. For instance, if a student needs office software like MS PowerPoint for a specific period, he / she does not have to purchase the whole product, rather he / she only has to pay for the software resources needed by the buyer. Example – Dropbox and Google Workspace.

Platform as a Service (PaaS) – In traditional terms, PaaS literally paves the way or offers software developers a forum to create their goods or services over the Internet or a network. For example, if a developer now uses MacOS and needs to operate in a Windows environment, then that platform is provided by CSP. Example – Microsoft Azure, GAE (Google App Engine) and AWS Elastic Beanstalk.

Infrastructure as a Service (IaaS) – This service facilitates virtual storage for the users. The data is actually stored in the Cloud Service Provider's servers. Since the corporate world is consuming a lot of data today, IaaS's use has increased extensively. Example – GCP (Google Cloud Platform/Google Compute Engine), Microsoft Azure and AWS (Amazon Web Services) [1].

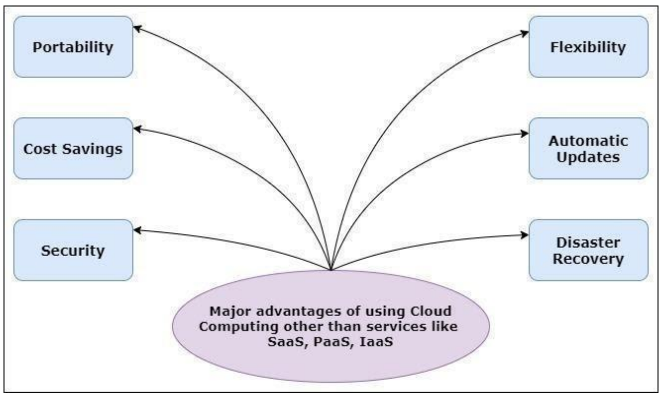

Portability
Flexibility
Cost Savings
Automatic Updates
Security
Disaster Recovery
Major advantages of using Cloud Computing other than services like SaaS, PaaS, IaaS

Fig. 1. Various advantages of using Cloud computing other than SaaS, PaaS, IaaS

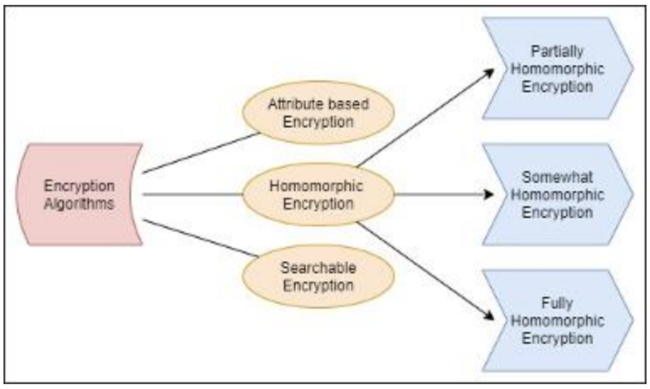
The major deal breaker when it comes to using a technology like Cloud Computing is Security. That is, the potential data breach by third-party vendors while putting sensitive data on the cloud. The repercussions of not having secured data: A chance of attack on the data either by simple manipulations or the whole chunk of data may be compromised [1]. Some other vital controls that harms Cloud Security w.r.t. compliance standards are given below [2]:

- Encryption of data and key administration
- Security of the media
- Recognize, authenticate and approve
- Virtualization and Resource Abstraction
- Interoperability and portability
- Security programme
- Identifying and managing security threats
- Anonymity, e-discovery and ethics
- Planning of emergencies
- Operations and maintenance of the Data Center
- Answer incident
- Enforcement, Transparency and Audit
- Awareness and Training

To remedy these issues, an obvious solution of cryptography was introduced to the cloud. Simply stating, Cryptography is the art of hiding any kind of information or data and keeping it secure and limited to approved eyes only. To successfully pull Cryptography, two techniques play a major role in it. Encryption converts different formats of data into unreadable format called ciphertext. Decryption is the other side of the coin which converts that ciphertext into original plain text [1].

II. ALGORITHM ELABORATION

2.1 Hybrid Homomorphic Encryption Scheme


Encryption Algorithms
Attribute based Encryption
Homomorphic Encryption
Searchable Encryption
Partially Homomorphic Encryption
Somewhat Homomorphic Encryption
Fully Homomorphic Encryption
Figure 2. General classification of Encryption Algorithms

A. HOMOMORPHIC ENCRYPTION ALGORITHM

To tackle the problem of data security, various types of encryption algorithms were brought to light. As data can be stored in an encrypted cloud form, an algorithm that could be performed on the encrypted data was needed. This has contributed to a Homomorphic Encryption technique.

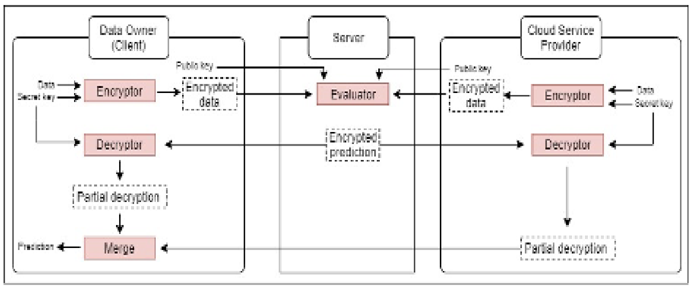


Figure 3. General working of Homomorphic Encryption

The big advantage of homomorphic encryption is the computing on encrypted data without knowledge of the private key, i.e. without decrypting it. Since the given data for computation is encrypted, the outcome of the calculations is encrypted as well. Also, the product of any computation or operation on the encrypted data mirrors the product of raw data perfectly [3].

Mathematical Representation – The system is Homomorphic encryption if Enc (a) and Enc (b) can calculate Enc (f (a, b)), where f can be: ADD, MULTIPLY, XOR

The Provider should have access to the secret key for data decryption if any type of computation is to be performed by the Client. However, sharing the key also grants the cloud provider access privileges. Homomorphic encryption is then used to overcome this problem so that the data can be computed without decrypting by cloud providers. In addition, as the client is the sole holder of a secret key, every other party cannot decode and access any data , the data will be returned in encrypted form [3].

The general processes involved in a Homomorphic Encryption system are described below:

1. Key Generation: The client generates the secret key Ks and public key Kp.

   $(K_s, K_p) = \text{keyGen}(s)$

2. Encryption: An encryption algorithm which uses the public key to encrypt plain text (M) and converts it to ciphertext (C).

   $C = \text{Enc}_{pk}(m)$

3. Evaluation: Use of the public key to apply function f to ciphertext c.

   $C* = \text{Eval}_{pk}(f, c)$.

4. Decryption: Decryption algorithm that retrieves plain text M with the ciphertext c and secret key.

   $M = \text{Dec}_{sk}(c)$

Homomorphic Encryption categorization: Homomorphic algorithm classification is performed according to the processes described above[3]:

- Partial Homomorphic Encryption (PHE): Allows an addition or multiplication of only one method for the encrypted data.
- Somewhat Homomorphic Encryption (SWHE): Allows multiplication and addition of more than one process, but the number of processes is limited.

- Fully Homomorphic Encryption (FHE): Enable multiple operations-multiplication and addition without restriction on the number of operations.
- Homomorphic encryption properties: The following properties shall be satisfied by a homomorphic encryption scheme:
- Additive homomorphism (AH): A homomorphic encryption is additive if,

$$Dec_{sk}(Enc_{pk}(M1) + Enc_{pk}(M2)) = M1 + M2 \qquad \textbf{.....(i)}$$

- Multiplicative homomorphism (MH): A homomorphic encryption is multiplicative if

$$Dec_{k}(C_{k}(M1) * C_{k}(M2)) = M1 * M2 \qquad \textbf{.....(ii)}$$

Now, when a Homomorphic algorithm satisfies both the above-mentioned properties simultaneously, it is known as Fully Homomorphic Encryption Algorithm [3].

B. FULLY HOMOMORPHIC ENCRYPTION ALGORITHM

This paper will discuss the most efficient form of homomorphic encryption, which is FHE. Initially, this technique was speculated by Rivest, Adleman and Dertouzous after the three of them put forward Privacy Homomorphism. In this proposal, they describe a way of facilitating h/w approach to achieve FHE. Consequently, this approach gave rise to performance problems. Later on, Craig Gentry used the approach of bootstrapping and ideal lattices to get the previously problematic full homomorphic encryption to work [4]. With any homomorphic encryption, FHE also enables operations on encrypted cloud data to provide users' sensitive data stored in cloud storage with data protection and confidentiality.

Let us discuss the cons for using FHE. In contrast, encryption takes more time and memory than unencrypted data takes for computing[5]. Besides time and memory, FHE also has some security issues and drawbacks like large key size and low calculation efficiency. Hence, the practical use of this encryption technique is kept limited.

III. RESEARCH AND IMPLEMENTATION

A. IMPLEMENTATION OF FULLY HOMOMORPHIC ENCRYPTION SCHEME

Cloud Environment

The console (PythonAnywhere) support versions2.7, 3.5, 3.6, 3.7 and 3.8 of Python, while the console also includes many useful libraries namely NumPy, SciPy, Mechanize, BeautifulSoup, Pycrypt, CherryPy, tweePy, GitPython, and many more.

The console supports all of Python's installs. The Console runs on servers hosted by Amazon EC2 so heavy duty processing is also possible. It also supports Simple Automation for running scheduled tasks on scripts periodically.

We will implement our Fully Homomorphic Encryption scheme on a Cloud Console that can run python scripts with numpy library. For that, we will create a random Cloud Console on the platform and create a python file for running in that console.

Cloud Console Details:-
Custom Console ID: 19645012
CPU Usage: 4% used – 4.27s of 100s.
File storage: 0% full – 108.0 KB of your 512.0 MB
Script Info: /home/parthtandel99/fhe.py
Batteries Included: NumPy

Basic Notations

Symbol	Description
Z_q	Integers between (−q/2,q/2], q>1 ,q∈Z (mod q)

$[.]_m$	Specify that we are applying modulo m
$[.]$	Rounding to the nearest integer
<a,b>	Inner product of two elements $a,b \in \mathbb{Z}^n_q$ and is defined as follows: $\langle a,b\rangle = \sum^{i}_{n} a_i \cdot b_i (\text{mod } q)$
x	x is a positive integer
v	$v \in \mathbb{Z}^n_q$ would be simply a vector of n elements in $\mathbb{Z}_q$

Table 1. Various notations used in the implementation [6],[7],[8],[9].

Additive and Multiplicative Properties

- Additive homomorphism (AH): A homomorphic encryption is additive if,

$Dec_{sk}(Enc_{pk}(M1) + Enc_{pk}(M2)) = M1 + M2$ **…..from (i)**

[1] Multiplicative homomorphism (MH): A homomorphic encryption is multiplicative if,

$Dec_k(C_k(M1) * C_k(M2)) = M1 * M2$ **….. from (ii)**

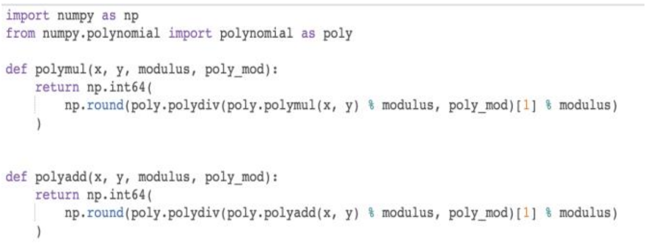
sk = Secret key pk = Public key
M = Plain text C = Ciphertext

Illustrations

a(x)=7x3+4x2+9 and b(x)=x3+10x2+3x+5
Therefore,
a(x)+b(x)=(7+1mod11)x3+(4+10mod11)x2+(3mod11)x+(9+5mod11)
a(x)=5x2+3 and b(x)=3x3+4x2
a(x)·b(x)=5x2·(3x3+4x2)+3·(3x3+4x2)
a(x)·b(x)=(15mod11)x5+(20mod11)x4+(9mod11)x3+(12mod11)x2

Implementation of helper functions

Let's begin with importing the awesome library of Numpy, then define 2 helper functions to add and multiply polynomials around a ring.
$R_q=\mathbb{Z}_q/\langle x^n+1\rangle$.



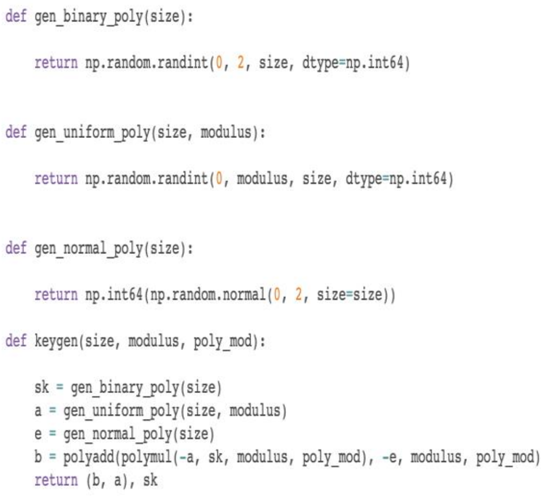
```
import numpy as np
from numpy.polynomial import polynomial as poly

def polymul(x, y, modulus, poly_mod):
    return np.int64(
        np.round(poly.polydiv(poly.polymul(x, y) % modulus, poly_mod)[1] % modulus)
    )


def polyadd(x, y, modulus, poly_mod):
    return np.int64(
        np.round(poly.polydiv(poly.polyadd(x, y) % modulus, poly_mod)[1] % modulus)
    )
```

Generation of Key

Initially, we will produce an arbitrary secret key from a probability distribution, we will use a normal uniform distribution over R2, that implies that sk is a polynomial with coefficients of zero or one. We first uniformly sample a polynomial over Rq for the public key and a minor error of a discrete normal distribution over Rq. Then we make the public key the tuple

$$pk=([-(a\cdot sk+e)]_q,a) \qquad \text{….. (iii)}$$

Now we will implement the production of polynomials from different and arbitray probability distributions [6],[7],[8],[9].



```
def gen_binary_poly(size):

    return np.random.randint(0, 2, size, dtype=np.int64)


def gen_uniform_poly(size, modulus):

    return np.random.randint(0, modulus, size, dtype=np.int64)


def gen_normal_poly(size):

    return np.int64(np.random.normal(0, 2, size=size))

def keygen(size, modulus, poly_mod):

    sk = gen_binary_poly(size)
    a = gen_uniform_poly(size, modulus)
    e = gen_normal_poly(size)
    b = polyadd(polymul(-a, sk, modulus, poly_mod), -e, modulus, poly_mod)
    return (b, a), sk
```

We can now use these functions to describe our main generator, i.e. (b, a) and sk.

Encryption

We would allow polynomials to be encrypted in the where t is the plaintext modulus. In our case, we want to encrypt integers in Z, so we'll only encode a plaintext integer as the constant m(x)=pt polynomial. The encryption algorithm takes a public key $pk \in R_q \times R_q$ and a plaintext polynomial $m \in R_t$ and produces a ciphertext $ct \in R_q \times R_q$, which is a tuple of two polynomials ct0 and ct1 [6],[7],[8],[9].

$$ct0=[pk0\cdot u+e1+\delta\cdot m]q \qquad \text{.....(iv)}$$

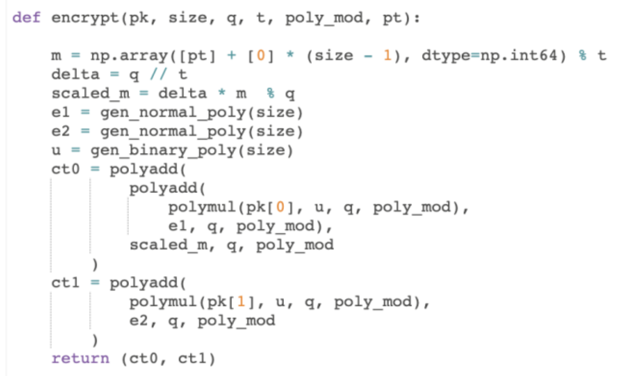
$$ct1=[pk1\cdot u+e2]q \qquad \text{.....(v)}$$

Where,

u is drawn from a uniform distribution over R2 (similar to the secret key),

e1 and e2 are drawn from a distinct normal distribution over Rq (similar to the error word in key generation) and

δ is used for rounding off to nearest integers with q over t.



```
def encrypt(pk, size, q, t, poly_mod, pt):

    m = np.array([pt] + [0] * (size - 1), dtype=np.int64) % t
    delta = q // t
    scaled_m = delta * m  % q
    e1 = gen_normal_poly(size)
    e2 = gen_normal_poly(size)
    u = gen_binary_poly(size)
    ct0 = polyadd(
            polyadd(
                polymul(pk[0], u, q, poly_mod),
                e1, q, poly_mod),
            scaled_m, q, poly_mod
        )
    ct1 = polyadd(
            polymul(pk[1], u, q, poly_mod),
            e2, q, poly_mod
        )
    return (ct0, ct1)
```
.

Decryption

Our scheme will support decryption of a ciphertext with arguments:-

sk: secret-key

size: size of polynomials

q: ciphertext modulus

t: plaintext modulus

poly_mod: polynomial modulus

ct: ciphertext

The decrypt function will return the integer representing the plaintext.

The assumption behind decryption is that $(pk1\cdot sk \approx -pk0)$ that is to say, they add up to a very small polynomial. Let's take a look at computing of $[ct0+ct1\cdot sk]q$:

$$[ct0+ct1\cdot sk]q=[pk0\cdot u+e1+\delta\cdot m+(pk1\cdot u+e2)\cdot sk]q$$

....from (iv) & (v)

$$[ct0+ct1\cdot sk]q=[pk0\cdot u+e1+\delta\cdot m+pk1\cdot sk\cdot u+e2\cdot sk]q$$

After expanding public key terms,

$$[ct0+ct1\cdot sk]q=[-(a\cdot sk+e)\cdot u+e1+\delta\cdot m+a\cdot sk\cdot u+e2\cdot sk]q$$

...from (iii)

$$[ct0+ct1\cdot sk]q=[-a\cdot sk\cdot u-e\cdot u+e1+\delta\cdot m+a\cdot sk\cdot u+e2\cdot sk]q$$

$$[ct0+ct1\cdot sk]q=[\delta\cdot m-e\cdot u+e1+e2\cdot sk]q \qquad \text{.....(vi)}$$

Only a scaled message and some error terms remain, multiplying by 1/δ,

$$1/\delta\cdot[ct0+ct1\cdot sk]q=[m+1/\delta\cdot errors]q$$

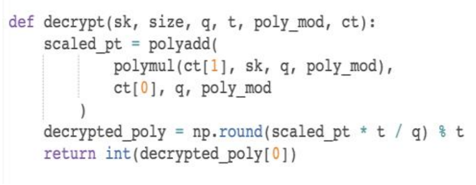
Rounding to the nearest integer and going back to Rt,

$$[\lfloor 1/\delta\cdot[ct0+ct1\cdot sk]q\rceil]t=[\lfloor [m+1/\delta\cdot errors]q\rceil]t$$

If the rounding to the nearest integer is not affected by the error terms, we will decrypt to the right value m, which means that the error terms must be bounded by 1/2.

$$1/\delta\cdot errors\leq 1/2 \Leftrightarrow errors\leq q/2t$$

So all those error terms must be within q/2t for a correct, efficient and error-free decryption.



```
def decrypt(sk, size, q, t, poly_mod, ct):
    scaled_pt = polyadd(
            polymul(ct[1], sk, q, poly_mod),
            ct[0], q, poly_mod
        )
    decrypted_poly = np.round(scaled_pt * t / q) % t
    return int(decrypted_poly[0])
```

After a basic encryption, one can conveniently select the parameters that ensure a proper decryption, but the goal of HE isn't only to encrypt/decrypt data, but also to compute on encrypted data (add and multiply). As a result, you can select your criteria based on your scheme, the level of protection you want to obtain, and the computation you want to execute [6],[7],[8],[9].

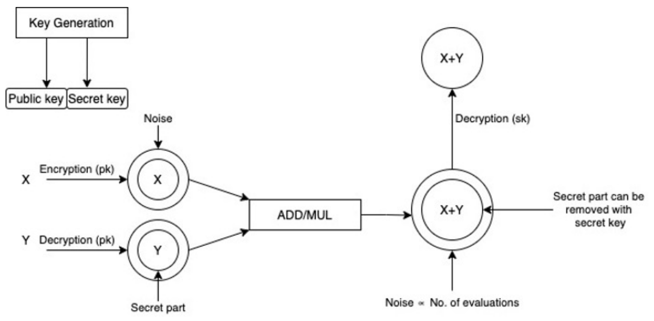


Figure 4. Framework of keys and encryption/decryption

We'll compute on the encrypted data now that we know how to produce keys, encrypt, and decrypt. With a ciphertext in hand, we may mix it with other ciphertexts or plaintexts to construct a new ciphertext. We'll introduce plain operations, which means we'll grant our scheme the power to add or multiply ciphertexts with integers (plaintexts) [6],[7],[8],[9].

Evaluation - Addition

Let us assume ct is a ciphertext encrypting a plaintext message m1,

$$ct=([pk0\cdot u+e1+\delta\cdot m1]q,[pk1\cdot u+e2]q)$$

....from (iv) & (v)

Adding m2 leaves us with a,

$$ct0=[pk0\cdot u+e1+\delta\cdot(m1+m2)]q$$

....from (iv)

Now, we require to scale new plaintext m2 by δ and add it to ct0.

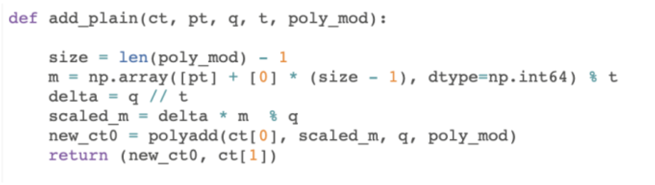
$$add\_plain(ct,m2)=([ct0+\delta\cdot m2]q,ct1)$$

This is how decryption will look like after the addition

$$[\lfloor 1/\delta\cdot[ct0+ct1\cdot sk]q\rceil]t=[\lfloor[m1+m2+1/\delta\cdot(-e\cdot u+e1+e2\cdot sk)]q\rceil]t$$

....from (iii), (iv), (v) & (vi)

As you might have noted, this procedure creates no additional noise, so we can execute as many basic additions as we like without incurring any sanction for noise.



```
def add_plain(ct, pt, q, t, poly_mod):

    size = len(poly_mod) - 1
    m = np.array([pt] + [0] * (size - 1), dtype=np.int64) % t
    delta = q // t
    scaled_m = delta * m  % q
    new_ct0 = polyadd(ct[0], scaled_m, q, poly_mod)
    return (new_ct0, ct[1])
```

Evaluation - Multiplication

Let us assume ct is a ciphertext encrypting a plaintext message m1 and after multiplying it with m2 we get,

$$ct,=[pk0\cdot u+e1+\delta\cdot m,\cdot m2]q$$

....from (iv) & (v)

Multiplying ct, with m2,

$$ct0\cdot m2=[pk0\cdot u\cdot m2+e1\cdot m2+\delta\cdot m1\cdot m2]q$$

....from (iv)

Expanding the public-key terms in [ct0+ct1·sk]q shows that the decryption will be incorrect.

$$[ct0+ct1\cdot sk]q=[pk0\cdot u\cdot m2+e1\cdot m2+\delta\cdot m1\cdot m2+pk1\cdot sk\cdot u+e2\cdot sk]q$$

....from (vi)

$$[ct0+ct1\cdot sk]q=[-(a\cdot sk+e)\cdot u\cdot m2+e1\cdot m2+\delta\cdot m1\cdot m2+a\cdot sk\cdot u+e2\cdot sk]q$$

$$[ct0+ct1\cdot sk]q=[-a\cdot sk\cdot u\cdot m2-e\cdot u\cdot m2+e1\cdot m2+\delta\cdot m1\cdot m2+a\cdot sk\cdot u+e2\cdot sk]q$$

Issue: –a·sk·u·m2 and a·sk·u won't cancel each other any more, and that's a major polynomial we've applied to our message. Decryption of m1*m2 would obviously fail. We'll just need to multiply ct1 by m2 for a proper decryption.

$$\text{mul\_plain}(ct,m2)=([ct0\cdot m2]q,[ct1\cdot m2]q)$$

Results,

$$[ct0+ct1\cdot sk]q=[-a\cdot sk\cdot u\cdot m2-e\cdot u\cdot m2+e1\cdot m2+\delta\cdot m1\cdot m2+a\cdot sk\cdot u\cdot m2+e2\cdot sk\cdot m2]q$$

....from (iii), (iv), (v) & (vi)

$$[ct0+ct1\cdot sk]q=[\delta\cdot m1\cdot m2-e\cdot u\cdot m2+e1\cdot m2+e2\cdot sk\cdot m2]q$$

Decryption circuit will result in,

$$[\lfloor 1/\delta\cdot[ct0+ct1\cdot sk]q\rceil]t=[\lfloor[m1\cdot m2+1/\delta\cdot(-e\cdot u\cdot m2+e1\cdot m2+e2\cdot sk\cdot m2)]q\rceil]t$$

Note: As compared to the plaintext addition, you can see that our error terms have been scaled up by our message m2, meaning that multiplying by large numbers can cause decryption to cause rounding errors.

```
def mul_plain(ct, pt, q, t, poly_mod):
    size = len(poly_mod) - 1
    # encode the integer into a plaintext polynomial
    m = np.array([pt] + [0] * (size - 1), dtype=np.int64) % t
    new_c0 = polymul(ct[0], m, q, poly_mod)
    new_c1 = polymul(ct[1], m, q, poly_mod)
    return (new_c0, new_c1)
```

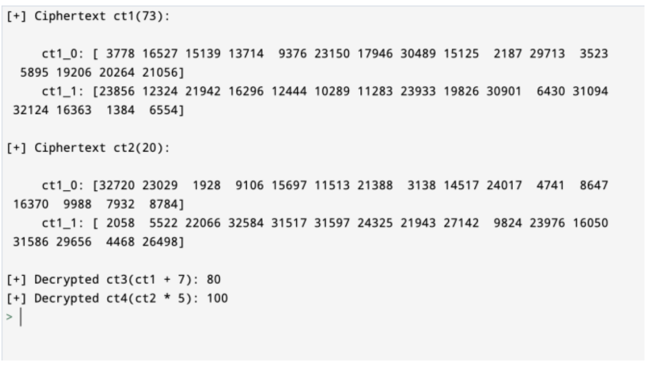
Now that all of the functionalities have been incorporated, let's put them all together.

```
n = 2**4
q = 2**15
t = 2**8
poly_mod = np.array([1] + [0] * (n - 1) + [1])
pk, sk = keygen(n, q, poly_mod)
pt1, pt2 = 73, 20
cst1, cst2 = 7, 5
ct1 = encrypt(pk, n, q, t, poly_mod, pt1)
ct2 = encrypt(pk, n, q, t, poly_mod, pt2)

print("[+] Ciphertext ct1({}):".format(pt1))
print("")
print("\t ct1_0:", ct1[0])
print("\t ct1_1:", ct1[1])
print("")
print("[+] Ciphertext ct2({}):".format(pt2))
print("")
print("\t ct1_0:", ct2[0])
print("\t ct1_1:", ct2[1])
print("")

ct3 = add_plain(ct1, cst1, q, t, poly_mod)
ct4 = mul_plain(ct2, cst2, q, t, poly_mod)
decrypted_ct3 = decrypt(sk, n, q, t, poly_mod, ct3)
decrypted_ct4 = decrypt(sk, n, q, t, poly_mod, ct4)

print("[+] Decrypted ct3(ct1 + {}): {}".format(cst1, decrypted_ct3))
print("[+] Decrypted ct4(ct2 * {}): {}".format(cst2, decrypted_ct4))
```

Thus, the Homomorphic Encryption is successfully implemented here [6],[7],[8],[9].

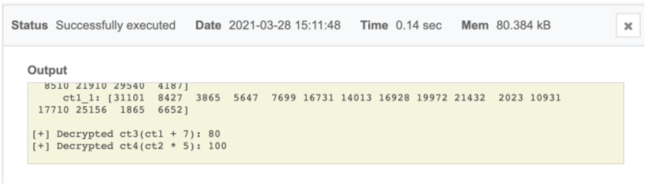
B. RESULTS AND ANALYSIS

Following snapshots are the results of the above implemented Fully Homomorphic Encryption Scheme.



```
[+] Ciphertext ct1(73):

     ct1_0: [ 3778 16527 15139 13714  9376 23150 17946 30489 15125  2187 29713  3523
  5895 19206 20264 21056]
     ct1_1: [23856 12324 21942 16296 12444 10289 11283 23933 19826 30901  6430 31094
 32124 16363  1384  6554]

[+] Ciphertext ct2(20):

     ct1_0: [32720 23029  1928  9106 15697 11513 21388  3138 14517 24017  4741  8647
 16370  9988  7932  8784]
     ct1_1: [ 2058  5522 22066 32584 31517 31597 24325 21943 27142  9824 23976 16050
 31586 29656  4468 26498]

[+] Decrypted ct3(ct1 + 7): 80
[+] Decrypted ct4(ct2 * 5): 100
> |
```

Figure 5. Actual Outcome of Integration Testing



Status Successfully executed **Date** 2021-03-28 15:11:48 **Time** 0.14 sec **Mem** 80.384 kB

Output

```
  8510 21910 29540  4187]
     ct1_1: [31101  8427  3865  5647  7699 16731 14013 16928 19972 21432  2023 10931
 17710 25156  1865  6652]

[+] Decrypted ct3(ct1 + 7): 80
[+] Decrypted ct4(ct2 * 5): 100
```

Figure 6. Actual Outcome of Integration Testing (with time and memory analysis)

Test Case : Encryption and Decryption of 73 and 20.
Expected Result: 80 (+7 because the addition property is evaluated as well) and 100 (*5 because the multiplication property is evaluated as well).
Actual Outcome - 80 and 100.

Since the multiplication property is satisfied as well, we can say that our Homomorphic scheme is a Fully Homomorphic Scheme.

Let us compare our execution time with the execution time of four widely used encryption algorithms called Data Encryption Standard (DES), Triple Encryption Data Standard (3DES), Rivest

Cipher (RC2/ARC2) and Advanced Encryption Standard or Rijndael.

Data Size = 4 bytes	
Algorithm	**Execution Time (seconds)**
DES	0.17
3DES	0.19
RC2	0.2
AES/Rijndael	0.16
FHE	0.14

Table 2. Tabular Representation of Execution Time of encryption algorithms with FHE[10]

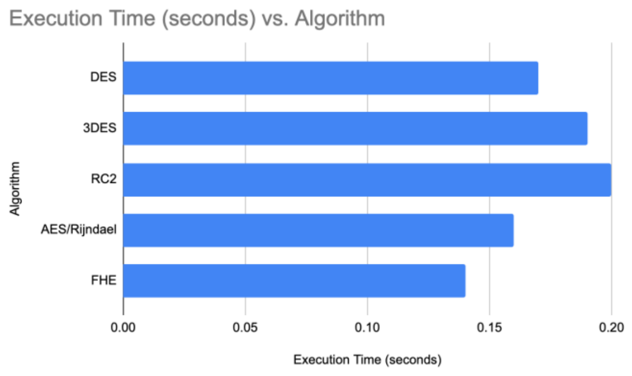


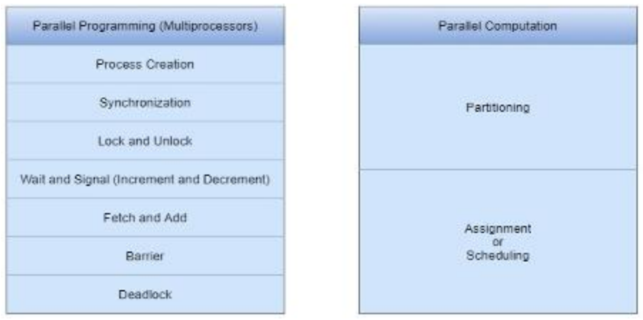
Figure 7. Execution Time of encryption algorithms with FHE

Please note that the difference is minimal because the data size is small as well. During practical implementation this minimal change will grow exponentially because of the bigger data size and therefore we can conclude with the result that FHE is better than most algorithms with reference to execution time.

C. PARALLEL PROCESSING AND PARTITIONING

This paper will specifically try to tackle the time and memory drawback of FHE. The main explanation is that FHE carries out cloud data (Fully homomorphic encryption) on one or more nodes and hence acquiring comparatively more time memory to compute than the one operation performed in simple text (unencrypted data). One of the obvious approaches to solve this issue would be Parallel Processing. It reduces processing time in cloud computing by imposing parallelism on encrypted data. As the name implies, parallel processing allows several nodes to work concurrently, so that the desired operation takes comparatively less time than the sequential process. Hence this approach will increase the performance of the traditional FHE [5].

Let 's speak about some parallel processing approaches. Conversion of a sequential algorithm to a parallel algorithm is one approach. If the problem already has a sequential algorithm, the inherent parallelism in the algorithm can be identified. It is a kind of parallelism which naturally occurs within an algorithm without special effort or algorithm shift. However, it is not fruitful or effective to use inherent parallelism in a sequential algorithm. A Parallel Algorithm to solve a similar but distinct problem is a safer and more successful solution for some problems. Another solution is the conception and substitution of a brand new parallel algorithm with the current algorithm.



Parallel Programming (Multiprocessors)
Process Creation
Synchronization
Lock and Unlock
Wait and Signal (Increment and Decrement)
Fetch and Add
Barrier
Deadlock

Parallel Computation
Partitioning
Assignment or Scheduling

Table 3. Tabular Representation of Parallel Programming Concepts

Also, the approach of Parallel processing can be used for Data partitioning method which tackles the security drawback of FHE. Client data can be separated into several sections of similarly sized chunks and stored on various servers. A public key is created on the client side to store and retrieve originally cloud-divided data. This can be accomplished by a parallel processing principle known as partitioning. The processing time is computed according to the time taken (on multiple nodes) to perform the operations applied simultaneously and to transmit the data. [5].

There are two methods of partitioning the task for parallel processing. They are : static partition and dynamic partition.

- Static Partitioning - The static partitioning technique separates tasks before execution and avoids conflict between processes. Input data then takes decisions including the exercise time and amount of data provided to systems, such as the amount of parallel calculation actually occurs. Therefore, during execution, certain processes cannot be kept working.
- Dynamic Partitioning - The dynamic partition approach divides up the tasks while running. As a consequence, systems are more busy and the input data isn't affected. The disadvantage is the amount of communication and coordination between processes that is required for implementation.
- Another classification of partitioning is done on the basis of processes.
- Data Partitioning (Data Parallelism) – The processes can be generated in such a way that every process carries out the same operation on several sections of the data. This partitioning is also called homogeneous multitasking because it creates multiple mirror processes. This approach derives parallelism from problem data organisation. Every part will be processed simultaneously with the data structure divided into pieces of information. An individual data item or the set of items may be a bit of information. In solving mathematical problems that handle large arrays and vectors, data partitioning is especially helpful. It is also useful for non-numerical topics, such as combinatorial search and sorting algorithms. Data Partitioning is especially suitable for creating multicomputer algorithms since a processor primarily calculates using its own local data and rarely with different processors.
- Function Partitioning (Control/Function Parallelism) - Processes may be generated to perform a different data operation, which is called Function Partitioning. This partitioning is known as heterogeneous multi-tasking, just as data partitioning is known as multitasking in a homogeneous manner, as many separate processes carry out various data tasks.
- Illustration of both the above discussed partitioning - Consider four vectors P, Q, S and T for the following computation.
  Y[n] = (P[n] / Q[n]) - (S[n] / T[n]), for n=1 to 5
- Data Partitioning - Five identical processes are generated such that n is run by each process. Thus, the computation of each of the Y[n] using several similar processes simultaneously was carried out in parallel.
- Function Partitioning – A and B are two processes which are derived and then we compute h = P[n] / Q[n] and sends the value of h to B. B then calculates f = S[n] / T[n] and inherits h from A for performing the calculation Y[n] = h + f. This is done for each index n. In this way, the functions of division are performed in parallel processing at the same time. This method typically organises the programme in such a way that the processes uses parallel processing while coding and not in the data.

Advantages of data partitioning over function partitioning:

1. More amount of parallel processing
2. Processes are given balanced/equal load
3. Subtle implementation

On the basis of Task Execution	On the basis of Process Creation
Static Partitioning	Data Partitioning
Dynamic Partitioning	Function Partitioning

Table 4. Classification of Partitioning

D. IMPLEMENTATION OF PARALLELISM

- The most common way to parallelize any operation is to take a feature that needs to be run several times and run it in parallel through several processors.
- To do so, build a Pool of n processors and call one of Pool's parallelization methods with the feature you want to parallelize.
- To execute all functions simultaneously, multiprocessing.Pool() includes the apply(), map(), and starmap() methods.
- The function to be parallelized is the key argument in both apply and map. However, apply() takes an args argument that includes the parameters transferred to the 'function-to-be-parallelized,' while map can only take as an argument, one iterable.

IV. CONCLUSION

In cloud computing, the power of computation and resources are much greater than just one computer where data calculations are carried out by computer clusters. The use of cloud storage to process large volumes of data is popular. For cloud computing, however, businesses are worried about data privacy. The privacy related problems in cloud related computing was described in this paper. Fully Homomorphic Encryption can be a resolution for solving data privacy in the cloud that processes information which is encrypted and returns the encrypted results. Beside so, generally Fully Homomorphic Encryption is slower and the faster schemes of Fully Homomorphic Encryption are needed to extend this way of processing things easily. We could practically implement partitioning methods with Fully homomorphic encryption as proposed in the third section. Also, we could implement other parallel programming concepts as described in Table 1. Lastly, we could also implement a whole new algorithm merged with FHE to achieve parallel processing. We could also implement algorithms with FHE with the sole purpose of tackling security issues and not the 'time and memory' drawback.

V. REFERENCES

[1]. G. S. Vennela, N. V. Varun, N. Neelima, L. S. Priya and J. Yeswanth, "Performance Analysis of Cryptographic Algorithms for Cloud Security," 2018 Second International Conference on Inventive Communication and Computational Technologies (ICICCT), Coimbatore, 2018, pp. 273-279, doi: 10.1109/ICICCT.2018.8473148.

[2]. A. Hendre and K. P. Joshi, "A Semantic Approach to Cloud Security and Compliance," 2015 IEEE 8thInternational Conference on Cloud Computing, New York, NY, 2015, pp. 1081-1084, doi:10.1109/CLOUD.2015.157.

[3]. Z. H. Mahmood and M. K. Ibrahem, "New Fully Homomorphic Encryption Scheme Based on Multistage Partial Homomorphic Encryption Applied in Cloud Computing," 2018 1st Annual International 8 Conference on

Information and Sciences (AiCIS), Fallujah, Iraq, 2018, pp. 182-186, doi: 10.1109/AiCIS.2018.00043.

[4]. P. Sha and Z. Zhu, "The modification of RSA algorithm to adapt fully homomorphic encryption algorithm in cloud computing," 2016 4th International Conference on Cloud Computing and Intelligence Systems (CCIS), Beijing, 2016, pp. 388-392, doi: 10.1109/CCIS.2016.7790289.

[5]. R. S. Patil and P. Biradar, "Secure Parallel Processing on Encrypted Cloud Data Using Fully Homomorphic Encryption," 2018 4th International Conference on Applied and Theoretical Computing and Communication Technology (iCATccT), Mangalore, India, 2018, pp. 242-247, doi: 10.1109/iCATccT44854.2018.9001284.

[6]. Chen, GL., Sun, GZ., Zhang, YQ. et al. Study on Parallel Computing. J Comput Sci Technol 21, 665–673 (2006). https://doi.org/10.1007/s11390-006-0665-9.

[7]. Hsu, Ching-Hsien & Salapura, Valentina. (2014). Network and Parallel Computing. International Journal of Parallel Programming. 44. 10.1007/s10766-014-0345-2.

[8]. Navarro, Cristobal & Hitschfeld, Nancy & Mateu, Luis. (2013). A Survey on Parallel Computing and its Applications in Data-Parallel Problems Using GPU Architectures. Communications in Computational Physics. 15. 285-329. 10.4208/cicp.110113.010813a.

[9]. Yang Liu, Wanneng Shu and Chrish Zhang. (2016). A Parallel Task Scheduling Optimization Algorithm Based on Clonal Operator in Green Cloud Computing. Journal of Communications Vol. 11, 2016. ISSN: 2315-4462.

[10]. Abdel-Karim Al Tamimi, “Performance Analysis of Data Encryption Algorithms” https://www.cse.wustl.edu/~jain/cse567-06/ftp/encryption_perf

[11]. D. R. Bharadwaj, A. Bhattacharya and M. Chakkaravarthy, "Cloud Threat Defense – A Threat Protectionand Security Compliance Solution," 2018 IEEE International Conference on Cloud Computing in Emerging Markets (CCEM), Bangalore, India, 2018, pp. 95-99, doi: 10.1109/CCEM.2018.00024.

[12]. V. K. Soman and V. Natarajan, "An enhanced hybrid data security algorithm for cloud," 2017 International Conference on Networks & Advances in Computational Technologies (NetACT), Thiruvanthapuram, 2017, pp. 416-419, doi: 10.1109/NETACT.2017.8076807.

[13]. Nasarul Islam.K.V, Mohamed Riyas.K.V, “Analysis of Various Encryption Algorithms in Cloud Computing,” International Journal of Computer Science and Mobile Computing, Vol.6 Issue.7, July- 2017, pg. 90-97

[14]. Kartit, Zaid & Azougaghe, Ali & Idrissi, H. & El marraki, Mohamed & Mustapha, Hedabou & Belkasmi, Mostafa & Ali, Kartit. (2016). Applying Encryption Algorithm for Data Security in Cloud Storage. 10.1007/978-981-287-990-5_12.

[15]. Alrubaee, Saif. (2019). Security Algorithms in Cloud Computing- Review Paper. 10.13140/RG.2.2.27320.19200.

[16]. R. Nivedhaa and J. J. Justus, "A Secure Erasure Cloud Storage System Using Advanced Encryption Standard Algorithm and Proxy Re-Encryption," 2018 International Conference on Communication and Signal Processing (ICCSP), Chennai, 2018, pp. 0755-0759, doi: 10.1109/ICCSP.2018.8524257.

[17]. K. K. Chennam, L. Muddana and R. K. Aluvalu, "Performance analysis of various encryption algorithms for usage in multistage encryption for securing data in cloud," 2017 2nd IEEE International Conference on Recent Trends in

Electronics, Information & Communication Technology (RTEICT), Bangalore, 2017, pp. 2030-2033, doi: 10.1109/RTEICT.2017.8256955.

[18]. V. S. Mahalle and A. K. Shahade, "Enhancing the data security in Cloud by implementing hybrid (Rsa & Aes) encryption algorithm," 2014 International Conference on Power, Automation and Communication (INPAC), Amravati, 2014, pp. 146-149, doi: 10.1109/INPAC.2014.6981152.

[19]. P. Rewagad and Y. Pawar, "Use of Digital Signature with Diffie Hellman Key Exchange and AES Encryption Algorithm to Enhance Data Security in Cloud Computing," 2013 International Conference on Communication Systems and Network Technologies, Gwalior, 2013, pp. 437-439, doi: 10.1109/CSNT.2013.97.

[20]. X. Song and Y. Wang, "Homomorphic cloud computing scheme based on hybrid homomorphic encryption," 2017 3rd IEEE International Conference on Computer and Communications (ICCC), Chengdu, 2017, pp. 2450-2453, doi: 10.1109/CompComm.2017.8322975.

[21]. M. B. Yassein, S. Aljawarneh, E. Qawasmeh, W. Mardini and Y. Khamayseh, "Comprehensive study of symmetric key and asymmetric key encryption algorithms," 2017 International Conference on Engineering and Technology (ICET), Antalya, 2017, pp. 1-7, doi: 10.1109/ICEngTechnol.2017.8308215.

[22]. P. Varalakshmi and H. Deventhiran, "Integrity checking for cloud environment using encryption algorithm," 2012 International Conference on Recent Trends in Information Technology, Chennai, Tamil Nadu, 2012, pp. 228-232, doi: 10.1109/ICRTIT.2012.6206833.

[23]. A. Azougaghe, Z. Kartit, M. Hedabou, M. Belkasmi and M. El Marraki, "An efficient algorithm for data security in Cloud storage," 2015 15th International Conference on Intelligent Systems Design and Applications (ISDA), Marrakech, 2015, pp. 421-427, doi: 10.1109/ISDA.2015.7489267.

Cite this article as :

Parth Tandel, Abhinav Shubhrant, Mayank Sohani, "Modified Fully Homomorphic Encryption based on Parallel Processing in Cloud Computing", International Journal of Scientific Research in Computer Science, Engineering and Information Technology (IJSRCSEIT), ISSN : 2456-3307, Volume 7, Issue 2, pp.250-262, March-April-2021. Available at
doi : https://doi.org/10.32628/CSEIT217252
Journal URL : https://ijsrcseit.com/CSEIT217252