Parallel Execution of Fully Homomorphic Encryption for Addition of Integers

Sai Sravanthi Lakkimsetty

Problem Report submitted to the

College of Engineering and Mineral Resource

At

West Virginia University

In partial fulfillment of the requirements for the degree of

Master of Science

In

Computer Science

Roy S. Nutter, Jr., Ph.D., Chair

Elaine M. Eschen, Ph.D.

Vinod Kulathumani, Ph.D.

Lane Department of Computer Science and Electrical Engineering

Morgantown, West Virginia

2016

Keywords: Homomorphic Encryption, Parallel, FHE, Encryption, Decryption, Keys, Python, Algorithms

ABSTRACT

Parallel Execution of Fully Homomorphic Encryption
for
Addition of Integers

Sai Sravanthi Lakkimsetty

Fully Homomorphic encryption (FHE) is a means by which cloud computing can be performed on encrypted data, eliminating the data security concerns involved in it. But, FHE has its own set of drawbacks in terms of cost, processing time and memory. Attempts are being made to improve the speed of FHE by different methods. Once such attempt that has been addressed in this report is whether an architecture change might improve the computational speed of FHE. This report presents a method for parallel processing of instructions in FHE using homomorphic addition of integers. It begins with a description of Craig Gentry's partial and fully homomorphic encryption schemes with examples. The later part of the report presents a possible architecture that has been considered for parallel processing of instructions and gives a detailed implementation of parallel homomorphic addition of integers. The report gives some comparisons, possible advantages and limitations of this work.

ACKNOWLEDGEMENTS

This work would not have been possible without the guidance and direction of Dr. Roy Nutter. I would like to extend my sincere gratitude to Dr. Eschen and Dr. Vinod Kulathumani for their time and consideration.

It gives me immense pleasure to thank my family without whose support, this would not be possible. I would like to thank the LCSEE department for their support and opportunity.

Table of Contents

List of Tables

List of Figures

Introduction

1.1 Overview

This report offers how parallel computing resources offered by the cloud might be used to decrease the time taken to compute data that is encrypted using a Fully Homomorphic encryption algorithm. To gain a background, the basic concept of Cloud computing using encryption is explained in detail in this report. In this implementation, the computation that is being performed for encrypted data will be broken into several smaller tasks and will be analyzed for dependencies. These subtasks will be assigned to different computational servers for execution based on their dependency. The possible implementation method is explained in detail with concerned examples in this report.

1.2 Outline of the Problem Report

This report provides an insight into Cloud computing, its services and the threats that are involved. It also gives an overview of how the need for encryption has emerged in Cloud computing and this has evolved over years to use homomorphic encryption for supporting different data computations. The sections following this will talk about some of the basic Homomorphic Algorithms that have emerged in the past and the final section gives a detailed implementation of parallel processing of homomorphic encryption of data for addition of integers.

2 Background

2.1 Cloud Computing

Cloud Computing is an on-demand computing in which organizations can match available computing resources to need. Cloud computing is one means that would reduce the burden of data center maintenance for an organization while still having access to the hardware and software to run business applications. Resources of multiple cloud users may reside on the same

hardware while the user does not have access to the information of others. The cost for this service is usually measured in terms of the real time cloud usage. [1]

2.1.1 Attributes of Cloud Computing [2]

- **Virtualization** – Cloud computing involves utilizing server and storage utilization to allocate and deallocate resources extensively
- **Multitenant Architecture** – All the resources that are allocated to different users are shored among multiple users
- **Network Access –** These resources can be accessed using a web browser on a network device.
- **On demand –** The consumer has the leverage to increase or decrease the services for his organization like storage space and time without any interaction with the service provider.
- **Measured Service –** All the resources that are being used by a consumer are monitored by the provider and are billed accordingly.

2.1.2 Services

A Cloud provider generally offers three levels of services namely infrastructure, platform and software as a service. [3] [4]

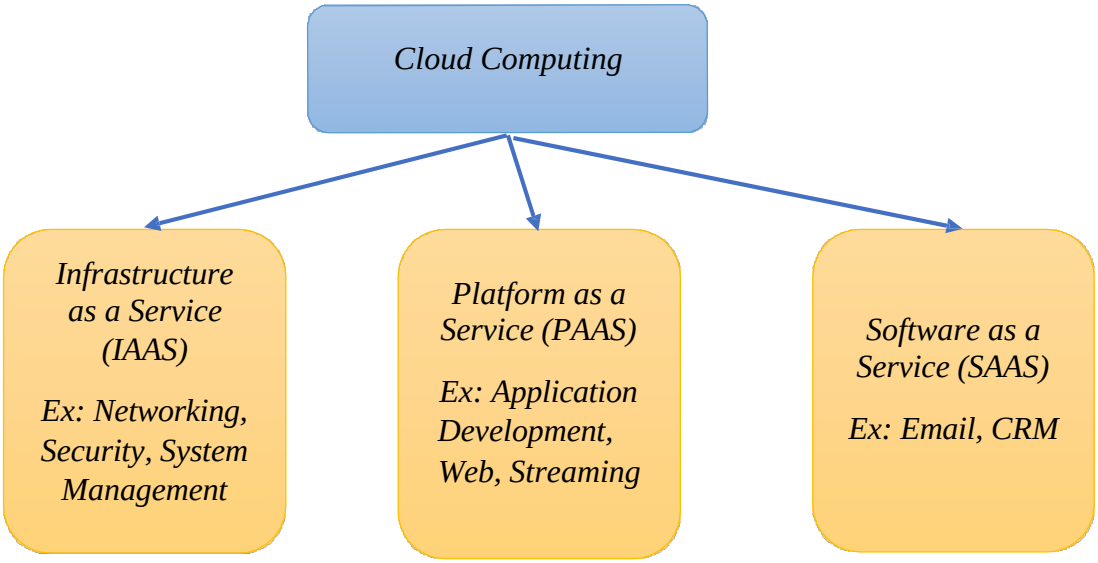

Figure 1: Cloud Computing Services

2.1.2.1 Infrastructure as a Service

The Cloud offers a virtual server for the user to start and configure as their own. The user would have full access and control to this space including the Operating System, Software and data that is used. But the user will not have any control over the physical infrastructure that is used to run this server space that is allocated to them.

Example of an IaaS service is Amazon's EC2. Amazon EC2 from AWS provides a server space virtually to run different configurations as desired. It is billed based on the amount of time the server is used and run.

2.1.2.2 Platform as a Service

This is generally used by developers who wish to develop and maintain their applications online. It basically provides and relies on a wide range of storage options.

One example of PaaS is the google App Engine in which different users are able to simultaneously build different apps written in different programming languages. This is possible with some built in services and APIs that it provides such as NoSQL data stores, memcache, and a user authentication API.

2.1.2.3 Software as a Service

SaaS on the other hand provides the user an access to an application (including data) through a front end portal. Here, the user does not have any control over the infrastructure that goes behind it.

One example of SaaS is Dropbox, in which cloud users can synchronize their files on the cloud, but have no other capabilities. [5]

2.1.3 Data Security

Though many businesses are moving towards cloud utilization in the recent times, many are not. The main reason behind this is data security. Sensitive data such as financial information and health records that fall into wrong hands may prove to be fatal to a company. So, data

security helps to enhance the user's trust in Cloud computing environment. Cloud providers do offer a level of security in terms of Confidentiality, Integrity and Availability. [6]

2.1.3.1 Confidentiality

Confidentiality is mainly about keeping the data private. Authentication and access control strategies are used to ensure data confidentiality. Only authorized personnel should be given access to data. One way by which this may be ensured is through encryption of data. If the data is passed through an encrypting function along with the secret key, then the encrypted data can be safely stored and transmitted anywhere. Only a person who has the decryption key will be able to obtain the original data by decrypting it.

2.1.3.2 Integrity

Integrity is the degree of confidence one has in the cloud provider. It ensures that the data is modified only by authorized people and includes protecting data from unauthorized deletion, modification and fabrication. There are several tools to check the integrity of data. One way is to *hash* the message that is sent using a cyclic redundancy check. This can be cross checked with the hash of the message received to know whether the data has been modified by another person without authorization. There are several other techniques to ensure data integrity on the cloud like digital signatures etc.

2.1.3.3 Availability

Availability ensures that the data is available to concerned people at the appropriate time. When incidents like fire and network failures occur, availability ensures that the data is still recoverable and available to the concerned people. For example, Denial of Service attacks are possible in the cloud during which the cloud users' data availability is affected.

2.1.4 Cryptography in Cloud

As discussed in previous sections, Cryptography is a convenient solution for confidentially and integrity of data. When the cloud provider ensures that the data is encrypted during storage and transmission, then even if an unauthorized bad user obtains access to this

data, all the bad user will see is meaningless gibberish. Without the encryption keys, modification of the data is not possible.

This gives an additional layer of security in the cloud. But, this process involves encrypting data before sending it to the cloud provider. The drawback of this involves performing calculations on the data. To perform any computations on this encrypted data, the data has to be decrypted first to work on it. Especially, when this process involves a third party, the possibility of keeping the data safe again becomes obscure. So, a mechanism that is used to perform computations on the encrypted data without decrypting them is needed. This technique is called Homomorphic Encryption.

2.2 Cryptography

Cryptography is a method in which plain text is converted into a form such that only authorized parties can read it. The converted form of the message is called a cipher text. Any form of cryptosystem involves a key that is used to convert the plain text to cypher text (encryption process) and vice versa (decryption process).

2.2.1 Overview of Cryptography

Any cryptosystem basically involves 5 tuples E, D, M, K and C

- E → Encryption Function
- D → Decryption Function
- M → Plain text message
- C → Cipher text for the message
- K → All keys involved

When an encryption function is applied on the plain text, M using an encryption key, k_e , then a cipher text C is obtained.

$$E (k_e, M) \rightarrow C$$

When a decryption function is applied on the cipher text, C using a decryption key, k_d, then the original plain text, M is obtained.

$$D\ (k_{d,}\ C) \rightarrow M$$

A diagrammatic representation of the above is shown in Figures 2 and 3.

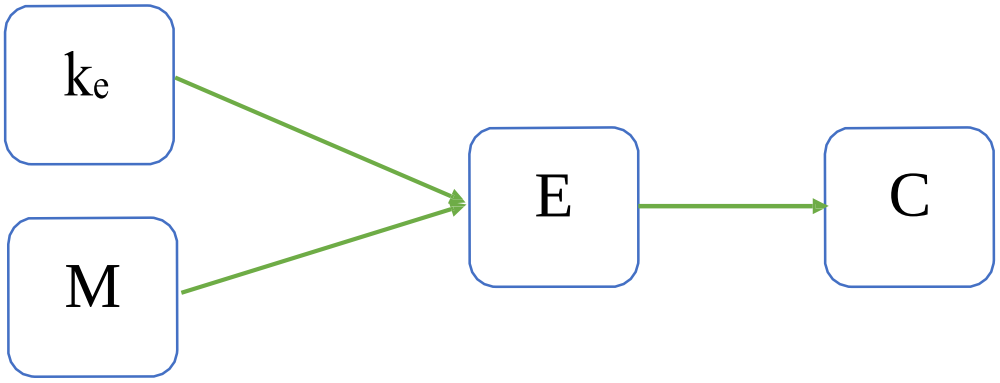

Figure 2: Encryption Process

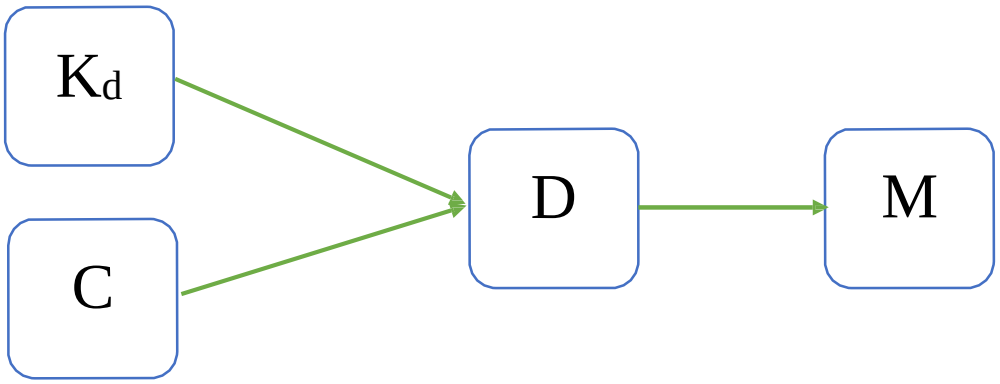

Figure 3: Decryption Process

2.2.1.1 Types of Cryptography

The encryption and decryption keys may be same or different. Two different cryptography styles have been defined based on the keys used. [7]

- **Symmetric Cryptography**

  It is also called as Private Key Encryption in which both the encryption and decryption keys (k_e and k_d) are same. The sender is responsible for sending the same key used for encryption to the receiver. The receiver then uses this key to decrypt the message and recover the plain text. The main challenge of this type of cryptography is to securely transmit the key to the recipient without allowing it to fall into the wrong hands.

  Some of the widely used Private Key encryption systems used today are DES (Data Encryption Standard), AES (Advanced Encryption Standard) etc.

- **Asymmetric Cryptography**

  This is also called Public Key Encryption. The encryption and decryption keys are different but related for public key encryption. This relationship allows one of the keys to be kept secret and the other key can be made public. It is not important on which key is used first. But both the keys are a must for the communication to happen. The public key is published to everyone to use and encrypt messages. However, only the receiving user would have access to the decryption key/secret key that enables the messages to be read again. This ensures a secure communication between two parties.

  Some of the Public Key Encryption algorithms used today are RSA (Rivest, Shamir and Adleman), ElGamal etc.

2.2.1.2 Significance of Key Length

The size of the key controls the level of complexity of an encrypted data. The larger the key, the harder it is to crack the encrypted data. With computers coming up with brute force attacks to check every possible combination of key, it has become easier to find the key used for encryption. Hence, larger keys are usually preferred. With the addition of each bit to the key length, number of possible key values almost double making it an even more difficult job for the person trying to find the key. [7]

2.2.2 Need for Privacy Homomorphism [8]

Privacy Homomorphism is one form of encryption that allows computations to be performed on cipher text thus generating an encrypted result, which when decrypted matches the result of operations performed on plain text.

There is a need to chain multiple services together in businesses for processing various information. Sometimes data processing might be done by a third party. For this, the data must be decrypted and exposed to the untrusted third party in order to process it. Then data is re-encrypted and stored in the system. During the time in which the data remains with the third party, it stays exposed to several threats and vulnerabilities. So, sensitive data is residing in the hands of a company that offers computing as a service.

Privacy Homomorphism was first put forth by Rivest, Adleman and Dertouros in which they proposed a homomorphic function that enables a user to: store the data in the encrypted form; send the encrypted data to the third party which would evaluate this encrypted data; and send the results back to the user for decryption. Several initial cryptosystems that have been proposed supported only partial computations. First Fully Homomorphic Encryption scheme was proposed by Gentry in 2009.

2.2.3 Comparison of different Cryptosystems

2.2.3.1 RSA Algorithm

RSA is a public key cryptosystem in which the algorithm was published by Rivest, Shamir and Adelman. It uses two different keys for encryption and decryption. This algorithm is popular for its security that comes from its difficulty of factoring large prime integers used for key generation. The algorithms for Key generation, Encryption and Decryption are shown below. [7]

Algorithm 1: KeyGen Algorithm

```
Data: None
Result: Encryption Key: (e,n) and decryption key: (d,n)
begin
     Compute (p,q) ∈ P be two prime integers chosen randomly;
```

```
     Compute n= p*q such that n is an integer
     Compute ϕ(n) = (p-1)(q-1), ϕ(n)∈ Z ;
     Randomly select d ∈ Z which is relatively prime with ϕ(n);
     Find e ∈ Z such that e*d = 1 (mod (p-1)(q-1));
End
```

Table 1: KeyGen for RSA Algorithm

Algorithm 2: Encrypt Algorithm

```
Data: Encryption Key: (e,n) and plain text message M ∈ [0, n-1]
Result: Cipher text C ∈ [0, n-1]
begin
     Compute C = Mᵉ (mod n);
End
```

Table 2: RSA Encrypt Algorithm

Algorithm 3: Decrypt Algorithm

```
Data: Decryption Key: (d,n) and Cipher text C ∈ [0, n-1]
Result: Plain text message M ∈ [0, n-1]
begin
     Compute M = Cᵈ (mod n);
End
```

Table 3: RSA Decrypt Algorithm

2.2.3.1.1 RSA Homomorphism

RSA is multiplicatively homomorphic, but not additively homomorphic. That is, the product of two cipher texts when decrypted is equal to the product of the two plain texts. [9]

$$Enc(X * Y) = Enc(X) * Enc(Y)$$

Example 1: RSA Homomorphism

```
Let M₀ = 2 and M₁ = 3

Assuming p = 3 and q = 11
From the algorithms above,

n = 3 * 11 = 33, ϕ (n) = (2) * (10) = 20;
d is a number that is relatively prime with ϕ (n) and so let d
= 7;
```

Hence, $e * d = 1 \pmod{\phi(n)} \Rightarrow e * 7 = 1 \pmod{20} \Rightarrow e \sim 3$.

Calculating the cipher texts C_0, C_1 respectively:
$C_0 = M_0^e \pmod{n} = 23 \pmod{33} = 8$
$C_1 = M_1^e \pmod{n} = 33 \pmod{33} = 27$
Taking the product of the two cipher texts,
$C_0 * C_1 = 8 * 27 = 216$

Decrypting this product $\Rightarrow C_d \pmod{n} = 2167 \pmod{33} = 6$ which is equal to the product of the plain texts i.e., $M_0 * M_1 = 2 * 3 = 6$.

Table 4: RSA Homomorphism Example

2.2.3.2 Paillier Algorithm

This is another asymmetric cryptosystem proposed by Paillier in 1999 which has additive properties rather than multiplicative. It also involves creating an encryption and decryption key. The Algorithms for KeyGen, Encryption and Decryption are shown below [10]:

Algorithm 4: KeyGen Algorithm

Data: None
Result: Encryption Key: (n,g) and decryption key: (p,q,g)
begin
    Randomly select $(p,q) \in P$, a set of prime numbers such that $GCD(pq,(p-1)(q-1))=1$;
    Compute $n = p*q$ where $n \in Z$;
    Compute $\lambda = LCM(p-1, q-1)$ where $\lambda \in Z$;
    Compute continue = false;
    while continue = false do
        Compute continue = true;
        Randomly select g from Z_{n2}^* where g is an integer;
        If g does not have an order in Z_{n2} which is a multiple of n **then**
            Compute continue = false;
        end
        If $(g^{\lambda n} - 1)/n \pmod{n^2}$ is not in the invertible elements of Z_n **then**
            Compute continue = false;
        end
    end

end

Note: Z_n is the set of integers modulo an integer n and Z_{n2}^* is the multiplicative group of Z_n i.e., the set of congruence class of integers modulo n that are relatively prime to n

$Z_{n2}^* = \{x \in Z_n\}$ such that GCD(x,n) = 1

Table 5:KeyGen for Paillier Algorithm

Algorithm 5: Encrypt Algorithm

Data: Encryption Key: (n,g) and plain text message $M \in Z_n$
Result: Cipher text $C \in Z_{n2}$
begin
Choose a random non zero integer $r \in Z_{n2}^*$;
Compute $C = g^M.r^n (\text{mod } n^2)$;
end

Table 6: Paillier Encrypt Algorithm

Algorithm 6: Decrypt Algorithm

Data: Decryption Key: (p,q,g) and Cipher text $C \in Z_{n2}$
Result: Plain text message $M \in Z_n$
begin
Choose $k = (g^{\lambda n} - 1)/n \ (\text{mod } n^2)$;
Compute $\mu = k^{-1} (\text{mod } n)$;
Compute $M = ((c^{\lambda n} - 1)/n \ (\text{mod } n^2)).\mu (\text{mod } n)$;
end

Table 7: Paillier Decrypt Algorithm

2.2.3.2.1 Paillier Homomorphism

As mentioned above, this is additively homomorphic i.e. the computing the product of their cipher texts and decrypting it would give the summation of their plain texts. [10]

$$\text{Enc}(X + Y) = \text{Enc}(X) * \text{Enc}(Y)$$

Example 2: Paillier Homomorphism

Let $M_0 = 2$ and $M_1 = 3$

Assuming p = 3 and q = 5

From the algorithms above,

$n = 3 * 5 = 15$, $\lambda = LCM(2,4) = 4$
Compute $g = 29$ where $g \in Z_{225}^*$
Verify that g is invertible in Z_{225}^*: $29*194 \pmod{225} = 1$
Verify that the order of g in Z_{225}^* is a multiple of n: $29^{135} \pmod{225} = 1$ and 135 is a multiple of 15.
Verify that $(g^{\lambda n} -1)/n \pmod{n^2}$ is in the invertible elements of Z_{15}: $(29^4-1)/n \pmod{225} = 7$ which is invertible in Z_n because $7 * 193 \pmod{15} = 1$.

Calculating the cipher texts C_0 , C_1 respectively by choosing a random r = 11:
$C_0 = g^M.r^n \pmod{n^2} = 29^{2\cdot} 11^{15} \pmod{225} = 41$
$C_1 = g^M.r^n \pmod{n^2} = 29^{3\cdot} 11^{15} \pmod{225} = 64$
Taking the product of the two cipher texts ,
$C_0 * C_1 = 41 * 64 = 2624$

Decrypting this product =>
Compute $k = (g^{\lambda n} -1)/n \pmod{n^2} = (29^4 -1)/n \pmod{225} = 7$.
Compute $\mu = k^{-1} \pmod{n} = 7^{-1} \pmod{15} = 13$
$M = ((c^{\lambda n}-1)/n \pmod{n^2}).\mu \pmod{n} = ((2624^4-1)/15 \pmod{225}).13 \pmod{15} = 5$ which is same as the sum of the plain text messages i.e., $M_0 + M_1 = 2 + 3 = 5$.

Table 8: Paillier Homomorphism Example

2.2.3.3 Comparison [9]

Property	RSA	Paillier
Platform	Cloud Computing	Cloud Computing
Homomorphism	Multiplicative	Additive
Keys used by	The client (Different keys are used for encryption and decryption)	The client (Different keys are used for encryption and decryption)
Encryption Complexity	$O(n^2)$	$O(\|n\|^{2\cdot} \|\alpha\|)$
Decryption Complexity	$O(n^3)$	$O(\|n\|^{2+e})$
Privacy of Data	Is ensured in communication and storage processes	Is ensured in communication and storage processes
Security Applied to	Cloud Provider Server	Cloud Provider Server

Table 9: Comparison of Different Homomorphic Encryption Schemes

3 Literature Survey

3.1 Craig Gentry's Algorithm

Craig Gentry built the first fully homomorphic encryption scheme in 2009. A fully homomorphic encryption scheme is one that supports arbitrary computations on cipher texts unlike partial homomorphic schemes that support either addition or only multiplication. Since a program with this scheme does not need one to decrypt its inputs at any point, this proves to be a major breakthrough in cloud computing especially when outsourcing to third party companies.

Craig Gentry also started his algorithm from a somewhat homomorphic scheme that supported only addition and multiplication. He then later modified it by introducing the concept of *bootstrapping* that converted the partial homomorphic scheme to a fully homomorphic encryption scheme. [11] [12]

3.1.1 Partial Homomorphic Scheme

This scheme presented by Gentry and Halevi supports addition and multiplication of cipher texts. The homomorphic addition or multiplication is simply the addition or multiplication of two cipher texts and then taking the result modulo d. The decryption of the result is the binary addition or multiplication of the plain texts without the carry.

For this, Gentry used the concept of ideal lattices. Ideal lattices are special lattices that are used in cryptography to reduce the number of parameters required to describe a lattice. This scheme

uses an ideal lattice, J to encrypt a bit b. The value of the bit is encoded to a vector $\vec{c}$ and its integer distance from a J-point. If the distance is even, then the bit is 0, otherwise its 1.So, the encryption takes that from $\vec{c} \leftarrow 2\vec{r} +$ [illegible] (mod B_{pk}) where $\vec{r}$ is a random noise vector < u_0, u_1, u_2….. u_{n-1} > with each entry chosen as 0 with some probability q and ±1 with probability (1-q)/2 each.

The secret key is denoted using another vector $\vec{w}$. The original plain text message can be obtained by multiplying $\vec{w}^{-1}$ (mod 2). [12] [13]

Algorithm 7: KeyGen Algorithm

Data: t, the bit-size of integers to use; n, the dimension of lattice to use which is a power of 2;

Result: A single odd coefficient of $\vec{r}$ and a public key pk consisting of integers d and r;

begin

Compute $f_x(x) \leftarrow x^n + 1$; // monic and irreducible polynomial in x and n

Randomly create an n-dimensional vector $\vec{w}$ where each element $w_i \leftarrow \sum_{i=} a \cdot r_i$ random t bit signed integer;

Compute $w(x) \leftarrow \sum_{i=} a \cdot r_i$;

Compute a matrix V which is a rotation of $\vec{w}$

Find w(x) such that w(x) X v(x) = resultant(v(x),$f_n(x)$)(mode $f_n(x)$);

// where v(x) is a polynomial on matrix V and the resultant of v(x) and $f_n(x)$ would give a variable d such that d $\in \mathbb{Z}$

Compute w to be an odd coefficient of w(x);

If *there does not exist a vector* $\vec{r}$ *and integer r such that* $\vec{r}$ *X V = <-r, 1, 0, ……, 0>* **then**

Go to Generation of n dimensional vector at line 2

end

end

Table 10: Keygen Algorithm for Gentry's Partial Homomorphic Scheme

Algorithm 8: Encrypt Algorithm

Data: A bit b $\in$ {0,1}

Result: Cipher text C, an integer

begin

Randomly generate an n-dimensional vector $\vec{r}$ where each element r_i is selected from {0, -1, 1} with probabilities {0.5, 0.25, 0.25}

Compute $C \leftarrow m + 2\sum_{i=0}^{n-1} r_i w_i \pmod{x_0}$;

end

Table 11: Gentry Partial Encrypt Algorithm

Algorithm 9: Decrypt Algorithm

Data: Cipher text C, an integer and a single off coefficient $\vec{w}$ denoted as w_i;

Result: A bit b $\in$ {0,1}

```
    Compute b ← [𝑐]𝑠(𝑥);
end
```

Table 12: Gentry Partial Decrypt Algorithm

Example 3: Gentry Partial Homomorphism

Let bit b = 1 to be encrypted and noise vector u = <u_0,u_1> = <-1, -1>;

Assuming n = 2 and t = 4
From the algorithms above,
Compute $f_n(x) = x^2+1$;
Compute $\vec{v}$ = <3,-8>, v(x) = <3-8x>,
W = [$\vec{v}$ $-\vec{v}$];
d = resultant(v(x),f$_n$(x)) = resultant(3-8x,x^2+1) = 73;

PolynomialGCD(-8x+3,x^2+1,Modulus -> 73) = x+27;
Computing r = 27;

Computing W = [ ];

So, w(x) = 3x-8 and w=odd coefficient in w(x) = 3.

Calculating the cipher text C by choosing a noise vector u:
u= <u_0,u_1>=<-1,-1>;
C = m + $\sum_{i=}^{}$ r (x) = [1+2(-1*27)]$_{73}$ = 20;

Taking the sum of the cipher text,
C′= [C+C]$_d$ = [20+20]$_{73}$ = -33
Decrypting this Sum =>
Compute b = [𝑐]𝑠(𝑥) = [-33*3]$_{73}$(mod 2) = -26(mod 2) = 0, which is same as the sum of the binary addition of the bits
without the carry.

But there is a reason why this algorithm is partial or somewhat homomorphic. This algorithm supports only a limited number of additions and multiplications. When two cipher texts are added, the amount of error associated with each of them also gets added. Similarly, when two cipher texts are multiplied, the amount of error also increases dramatically.

So, as the number of times that an operation is performed is increased, the scope of error or noise associated with the cipher text will become too large for the scheme to correct and handle. The error comes from the distance of the cipher text C to its nearest lattice vector. This distance should not be too large nor short as this would make it possible for the cipher text to be nearer to another lattice vector resulting in a different message after decryption.

3.1.2 Fully Homomorphic Encryption

Fully Homomorphic encryption allows unlimited number of operations to be performed on encrypted data. To convert the Somewhat homomorphic encryption to a Fully Homomorphic encryption and to get a less noisy encrypted value, Gentry used a concept called Bootstrapping. [14]

3.1.2.1 Bootstrapping

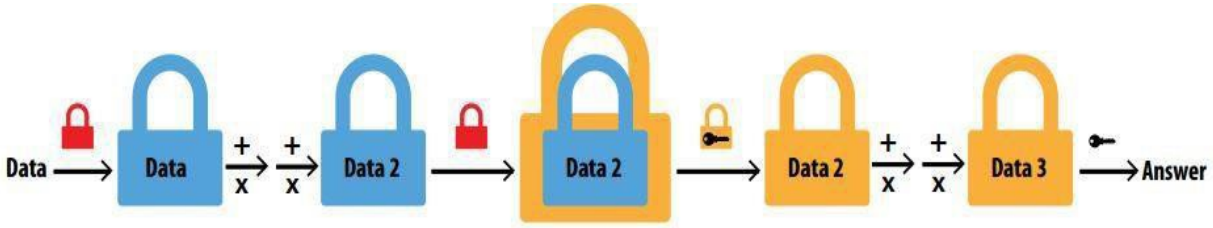

Figure 4: Bootstrapping Procedure

Bootstrapping is a process that helps to reduce the amount of noise involved in homomorphic encryption thereby facilitating the performance of some more operations before the noise becomes high again. This process involves refreshing the cipher text C obtained after homomorphic encryption to produce C' using an encrypted version of the secret key. Each bit in the secret key in the decryption circuit is replaced by a large cipher text that encrypts that bit. So, the public key also contains an encrypted version of the secret key that is used for decryption circuit. This process is all about evaluation of the decryption circuit homomorphically. If an Encryption scheme is smart enough to evaluate its own decryption circuit, it is said to be bootstrappable. This would make the scheme totally self sustainable.

From the previous section, the homomorphic evaluation is performed on the cipher text and would result in one such cipher text. When this cipher text is decrypted, it gives the same result

that is obtained when the same operation is performed on the plain text. This implies that a homomorphic evaluation of the decryption circuit results in the encrypted version of the decryption i.e. representing the same value.

In general, when freshly obtained cipher texts x1,x2,….xn (shown in figure 5) are given as inputs to evaluate a polynomial or a circuit, f, the result is another cipher text, y(shown as output in figure 5) that has some noise.

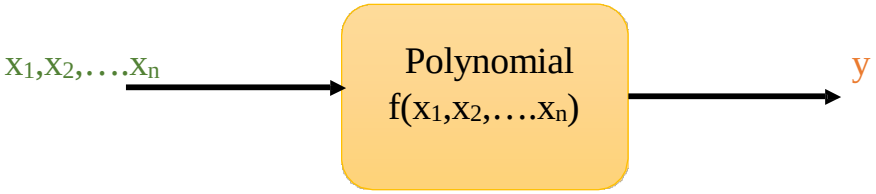

Figure 5: Possible Homomorphic Encryption

This cipher text can still be decrypted as the noise did not exceed the threshold. But, no more operations can be performed on this cipher text y as that would result in the noise exceeding the threshold and making the cipher text, y unfit for decryption.

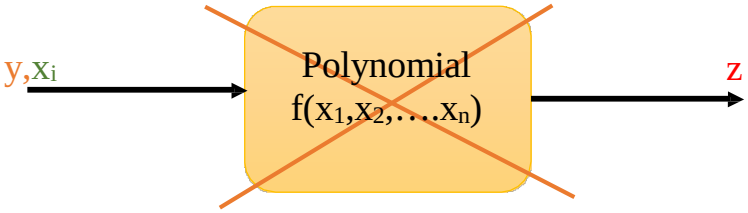

Figure 6: Homomorphic Encryption - Not Possible

In Bootstrapping, the homomorphic encryption scheme performs some additional functions. It refreshes the cipher text obtained after homomorphic encryption in the encryption scheme to reduce its noise. This is done by decrypting the cipher text using the secret key that is sent to the scheme in encrypted form. Then, homomorphic evaluation is performed on the decrypted values of those cipher texts and a fresh cipher text C' is generated. This process is referred as recrypt or homomorphic decryption.

$$M = \text{decrypt}(C) = \text{decrypt}(C')$$

.

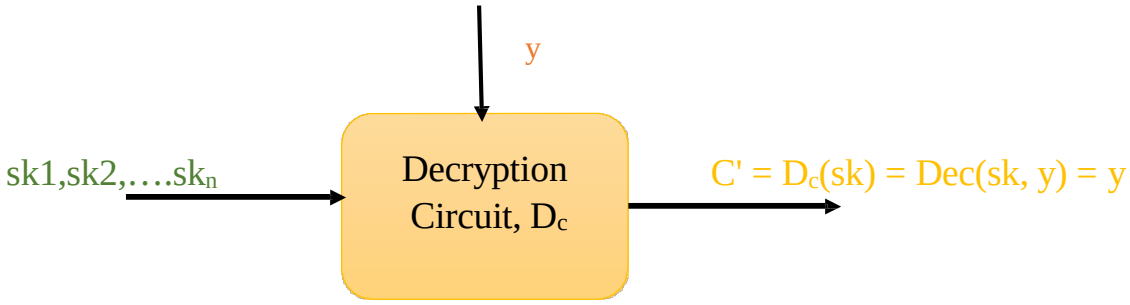

Figure 7: Bootstrapping: Homomorphic Evaluation of decryption circuit

3.1.2.2 Squashing the decryption function

The decryption function that is being used in Gentry's partial homomorphic encryption has a higher order polynomial and higher order polynomials are not suitable for homomorphic evaluation. Only a polynomial of small degree can be evaluated homomorphically. Otherwise, the noise increases at a faster rate in which the decryption results in an incorrect value. To do this, Gentry used a *squash* technique in which the polynomial is expressed as a low degree polynomial and is used to decrypt the cipher text with the new secret key.

4 Statement of Problem

Several encryption schemes exist that perform mathematical operations on encrypted data. One of them is Craig Gentry's homomorphic encryption scheme. Though Gentry has transformed his "somewhat homomorphic encryption" scheme to a "fully homomorphic encryption [FHE]" scheme using refreshed cipher texts, this concept involves adding one additional layer of encryption to increase the number of sequential computations possible. This does increase the overall computational effort and time required for the same set of operations. Gentry's FHE scheme takes longer to perform operations especially multiplication and division. It is thus impractical for many applications. FHE would need to become many times faster, in order to achieve generic practicality. Though this concept has proven to be a game changer in cloud computing, this lack of computational efficiency has led to it not being well received in practical applications.

Considering the demand for encrypted computation in cloud computing, attempts are being made to make this algorithm more efficient in terms of robustness, efficiency and data processing time.

This report provides an alternative solution that could improve the computational speed of homomorphic encryption. The problem addressed herein is whether an architecture can be designed that might improve the speed of computation of FHE.

5 Environmental Setup

To increase the robustness of homomorphic encryption systems, the size of the keys used must be increased. This in turn would impact the processing time and speed of the system making it impractical. One way of reducing the computational time for processing multiple operations is to distribute the number of computational requests to multiple nodes on the cloud. This calls for parallel processing of homomorphic encryption. In order to evaluate this possibility, the following environment was considered.

A client server architecture has been proposed to evaluate Gentry's scheme in which computational requests are sent to multiple servers. They are processed and the responses from these servers are sent back to the client. The client accumulates the responses and gives the final result to the user. Each request that is sent to the server must hold the instruction to be performed, calculations to be performed and the public key for the data that allows bootstrapping. Splitting the work to be done by a single server to multiple servers results in parallel execution of the same set of instructions and also reduces the computational time.

5.1 Architecture of the Cloud Platform

The architecture mainly consists of three important components: Client, Computational Dispatcher and the Server(s). The total number of operations that need to be performed for the evaluation of data is broken down to subtasks and are analyzed for dependencies between them. Based on these dependencies, the tasks are queued and sent to different servers. Once the responses are obtained from these servers, the dependencies are evaluated again based on the responses and the process is repeated by sending the subsequent set of tasks to the servers. The tasks that are finished are marked complete.

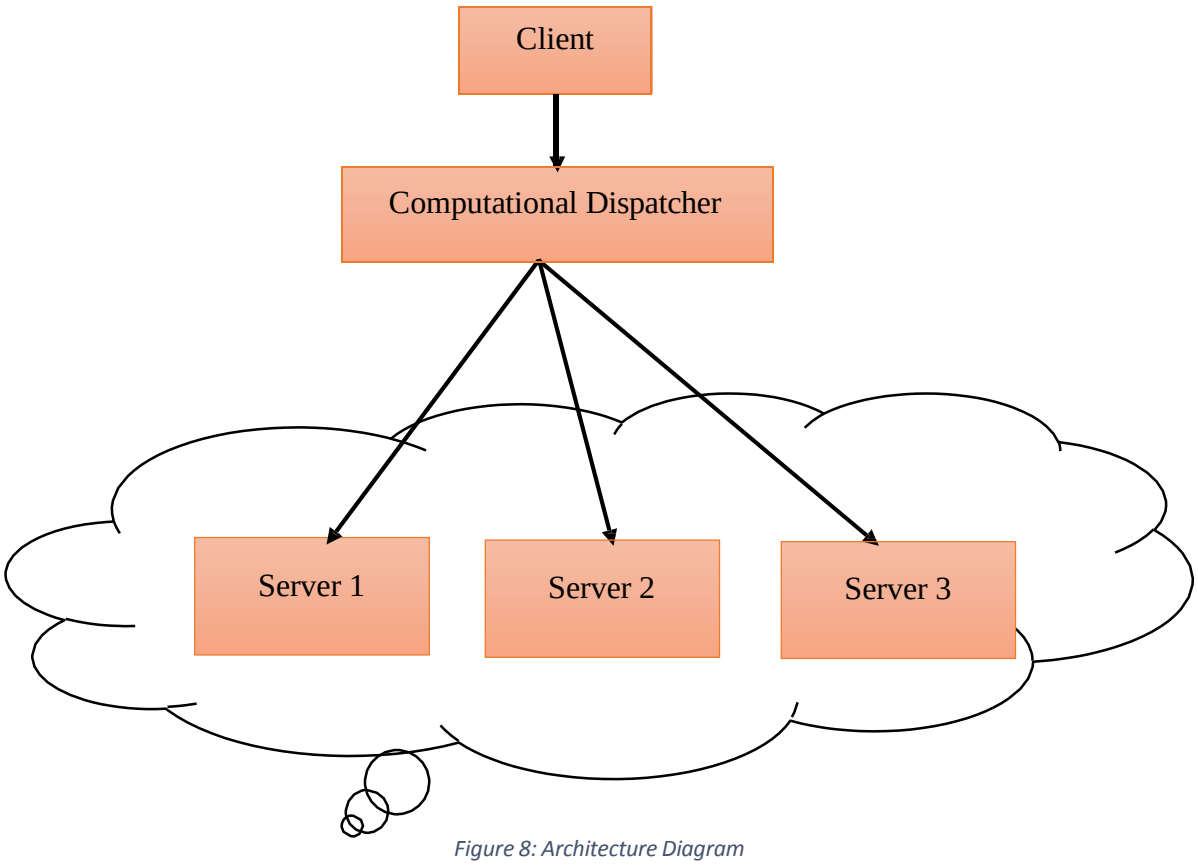

Figure 8: Architecture Diagram

5.1.1 Client

The Client is mainly a user interface by means of which the input is given to the system. The input that is passed includes a set of integers on which the computation is to be performed, the set of computations, and the list of servers. The client is responsible for transferring this information securely to the Computational Dispatcher in a suitable format. For security reasons, it is assumed that both Client and Dispatcher lie in the same system to avoid network security issues.

For convenience, a common format has been considered for giving the input that would make processing a little easier. Input integers are labeled as i0, i1, i2….., iN. The output variables are given as o1, o2,….., oN. These output variables are returned to the client. The instruction must include a valid operation along with one of the input or output arrays.

5.1.2 Computational Dispatcher

The Computational dispatcher is responsible for taking the input i.e. the data, list of computations and the list of servers from the client. The computational Dispatcher is responsible for three major functions:

- Encryption of input data into an array of bits
- Dividing the input into subtasks and finding dependencies among them i.e. constructing a dependency graph.
- Decryption of the returned output into plain text/integers

All the tasks/computations that are independent of outputs of other tasks are sent first to the servers for processing. Once they are sent, they are marked as sent to track them. Once the responses are obtained, these task are marked complete and all the dependencies on these tasks are checked again. The entire dependency graph is updated based on the outputs obtained. Once updated, the next set of tasks are sent for computation. This process repeats until no further tasks are left. Dispatcher is a key component to manage the entire computation process. Once there are no tasks left, then the dispatcher converts the encrypted bits back to their respective integers and returns the array of integers to the client as output.

5.1.2.1 Creating Data Dependency Graph

Once a request is sent to the dispatcher from the user interface, it creates a data dependency graph for the set of instructions it has received. For a circuit that basically involves multiple calculations, two arrays are used to store the Parent and Child nodes namely P and C respectively. P is an array of all the parent pointers and C is an array that holds all the child pointers in the graph. D is an associative array that holds the data owners. To begin with, D is empty indicating the root node.

Once initialization is done, all the instructions are iterated to calculate the dependencies between different instructions using the algorithm below. Each node indicates a calculation that is to be performed and each edge between two nodes depicts the dependency between nodes.

Algorithm 10: Dependency Graph

```
Data: E[] where each E has instruction, inputs and outputs
Result: A directed graph that shows data dependency between
nodes (a,b)∈ E;
begin
          Let P and C be arrays;
          Let P[index]=C[index] for each index(offset 1) of E;
          Let D be an associative arrays
          foreach e, index in E do
             foreach input in e.inputs do
               if D[input] is null then
                    D[input]=0;
               end
               Let d=D[input];
               Add d to P[index]unless already there;
               Add index to C[d] unless already there;
            end

            foreach output in e.outputs do
               if D[output] is null then
                    D[output]=0;
               end
               Let owner=D[output];
               if owner is not the root node then
                    foreach dependent in C[owner] do
                         Add dependent to P[index] unless
already there;
                         Add index to C[dependent] unless
already there;

                    end
               end

               let D[output]=index;
            end
         end
end
```

Table 13: Dependency Graph Generation

The code that has been used to generate the dependency graph has been executed in Python [15] and is added at the end of this document in Appendix A.

An example for constructing a dependency graph is shown below:

Number	Instruction
1	a=b+e
2	c=e+f
3	d=a+c

Table 14: Sample Instructions

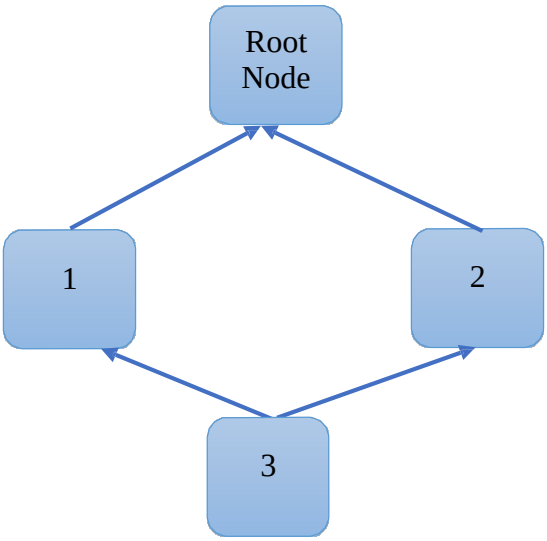

Figure 9: Example of Dependency Graph

The dependency graph for the sample set of instructions in the table above is shown. In this graph, the three nodes 1, 2 and 3 correspond to the three different calculations that are being performed as per instructions 1, 2 and 3 respectively. Node 1 and node 2 are not dependent on any other nodes and hence are connected only to the root. However, node 3 corresponding to the instruction, d=a+c depends on the values of a and c which are being computed as per instructions 1 and 2. Hence, node 3 is dependent on these two nodes.

5.1.2.2 Finding Subcircuits

Once the dependency graph is constructed, the dispatcher finds the order in which the tasks are sent to the servers to be executed. Each computation that fulfils the final processing of the data is called a subcircuit. The dispatcher gives priority to any child node that is only dependent on the root node and adds them first to the Queue. This queue is responsible for storing the list of subcircuits to be evaluated in the priority order in which they are stored. Then

the nodes which are dependent on these child nodes are taken and marked for execution in the same order in the queue.

This process for finding and making an effective PendingQueue is shown with the algorithm below:

Algorithm 11: Finding SubCircuits

```
Data: P and C associative arrays that show the parent and Child
nodes
Result:E,a subcircuit;
Begin
          Let c be any element of C[0] not pending execution
and with only one parent;
          Mark c as pending execution
          Let E=[c];
          Let candidates = C[c];
     while length of candidates > 0 do
          Let c=Pop(candidates);
          Let add=true;
          foreach c_parent in P[c] do
               if c_parent not in E then
                    Let add=false;
               end
          end

          if add is true AND c is not marked as pending
execution then
               Add c to E;
               Mark c as pending execution;
               Add to candidates all members of C[c] unless
already there;
          end

     end
end
```

Table 15: Finding Subcircuits

This algorithm has been implemented using a Python Code that is added at the end of this document in Appendix A.

5.1.3 Server

The server takes its input i.e. public key, an array of encrypted bits and a subcircuit to evaluate from the dispatcher. It then performs the necessary instructions or calculations using

FHE algorithms and stores the result in an output array. This output array is then sent back to the dispatcher.

The subcircuit that are being sent to the server must be in the form of a delimited list i.e. the first item in the list must be the instruction and every other element on the list should be either an input or the output.

5.1.4 HElib for Fully Homomorphic Encryption

For performing Homomorphic encryption and decryption during this implementation, HElib has been used. It is a software library that implements FHE focusing mostly on Gentry-Halevi-Smart optimizations. It provides all the low level operations such as addition, multiplication, shift etc. This library is written in C++ and uses the NTL mathematical library. It supports multi-threading which is used for simultaneous execution of different processes. [16]

5.1.4.1 NTL Library

NTL is high performance, C++, mathematical and portable library providing data structures and algorithms that are used for manipulating integers, vectors, matrices etc. It is an open source software developed by Victor Shoup. [17]

5.1.5 Overall Process Flow

The overall process flow between different components for parallel evaluation of homomorphic addition is shown below:

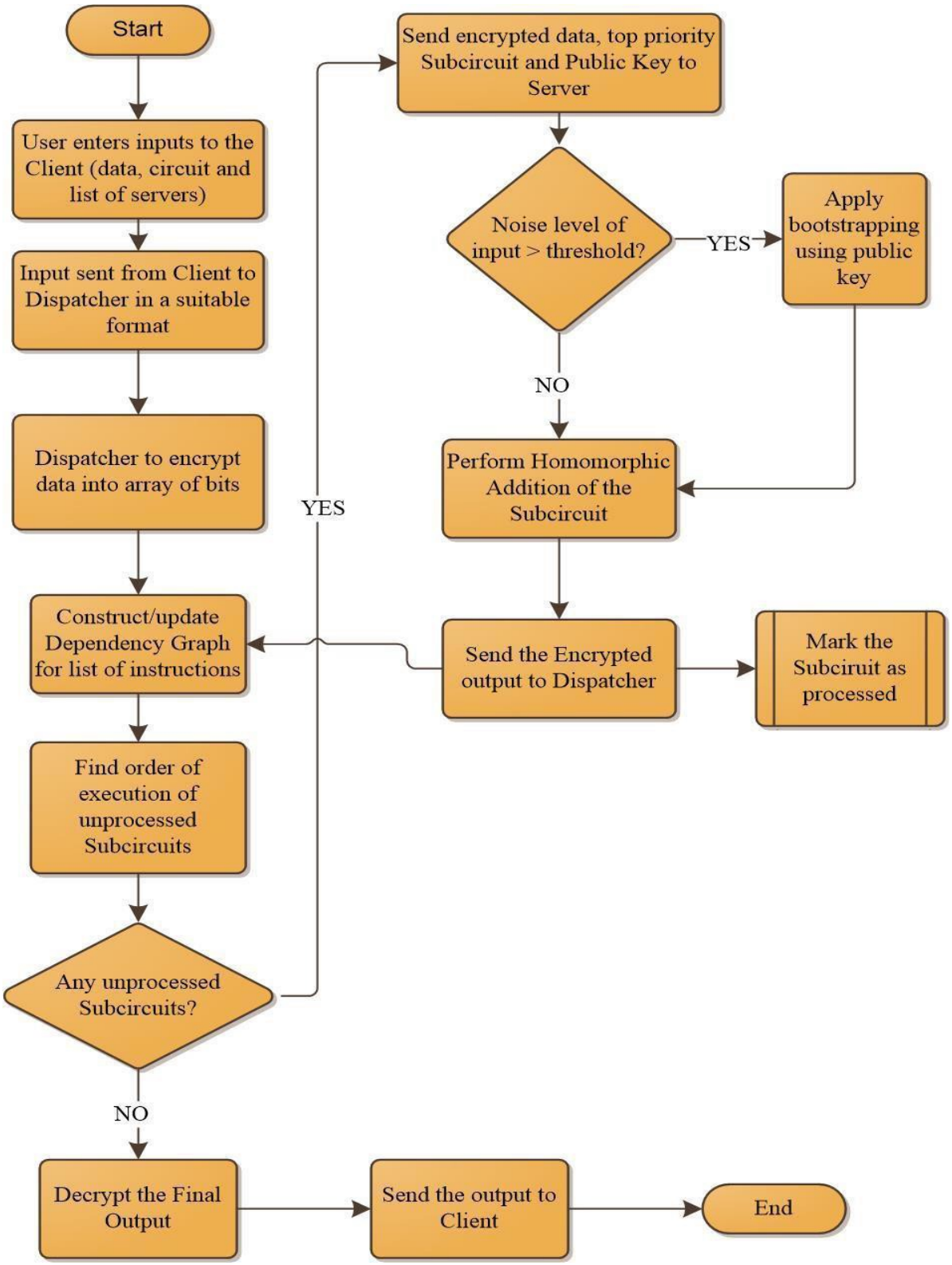

Figure 10: Process Flow Diagram

5.2 Implementation for Addition of Integers

In order to test the parallel processing for homomorphic encryption, evaluation of addition of integers has been chosen as the scenario.

For this experiment, integers were chosen at random. Unlike the usual sum where all the numbers are encrypted and added serially using homomorphic encryption, the sum of these integers was decided to be performed in parallel to process it efficiently. The algorithms mentioned were implemented using python code. HElib encryption scheme in C++ is used, based on Craig Gentry Homomorphic Encryption.

To process these integers in parallel, they are split into adjacent pairs and the sum of each pair is found out. The resulting integers from the previous step were then again split into pairs and summed. This process is continued until there is a single output. This process was chosen to increase the number of parallel processings involved in this computation.

Data Chosen: Random Integers generated by the system

The formula that was used to compute the sum of integers is $I = \sum_{i=1}^{n} I_i$
The visual representation of the parallel processing of integers is shown below for 8 integers:

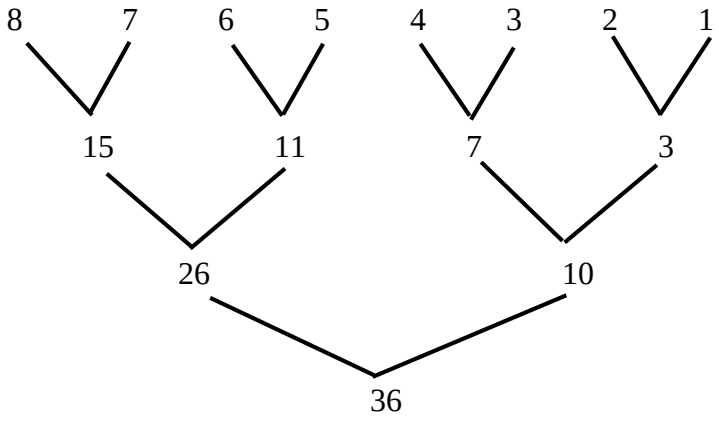

Figure 11: Sum Implementation

The input data mentioned above was considered as an example. If this input was given to the client as below:

Implementation process - Client

```
Data: Random 8 bit integers
```

Circuit to be evaluated:
ia,o0,i0,i1,ia,o1,i2,i3,ia,o2,i4,i5,ia,o3,i6,i7,ia,o4,o0,o1,ia,o5,o2,o3,ia,o6,o4,o5
This input given to the client is converted into arrays as shown below:

Data: For example [8,7,6,5,4,3,2,1] where i0 stands for 8, i1 stands for 7…so on.
o0,o1… are the placeholders to store the outputs for an instruction.
Circuit:
["ia,o0,i0,i1","ia,o1,i2,i3","ia,o2,i4,i5","ia,o3,i6,i7","ia,o4,o0,o1","ia,o5,o2,o3","ia,o6,o4,o5"]
Assuming the server array as below:
["1.1.1.1:1234","2.2.2.2:1234"]

This input is sent to the dispatcher which computes the encyption of the numbers sent to it. So, each of the input elements are converted into a data array of bits after encrypting them.

The entire process of encryption, decryption and key generation has been implemented using HElib code that is based on Gentry Halevi Scheme of encryption. For this process, the following parameters were used to generate the public key and secret key.

Parameter	Value	Description
m	0	Native Plain text space
p	257	The modulo parameter to generate 8 bit integer output
r	1	Native Plain text space
L	4	levels
c	3	Columns in key switching matrix
w	64	Hamming weight of secret key
d	0	Degree of field extension
security	128	Hardness parameter

Table 16: KeyGen Parameters

Please find below the files that have been generated using the above parameters for Public Key and Secret Key that have been used for this implementation.

Using this Public key, the input data is encrypted using FHE. Once the encrypted data bits are obtained, then the instructions are evaluated to build the data dependency graph in the dispatcher followed by finding the subcircuits in the set of instructions.

The order of evaluation of instructions and the data dependency graph for these instructions that are sent looks like below:

Number	Instruction
1	ia,o0,i0,i1
2	ia,o1,i2,i3
3	ia,o2,i4,i5
4	ia,o3,i6,i7
5	ia,o4,o0,o1
6	ia,o5,o2,o3
7	ia,o6,o4,o5

Table 17: List of Instructions for Implementation

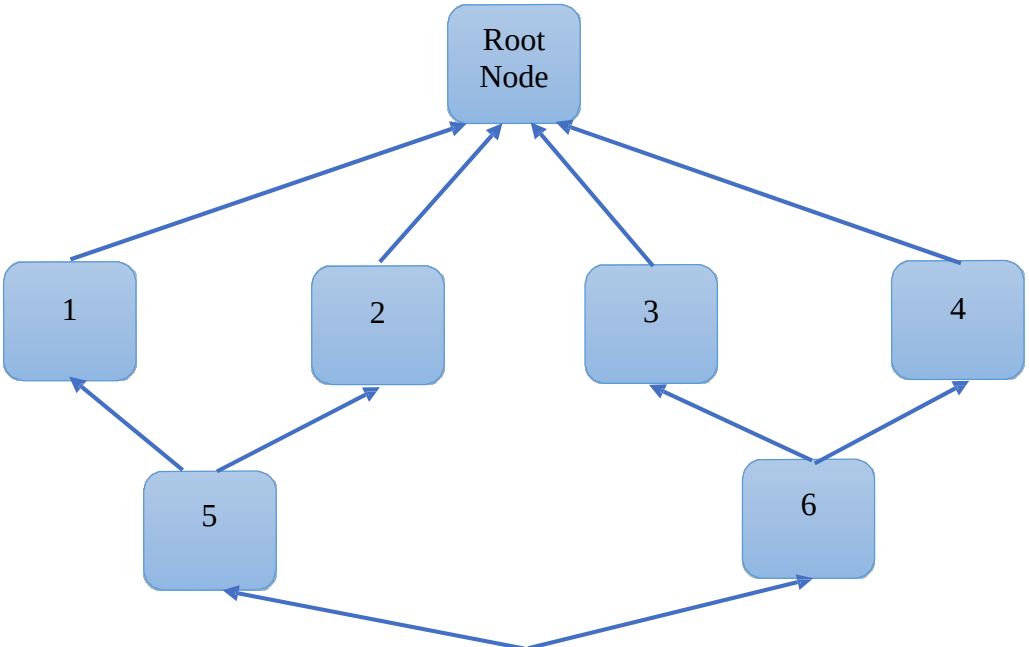

Figure 12: Dependency Graph for Implementation

When implementing the same algorithm for dependency graph using python, the following output has been obtained as shown below:

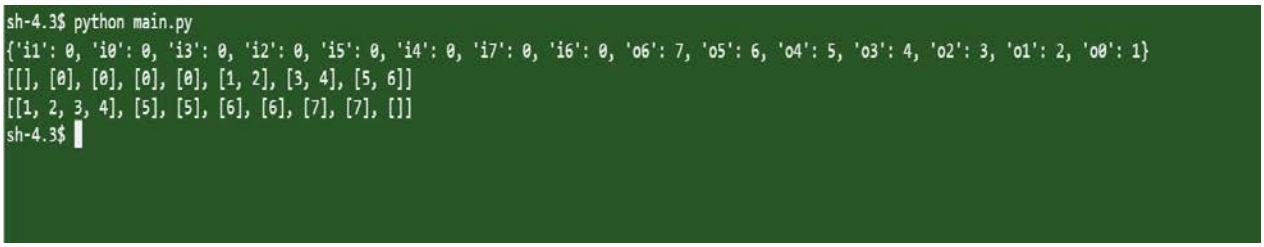

Figure 13: Dependency Graph Output - Screenshot

This window shows basically three arrays. The first one stands for the Data Owner Associative array. The second array shows the parent array P and the third array shows the child array C. The parent array is an array of arrays where the first element stands for the root node, the second one stands for instruction 1 or node 1 that is dependent on node 0 i.e. the root node. Similarly, the fifth node shows that it is dependent on node 1 and 2 for its computation. P and C can be read in the similar way that gives us the dependency graph constructed above.

The time taken for this dependency graph construction is 1.4 milliseconds.

Now that the dependency graph has been obtained, the initial subcircuits or subtasks are found from it to form an execution Queue. A screenshot of the elements of the initial execution Queue executed using python code is shown below:

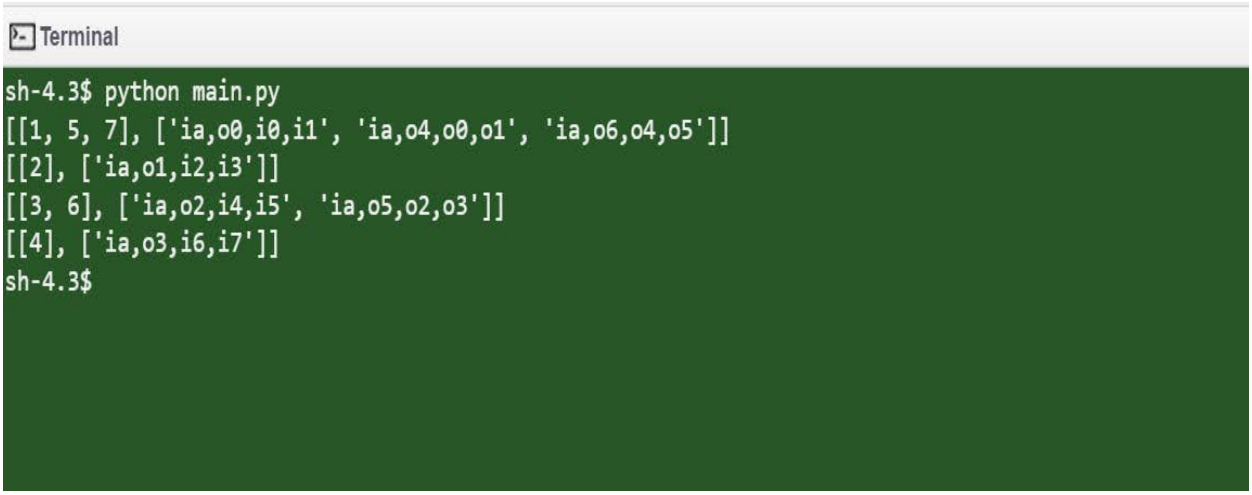

Figure 14: Subcircuits Output - Screenshot

This is the initial execution sequence only. This shows that the computations for node 1, 5 and 7 are executed first in that order followed by the others. This would be revised based on the responses from the servers for each of the computations that have been sent to them.

Once the Subcircuits have been computed, these are sent to multiple servers based on their availability. Two servers have been considered in our implementation that are running in parallel. So, each element in the Queue is sent to these two servers as a request and this Queue is updated based on responses.

For the Server to process the request, the public key, the input data for a circuit and the subcircuit to be evaluated are sent.

The server then performs a bit by bit XOR addition and returns the output to the dispatcher. The dispatcher then runs then updates the dependency graph all over again and evaluates the sequence of pending subciruits. It continues the same process with the servers again until there are no pending subcircuits left in the Queue. Once done, the values are decrypted and are converted to integers again. The C++ main code to generate the keys, encryption of data and decryption of data is shown in Appendix A.

5.3 Evaluation

While executing the above mentioned algorithms, two different scenarios were evaluated. One using sequential processing and the other using parallel processing. In the first case, a

sequential sum of 8 integers is found to set a point of comparison for parallel processing of the same set of data. And in each case, the time taken (in milliseconds) is calculated.

The time taken for sequential addition of integers is shown below:

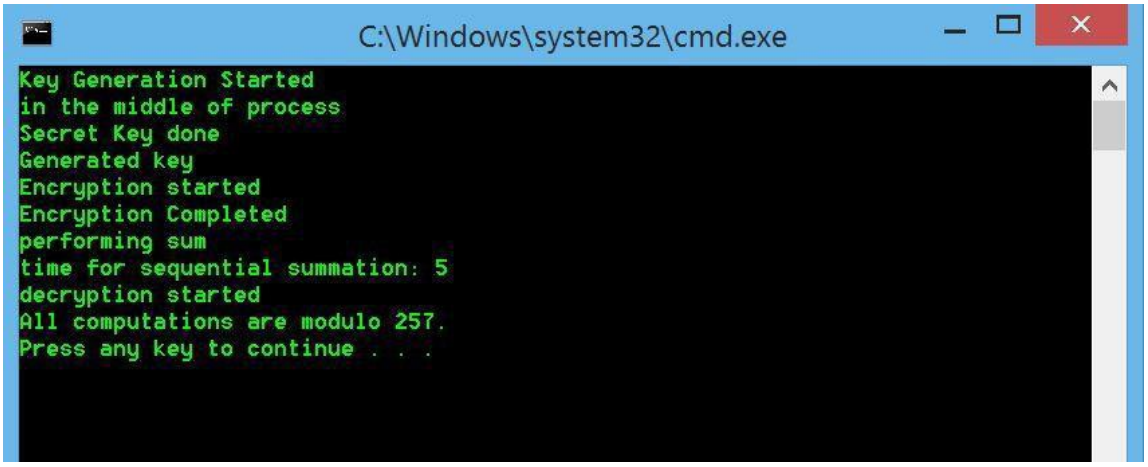

Figure 15: Sequential Processing Time Calculation

For the parallel processing of homomorphic addition, the times taken have been calculated for different sub steps involved and a mathematical derivation has been performed to obtain the approximate total amount of time for the addition of integers. This can be compared to the time calculated for sequential processing to get an overview of difference in processing speed in both cases.

The time taken for building a dependency graph and each subcircuit is recorded by the computational dispatcher. The time taken to create the dependency graph is shown below in seconds.

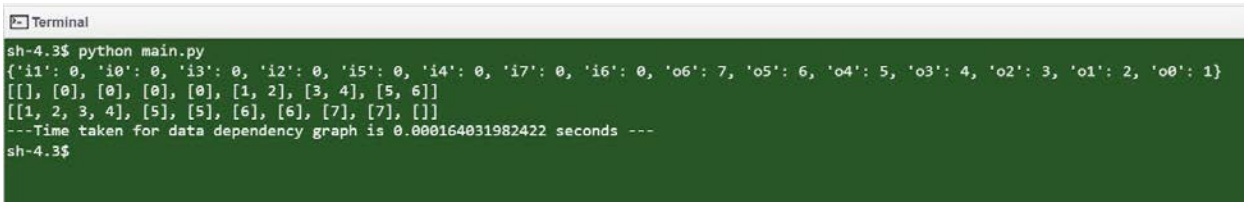

Figure 16: Dependency Graph - Time Calculation

The number of subcircuits involved in the parallel evaluation of sum of 8 integers is 7. The time to find the initial order of execution of subcircuits is as shown below:

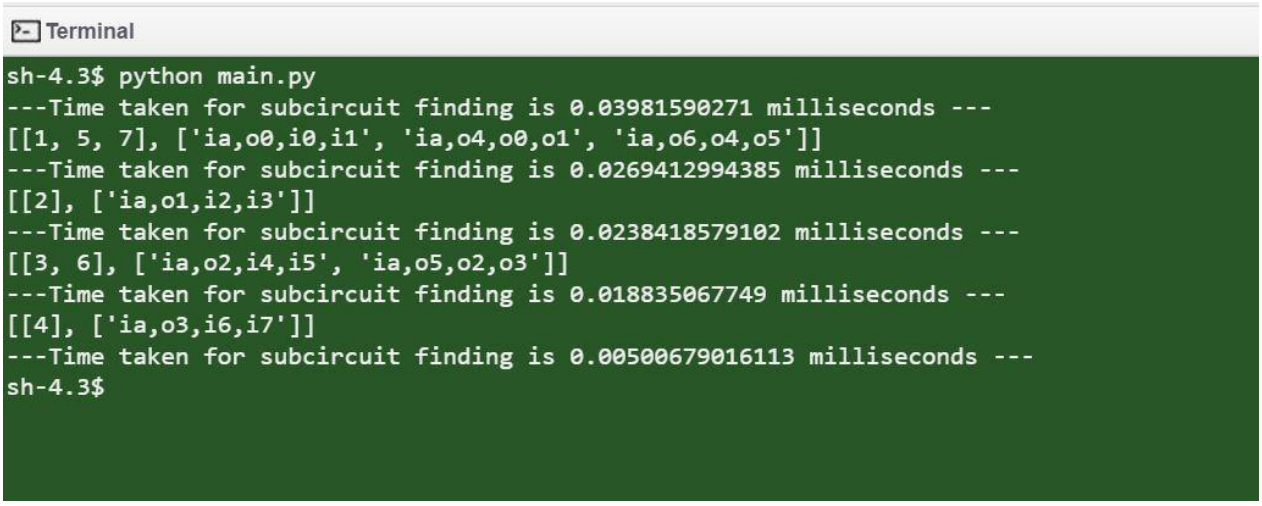

Figure 17: Subcircuit Finding - Time Calculation

Similarly, times for subcircuit findings after each response from server is received, is also computed. All these timings are added to get the total amount of time taken for subcircuit findings.

Parallel execution in Python can be executed using Daemon threads on a single system. The time taken for one subcircuit evaluation i.e. one summation instruction is recorded by the server and is shown below:

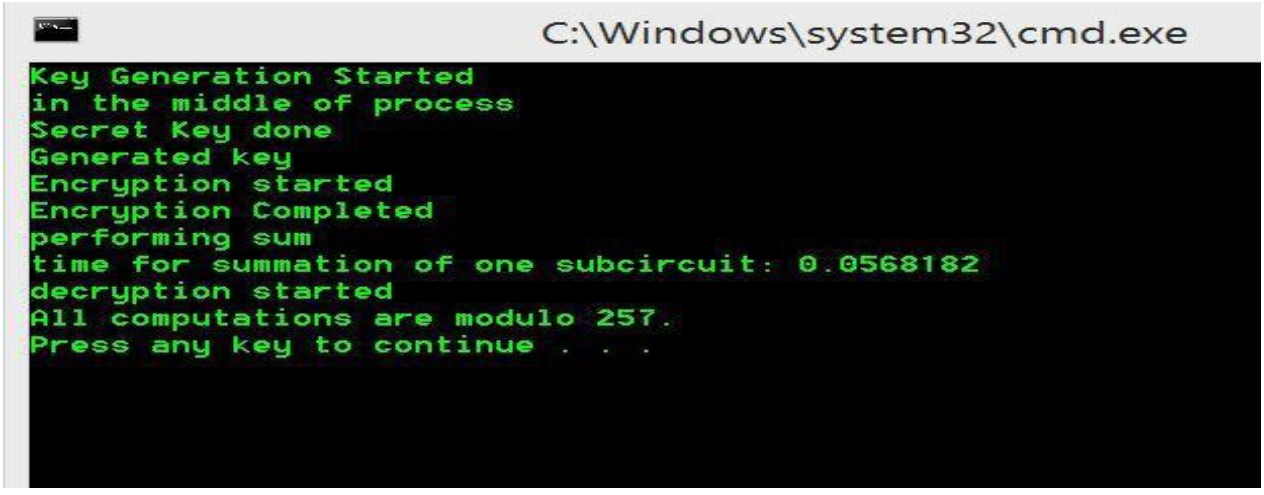

Figure 18: Subcircuit Evaluation - Time Calculation

Mathematically, the time taken to perform all subcircuit evaluations by servers would be approximately 7 times the time taken for one subcircuit evaluation since all the integers considered are 8 bit integers. Eliminating the network delays and data transfer times between two servers and dispatcher for convenience and summarizing the different times taken for processing of homomorphic addition of integers, the values obtained are shown below. The time for Parallel processing of data does not include the time taken to generate the public and private keys, time for the initial encryption of integers and the final decryption of the result. However, the Subciruit evaluation time shown in the table includes the time for bootstrapping (if necessary) invloved during homomorphic evaluation in the servers.

Type of Processing	Number of Servers/Nodes	Operation	Time Taken(ms)
Parallel	**2**	Dependency graph	1.6
		Subcircuit Finding	0.242
		Subcircuit Evaluation	0.398
		Total Time	**2.24**
Sequential	**1**	**Total Time**	**5**

Table 18: Time Calculation - Summary:

Comparing the times taken for homomorphic addition of integers in both cases i.e. sequential and parallel processing, we observe that parallel architecture proves to be a better solution of homomorphic evaluation of data. Parallel evaluation reduces the processing time thereby increasing the speed of the operation. The time for execution reduced from 5 ms to 2.24 ms in parallel processing of homomorphic addition. A graphical representation of the above result is shown below:

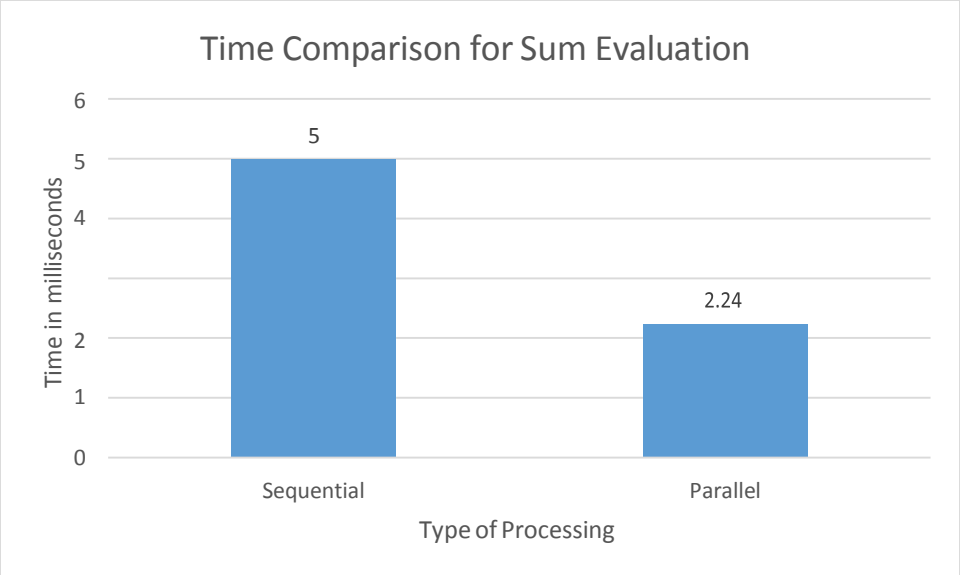

Figure 19: Time Comparison Graph

6 Conclusion

This report presents an architecture for using homomorphic encryption for the addition of integers on the cloud. It is aimed at reducing performance related issues and improving the speed of fully homomorphic encryption[FHE]. The underlying idea for the design is to execute the series of instructions received, in parallel using multiple servers rather than in a sequential manner. The design uses a dependency graph based on the series of instructions received. The subcircuits involved in the parallel execution of instructions are then evaluated. To perform this implementation, the HElib code for encryption, decryption and Keygen has been used. Additional algorithms and code have been written in Python to generate the dependency graph and subciruits. The time taken to perform each step has been calculated in this process and based on a simple mathematics, an approximate total amount of time for Sum evaluation has been calculated. Based on the statistics obtained, this process shows that it could reduce the computational time when multiple servers are used for a large number of computations.

But this design has its own set of limitations. In the report, the time calculation did not take into account, network transfer times. When computations are short and network communications take too much time, it can prove to be a lengthy process. This design also did not consider the scenario when more than two servers are used. More servers in compuation would mean more number of data transfers between the computational dispatcher and servers. This would bump the transfer time and hence might increase the overall Sum evaluation time. Also, this solution could be further explored with other mathematical operations like product etc. to get better results. The design used herein would work if there is a good network speed. Also, the current process could be modified further to not pass the public key to servers each and every time. Doing this reduces the possibility of a security breach. The example design has not been performed on a set of practical real world scenario. This example is just an experimental evaluation. It is definitely a solution worth more exploration.

7 Bibliography

[1] Wikipedia, "Cloud Computing," [Online]. Available: https://en.wikipedia.org/wiki/Cloud_computing. [Accessed 22 January 2016].

[2] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph and R. Katz, "A View of Cloud Computing," Communications of the ACM, 2010. [Online].

[3] "CSA Cloud Security Alliance," 2011. [Online]. Available: https://cloudsecurityalliance.org/guidance/csaguide.v3.0.pdf. [Accessed 23 January 2016].

[4] "The Apprenda Library," Apprenda Inc., 2016. [Online]. Available: https://apprenda.com/library/cloud/. [Accessed March 2016].

[5] Amazon.com, "Types of Cloud Computing," Amazon, [Online]. Available: https://aws.amazon.com/types-of-cloud-computing/. [Accessed 22 January 2016].

[6] Y. Sun, J. Zhang, Y. Xiong and G. Zhu, "Data Security and Privacy in Cloud Computing," *International Journal of Distributed Sensor Networks*, vol. 2014, no. Article ID 190903, 2014.

[7] G. C. Kessler, "An Overview of Cryptography," Auerbach, 1999.

[8] R. L. Rivest, L. Adleman and M. L. Dertouzos, "ON DATA BANKS AND PRIVACY HOMOMORPHISMS," Foundations of secure computation, Massachusetts, 1978.

[9] M. TEBAA and S. EL HAJII , "Secure Cloud Computing through Homomorphic Encryption," *International Journal of Advancements in Computing Technology(IJACT),* vol. 5, no. 16, pp. 29-38, 2013.

[10] S. Choinyambuu, *Homomorphic Tallying with Paillier Cryptosystem*, 2009.

[11] C. Gentry, "A FULLY HOMOMORPHIC ENCRYPTION SCHEME," Stanford Crypto group, STANFORD, 2009.

[12] X. Yi, R. Paulet and E. Bertino, Homomorphic Encryption and Applications, Springer, 2014.

[13] C. Gentry and S. Halevi, "Implementing Gentry's Fully-Homomorphic Encryption Scheme," 2010.

[14] "Securing the Cloud with Homomorphic Encryption," *The Next Wave*, vol. 20, no. 3, pp. 1-4, 2014.

[15] "Python," Python Software Foundation, [Online]. Available: https://www.python.org/. [Accessed 12 April 2016].

[16] S. Halevi, "HElib," Github, Inc., [Online]. Available: https://github.com/shaih/HElib. [Accessed 12 April 2016].

[17] V. Shoup, "Number Theory Library," Wikipedia, [Online]. Available: https://en.wikipedia.org/wiki/Number_Theory_Library. [Accessed 12 April 2016].

8 Appendix A

8.1 Python Code for Generation of Dependency graph

```
def createDataDependencyGraph(E):
    P = [] #Parent Node array
    C = [] #Child Node Array
    D = {}     #Associative array for data owners
    for e in range(0,len(E)+1,1):
        P.append([])
        C.append([])
    for index in range(1,len(E)+1,1):
        e = E[index-1]
        elems = e.rsplit(',')
        op = elems.pop(0)
        inputs = []
        outputs = []
        if op == 'h' or op == 'f':
            outputs = elems[: 2]
            inputs = elems[2: ]
        else:
            outputs = elems[:1]
            inputs = elems[1:]

        for input in inputs:
            if input not in D:
                D[input] = 0
            d = D[input]
            if d not in P[index]:
```

```
                P[index].append(d)
            if index not in C[d]:
                C[d].append(index)

        for output in outputs:
            if output not in D:
                D[output] = 0
            d = D[output]

            if d > 0:
                for dependent in C[d]:
                    if dependent not in P[index]:
                        P[index].append(dependent)
                    if index not in C[dependent]:
                        C[dependent].append(index)
            D[output] = index
    print D
    print P
    print C

## Calling the function with the set of instructions to build
the graph
createDataDependencyGraph(["ia,o0,i0,i1","ia,o1,i2,i3","ia,o2,i4
,i5","ia,o3,i6,i7","ia,o4,o0,o1","ia,o5,o2,o3","ia,o6,o4,o5"])
```

8.2 Python code for Finding Subcircuits

```
import copy
import Queue


pendingExecution = [] #Queue that holds the subtasks pending for
execution
```

```
def findSubCircuit(P,C):
    c=0
    for cc in C[0]:
        if cc not in pendingExecution:
            if len(P[cc])==1 and P[cc][0] == 0:
                c=cc
                break
    if c==0 or len(C[0])==0:
        return []
    pendingExecution.append(c)
    E=[c]
    candidates = copy.deepcopy(C[c])

    while len(candidates)>0:
        c=candidates.pop(0)
        add=True;

        for c_parent in P[c]:
            if c_parent not in E and c_parent !=0:
                add=False;
            if add == True and c not in pendingExecution:
                E.append(c)
                pendingExecution.append(c)
                for member in C[c]:
                    if member not in candidates:
                        candidates.append(member)
    return E
```

```
def findSubs():
    computation =
["ia,o0,i0,i1","ia,o1,i2,i3","ia,o2,i4,i5","ia,o3,i6,i7","ia,o4,
o0,o1","ia,o5,o2,o3","ia,o6,o4,o5"]
    execQ = Queue.Queue()
    while True:

          E=findSubCircuit([[],[0],[0],[0],[0],[1,2],[3,4],[5,6]
          ],[[1,2,3,4],[5],[5],[6],[6],[7],[7],[]])
          if len(E) <=0:
            break
          c=[]
          for e in E:
            c.append(computation[e-1])
          execQ.put([E,c])
     print execQ.get()


findSubs() ## calling function to generate subcircuits
```

8.3 C++ main() Code to generate Keys, Encryption and decryption

```
// Testing.cpp : Defines the entry point for the console
application.
//

#include "stdafx.h"

#include <NTL/ZZ.h>
#include "FHE.h"
#include "EncryptedArray.h"
#include <NTL/lzz_pXFactoring.h>
#include <fstream>
#include <sstream>
#include <sys\timeb.h>

using namespace std;
```

```
/**
*
*/
int main(int argc, char** argv) {

     long m = 0, p = 257, r = 1; // Native plaintext space
                                      // Computations will be
'modulo p'
     long L = 4;          // Levels
     long c = 3;           // Columns in key switching matrix
     long w = 64;          // Hamming weight of secret key
     long d = 0;
     long security = 128;
     ZZX G;
     m = FindM(security, L, c, p, d, 0, 0);
     cout<<"Key Generation Started" << endl;
     FHEcontext context(m, p, r);
     // initialize context
     buildModChain(context, L, c);
     // modify the context, adding primes to the modulus chain
     FHESecKey secretKey(context);
     // construct a secret key structure
     const FHEPubKey& publicKey = secretKey;
     // an "upcast": FHESecKey is a subclass of FHEPubKey
     cout << "in the middle of process" << endl;
     //if(0 == d)
     G = context.alMod.getFactorsOverZZ()[0];

     secretKey.GenSecKey(w);
     cout << "Secret Key done" << endl;
     // actually generate a secret key with Hamming weight w
     addSome1DMatrices(secretKey);
     cout << "Generated key" << endl;

     EncryptedArray ea(context, G);
     long nslots = ea.size();

     vector<long> v1;
     for (int i = 0; i < nslots; i++) {
          v1.push_back(i*2);
     }
     cout << "Encryption started" << endl;
     Ctxt ct1(publicKey);
     ea.encrypt(ct1, publicKey, v1);

     vector<long> v2;
     Ctxt ct2(publicKey);
```

```
	for (int i = 0; i < nslots; i++) {
		v2.push_back((i*2) + 1);
	}
	ea.encrypt(ct2, publicKey, v2);

	cout << "performing sum" << endl;
	Ctxt ctSum = ct1;
	ctSum += ct2;

	cout << "decryption started" << endl;

	vector<long> res;
	ea.decrypt(ctSum, secretKey, res);

	cout << "All computations are modulo " << p << "." << endl;
	ofstream myfile;
	myfile.open("IntegerSumOutputFile.txt");

	for (int i = 0; i < res.size(); i++) {
		myfile << v1[i] << " + " << v2[i] << " = " << res[i]
<< endl;
	}
	myfile.close();
	return 0;
}
```

8.4 Abbreviations and symbols

Abbreviation	Full Form
AWS	Amazon Web Service
NTL	Number Theory Library
SHE	Somewhat Homomorphic Encryption
FHE	Fully Homomorphic Encryption
RSA	Rivest, Shamir and Adelman
API	Application Programming Interface
IaaS	Infrastructure as a Service
PaaS	Platform as a Service
SaaS	Software as a Service

Table 19: Abbreviations and Symbols