Introduction to Homomorphic Encryption

Harry Davis

May 3, 2024

1 Abstract Algebra Introduction:

1.1 Definition: Group

A group is a set X equipped with an operation $\cdot$ which satisfies the following properties:
Associativity: $(a \cdot b) \cdot c = a \cdot (b \cdot c)$ for $a, b, c \in X$
Identity: There exists a unique element $e \in X$ satisfying $a \cdot e = e \cdot a = a$ for all $a \in X$
Inverse: For $a \in X$, there exists a unique element $a^{-1} \in X$ satisfying $a \cdot a^{-1} = a^{-1} \cdot a = e$
Additionally, if $\cdot$ is commutative, $a \cdot b = b \cdot a$ for all $a, b \in X$, we call X an abelian group.

1.2 Group Examples:

The set of integers modulo a positive number n is an abelian group under addition: For $a, b \in \mathbb{Z}/n\mathbb{Z}$ $a+b = nr+c = c \mod n$ for some $r, c \in \mathbb{Z}$. Addition commutes over $\mathbb{Z}$, so $b+a = nr+c = c \mod n$, which shows commutativity. Associativity is shown similarly. 0 is the unique identity element and $a^{-1} = n - a$.

Let n a positive integer and consider the set of positive integers relatively prime with n. This set forms an abelian group under multiplication denoted by $(\mathbb{Z}/n\mathbb{Z})*$: Let $a, b \in (\mathbb{Z}/n\mathbb{Z})*$. Then $ab = (n_0r_0 + c_0)(n_1r_1 + c_1)$. Terms in the product with a factor of n go to 0 mod n, so $ab = c_0c_1 \mod n$. Similarly, $ba = c_0c_1 \mod n$, so the operation is commutative. Both a and b are relatively prime with n, so ab is also relatively prime with n. Associativity is shown similarly using the associativity of multiplication over $\mathbb{Z}$. Also, 1 is an identity element. Now suppose $ab = ac \mod n$, with a, b, c relatively prime with n. Then $ab - ac = a(b - c) = 0$ mod n, so $b = cmodn$. Thus, the set $\{ax : x \in (\mathbb{Z}/n\mathbb{Z})*\} = (\mathbb{Z}/n\mathbb{Z})*$, so there exists a unique x satisfying $ax = 1$. So $(\mathbb{Z}/n\mathbb{Z})*$ is an abelian group under multiplication.

1.3 Definition: Homomorphism

A homomorphism is a map $f : A \to B$, where A and B are groups, satisfying $f(x \cdot y) = f(x) \cdot f(y)$ for all $x, y \in A$, and where $\cdot$ denotes the group operations in A and B (which may be distinct). Homomorphisms are maps that preserve the algebraic structure between groups. For some intuition behind this idea, suppose $x \in A$ satisfies n is the least positive integer satisfying $x^n = e$, where e is the identity in A and x^n represents $x \cdot x \cdot x$ ... where the operation is applied n times. We define n to be the order of x. Then $f(e) = f(x^n) = f(x)^n$. Now $f(x) = f(x \cdot e) = f(x) \cdot f(e)$ shows that $f(e)$ is the identity element in B, and thus $f(x)$ has order less than or equal to n in B. Now let i denote the order of $f(x)$ in B, and suppose i doesn't divide n. We have $e = f(x^n) = f(x)^i \cdot f(x)^{n-i} = e \cdot f(x)^{n-i}$, so $f(x)^{n-i} = e$. Because i is the order of $f(x)$ and i doesn't divide n, $n - i > i$. Now $(f^{-1}(x))^i = f(x)^{-i} = e$ and consider $f(x)^{n-i} \cdot f(x)^{-ci} = f(x)^{n-i(c+1)}$, where c is the least positive integer such that $n - i(c + 1) < i$ Because i doesn't divide n, numbers of the form $n - i(c + 1)$ are never 0 mod i, so $n - i(c + 1)$ doesn't equal 0. Now if $n - i(c + 1) < 0$ the i consecutive numbers 0 through (i - 1) do not contain an integer of the form $n - i(b+1)$ where b is a positive integer, but this contradicts the pigeonhole principle. Thus, there must be some positive integer z between 0 and i that

satisfies $z = (n - i(c + 1))$. But then $f(x)^z = e$ contradicts that i is the order of $f(x)$. Thus, the order of $f(x)$ divides the order of x. With this example, we see that a homomorphism of groups can give us specific information on the orders of elements in B in relation to the orders of elements in A. Can you find other relationships?

1.4 Homomorphism Examples:

Let A the group $\mathbb{R}$ under addition and S^1 the group of complex numbers with magnitude 1 under multiplication. Then $f(x) = e^{ix}$ is a homomorphism from $\mathbb{R}$ to S^1. We have $f(x+y) = e^{i(x+y)} = e^{ix}e^{iy} = f(x)f(y)$. Consider $\mathbb{Z}$ under addition and $\mathbb{Z}/2\mathbb{Z}$ under addition mod 2. Then $f(x) = x$ is a homomorphism, as $f(n_0 + n_1) = f(n_0) + f(n_1) = n_0 + n_1 \mod 2$.

1.5 Isomorphism and Injective Homomorphism:

In cryptography, encryption functions should be one to one, otherwise a decryption function would have to decrypt the encryptions of different messages to just one message. Thus, the homomorphisms useful in cryptography should be one to one. This type of homomorphism is called an injective homomorphism. Because an isomorphism is a bijective homomorphism, an injective homomorphism from a A to B can be viewed as an isomorphism from A onto a subgroup of B. The properties of ismorphisms are stronger than those of homomorphisms; algebraically, groups that are isomorphic to each other are considered the same, or considered to have the same algebraic structure. For instance, in 1.3 we showed that if x has order n in A, and f is a homomorphism, the order of $f(x)$ in B divides the order of x. If f is instead an isomorphism, the order of $f(x)$ in B must be equal to n.

2 What is Homomorphic Encryption?

2.1 Overview:

Homomorphic encryption uses homomorphic encryption functions to allow operations on data without decryption. For instance, let a message space be a group under multiplication and take an encryption function satisfying $f(m_1m_2) = f(m_1)f(m_2)$ for all messages, that maps the message space into a group with multiplication operation. Now suppose you have an encryption of m_1 stored on a server, and you want to edit this encryption to be an encryption of m_1m_2. If the encryption function is not homomorphic, you may need to decrypt the encryption of m_1, compute m_1m_2, and then compute the encryption of m_1m_2, or give the server power to decrypt your messages. However, if the server is compromised, an attacker may be able to steal m_1 during this process. With a homomorphic encryption function, you can simply send the encryption of m_2 to the server and have it compute the product of the encryptions of m_1 and m_2. Even further, assume you want to consistently update your encryption. If your encryption function is not homomorphic, you will have to repeat this decryption process numerous times, but with homomorphic encryption, you can edit your message without ever having to decrypt it.

Homomorphic encryption schemes are generally classified as partially homomorphic, somewhat homomorphic, or fully homomorphic. Partially homomorphic schemes are encryption schemes that support homomorphic evaluation of one type of operation, for instance, multiplication or addition. Somewhat homomorphic schemes support finitely many evaluations of both addition and multiplication. Fully homomorphic schemes support unlimited homomorphic computations of both addition and multiplication.

2.2 Partially Homomorphic Encryption:

Although partially homomorphic crypto schemes limit the computations that can be done homomorphically, they still have some applications. Furthermore, partially homomorphic schemes are much less computationally expensive than fully homomorphic crypto schemes, so they are useful in applications that only require one homomorphic operation. We present two examples of partially homomorphic schemes below.

RSA Encryption Scheme:
The encryption function for RSA is $f(m) = m^e \mod n$ where n is the product of 2 large primes. Thus, $f(m_1 m_2) = (m_1 m_2)^e \mod n = (m_1)^e (m_2)^e \mod n = (m_1^e \mod n)(m_2^e \mod n) = f(m_1)f(m_2)$, so RSA provides homomorphic computation of multiplication. Although RSA is partially homomorphic by definition, its use is not intended for homomorphic computation.

Paillier Encryption Scheme:
The Paillier Cryptosystem is an additively homomorphic encryption scheme. For encryption, pick two large primes and compute the product n, choose a random $r < n$ that is relatively prime with n, and encrypt the message $m < n$ by computing $f(m, r) = (n+1)^m (r)^n \mod n^2$. It can be verified using the binomial theorem and properties of the euler totient function that decryptions of $f(m, r) = m$ for all r. Verifying the additive homomorphism, we have $f(m_1, r_1)f(m_2, r_2) = (n+1)^{m_1}(r_1)^n (n+1)^{m_2}(r_2)^n \mod n^2 = (n+1)^{m_1+m_2}(r_1 r_2)^n \mod n^2 = f(m_1 + m_2, r_1 r_2)$ which decrypts to $m_1 + m_2$.

The Paillier Cryptosystem was developed with intentions of creating an additively homomorphic cryptosystem, and, in general, partially homomorphic cryptosystems that are additive have clearer and more practical applications. For instance, consider a voting system where each voter encrypts their own vote of 0 or 1 and sends it off to be tabulated. If the tabulator used the Paillier Cryptosystem, it could compute the product of the encryptions of each vote, and return the decryption of this product to give the sum of votes without any individuals vote being revealed.

2.3 Somewhat Homomorphic Encryption:

Somewhat Homomorphic Encryption schemes support a finite number of operations of both addition and multiplication. These encryption schemes are mainly discussed and used in the process of building Fully Homomorphic encryption schemes. Recall that the Paillier Cryptosystem uses a random integer r when computing encryptions, which ensures that encryptions of m vary. Somewhat homomorphic encryptions schemes incorporate randomness into encryptions, called noise. These encryption schemes only support finitely many operations because after some number of operations, the noise grows too large and decryptions do not return the original message. To illustrate, we provide an example of a somewhat homomorphic "toy-scheme" resembling those used to create a fully homomorphic encryption scheme:

Let $m \in \{0,1\}$ and $f(m) = n + 2r + m$, where n is a large odd integer and $r < n$ is a randomly selected integer. We decrypt $f(m)$ by first taking $n + 2r + m \mod n$ and then taking $2r + m \mod 2$. We have $\sum_{i=1}^k f(m_i, r_i) = kn + 2\sum_{i=1}^k r_i + \sum_{i=1}^k m_i = 2\sum_{i=1}^k r_i + \sum_{i=1}^k m_i \mod n$. When $2\sum_{i=1}^k r_i$ is less than n, $2\sum_{i=1}^k r_i$ is a multiple of 2 $\mod n$ and the decryption is equal to $\sum_{i=1}^k m_i \mod 2$. However, if $2\sum_{i=1}^k r_i$ is greater than n, $2\sum_{i=1}^k r_i$ may be odd $\mod n$ and the decryption would calculate $1 + \sum_{i=1}^k m_i \mod 2$. Thus, in the "toy-scheme" finitely many homomorphic additions can be made until the noise becomes too loud, and the decryption fails.

2.4 Fully Homomorphic Encryption:

The first fully Homomorphic Encryption scheme was created in 2009 by Craig Gentry, when he modified a somewhat homomorphic encryption scheme into a fully homomorphic encryption scheme. He used a technique called "bootstrapping" to reduce the noise generated by the somewhat homomorphic encryption scheme and allow for unlimited homomorphic additions and multiplications. Gentry's implementation was too inefficient to be put to practical use (30 minutes per basic bit operation), but it proved that fully homomorphic schemes are possible. Over the past 15 years, the efficiency of fully homomorphic schemes has increased drastically, but the efficiency is not to the level that allows these encryption schemes to be implemented on a wide-scale level. Many open source fully homomorphic encryption libraries are currently operational, Microsoft SEAL being a popular example. Microsoft SEAL offers the homomorphic encryption schemes BFV, BGV, and CKKS. BFV and BGV can perform arbitrary homomorphic computations on integers and BGV and perform arbitrary approximate calculations on real and complex numbers. These schemes are too complex to be explained in detail for this report.

There are many potential applications of fully homomorphic encryption that may be incorporated if the encryption schemes can be made efficient enough. For example, fully homomorphic encryption could allow for secure queries of encrypted databases. A client can encrypt their query and send it to the server, and the server could process the encrypted query, retrieving data without ever revealing the query. Thus, users could access potentially sensitive information discreetly. For a second example, fully homomorphic encryption would allow for encrypted medical data analysis, letting clients encrypt medical information and have diagnoses securely computed on the server side. For instance, if a client smokes cigarettes and wants to understand their health risks related to smoking, but does not want their smoking status to affect their life insurance rates, their smoking information would be secure as it would always be encrypted on the server. A demonstration of this idea is shown in section 3.

3 Demonstrations of Homomorphic Encryption:

3.1 RSA Secret Multiplication:

Using the python lightphe library, it is easy to produce an example of secret multiplication using RSA encryption. In the example below, the code encrypts the numbers 50 and 20, computes the product of their encryptions, and decrypts that product, returning the product of 50 and 20 (1000). This is essentially a secret multiplication calculator, if we envision that a server is computing the products of encryptions, as the server only sees encryptions of 50 and 20. Of course, this secret calculator is limited in power, because RSA only supports homomorphic computation of multiplications.

```
from lightphe import LightPHE

cs = LightPHE(algorithm_name = 'RSA')

m1 = 50
m2 = 20
c1 = cs.encrypt(m1)
c2 = cs.encrypt(m2)
print("Encryption of 50:", c1, "\n")
print("Encryption of 20:", c2, "\n")
c3 = c1*c2
print("Product of Encryptions:", c3, "\n")
print("Decryption of Product:", cs.decrypt(c3))
```

```
PS C:\Users\hdavi> python RSA.py
Encryption of 50:  Ciphertext(288516763589981533849632007119656830506530075603740132026040149925408076856970744212971
133511760718034444895422257387346625684527273735876107905516395692311250352573173920655410017576354039875971108714931
431085690922650004501973173530180259934442797441153723821529934729044970263200602988946181327274 87)

Encryption of 20:  Ciphertext(538609921295851715522371285622019992232459418286888787929404569088763808777872241449831
537859139037997223721608611632513356895954556551825617762003741668094549141507879107971644838581083395372553508693760
790197590913074975598811082115912890251297265634683985820314562886100299953551615908669691325885 67)

Product of Encryptions:  Ciphertext(463221031934652421959592173458346230420352039607135607797879167434577490948073312
190167685095407078560705093410013458360216676957846368342235578826698102233169464444142150024046492506988018989927260
802357003443721223087116840110791784779444692454794732448518193464443828963611701464338392809786733277 93)

Decryption of Product:  1000
PS C:\Users\hdavi>
```

Figure 1: Console Output

3.2 Paillier Secret Binary Voting:

Again using the lightphe library, we can produce an example of secret binary voting. In the example below, 30 votes are randomly computed as 0 or 1, and are all encrypted before being stored in an array. The product of the encryptions of the 30 votes is computed, and the decryption of this product gives the number of votes for 1. This can be viewed as 30 individuals encrypting their vote and passing it to a tabulator, with the tabulator calculating the product of the encryptions of all votes and returning the encrypted result without knowing individual votes. Each individual can decrypt and view the final vote count, but individuals only know their own vote.

```
import random
from lightphe import LightPHE



cs = LightPHE(algorithm_name = 'Paillier')


encVotes = []
for i in range(30):
    num = random.randint(0,1)
    encNum = cs.encrypt(num)
    x = i+1
    print("Encryption of vote" ,x, ":" , encNum)
    print("\n")
    encVotes.append(cs.encrypt(num))
total = cs.encrypt(0)
for i in range(30):
    total = total + encVotes[i]

print("Encrypted sum of votes:", total, "\n")
decTotal = cs.decrypt(total)
```

```
print("Decrypted sum of votes:", decTotal)
```

```
(4708762232195273709158359518472889594813465670224407387322377730960532649940468991511315441674452127911226074212217414860
08090119610877748469821115575157749489267832439355)

Encryption of vote 29 :  Ciphertext(1242847993690898230411986557717763946519714481561656512013809734210939348014349061790
04311831104204960910584421059103966869471246215362602175041849261015779219384902598999312793271420778318804156979682538 5
96965104650979356030287434818780903229461858548113453495481014726810963935020845483321151030057618868135387238005346557 5
66669844725777872869089392642135632640011392262898409426516753330370060153160777259584213688913106755890907816571952692 2
19275412789948961722947223076129491456174864124301322936173107908828048099446957627307838408378882608168144072187990583 1
20462514382298859546964862375275383600281779655101)

Encryption of vote 30 :  Ciphertext(1826221067611816250218194707800372175158507792874870149618390176733482932445769478258
09450326748095965522550217728912145146978964151754528401164549686205342866380745925616667016863398285869895922990282744
72814470851188801467996887721361135171051067708749761009450367569939282642823503539937768950561852499854091466274010318 4
51666547022412996334979124897120101808775124092015418077535150052507134407237302971607705369365522874007993017004039743 9
04308787886256873243467722960196512201931739381650448104680462778747572706470585437058552082581402883589818667677630914 9
58565083910779887427055755446655227584957743926834 7)


Encrypted sum of votes:  Ciphertext(7840722214806921789064173317693742307056570680332124189791414677924536777796295562 99
93662973245342812560425704363925775819536144037520548507734183159195709081871784411472126821910815750057970417401252497 1
12551873283983460281860348246811939612343863760300667154221271085805968268448645079802439087581311741132975200493555343 4
12892396459518271923985854037185672230456655154747458344319568759861736701709155479838895310434639889291566227954430639 8
99247218480034200462691822329567456738501289325383543484577535200261295782864610773550421260002830142734482700004951422 2
50614939355448405087168473422881548476705331671671)

Decrypted sum of votes:  15
PS C:\Users\hdavi>
```

Figure 2: Console Output

3.3 Morfix Demonstration of Microsoft Seal:

Although downloading and using fully homomorphic encryption libraries is complex, we can experiement with fully homomorphic encryption schemes using Morfix, a web based user interface demonstrating fully homomorphic encryption schemes from the Microsoft Seal library. In the demonstration below, we use the BFV scheme, create encryption parameters and keys, and intitalize 3 plaintexts with integer 2-vectors. We encrypt each plaintext, and compute the product of encryptions of A and B, and then add this to the encryption of C. When we decrypt the final computation, we are returned we are returned the 2-vector corresponding to component wise multiplication of A and B added component wise to C.

Sandbox

Experiment with homomorphic encryption powered by Microsoft SEAL. All cryptographic operations occur in the browser thanks to node-seal. To see what client to server interactions look like, sign up to be a beta tester.

Encryption Parameters

These parameters can have drastic performance effects if not properly optimized. Most of these settings are arbitrarily defined and are safe to use; however, you are encouraged to optimize them.

Scheme Type
bfv
Select a Scheme Type

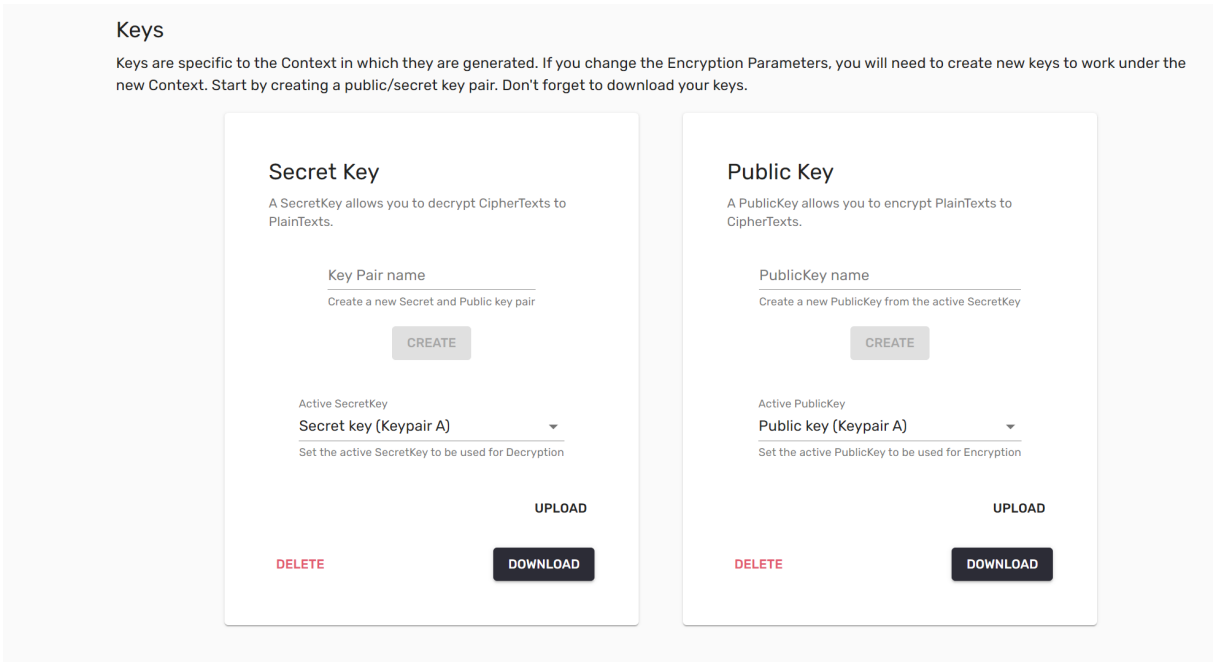
Security Level
128 Bits
Higher values allow for better security at the cost of more CPU and Memory resources

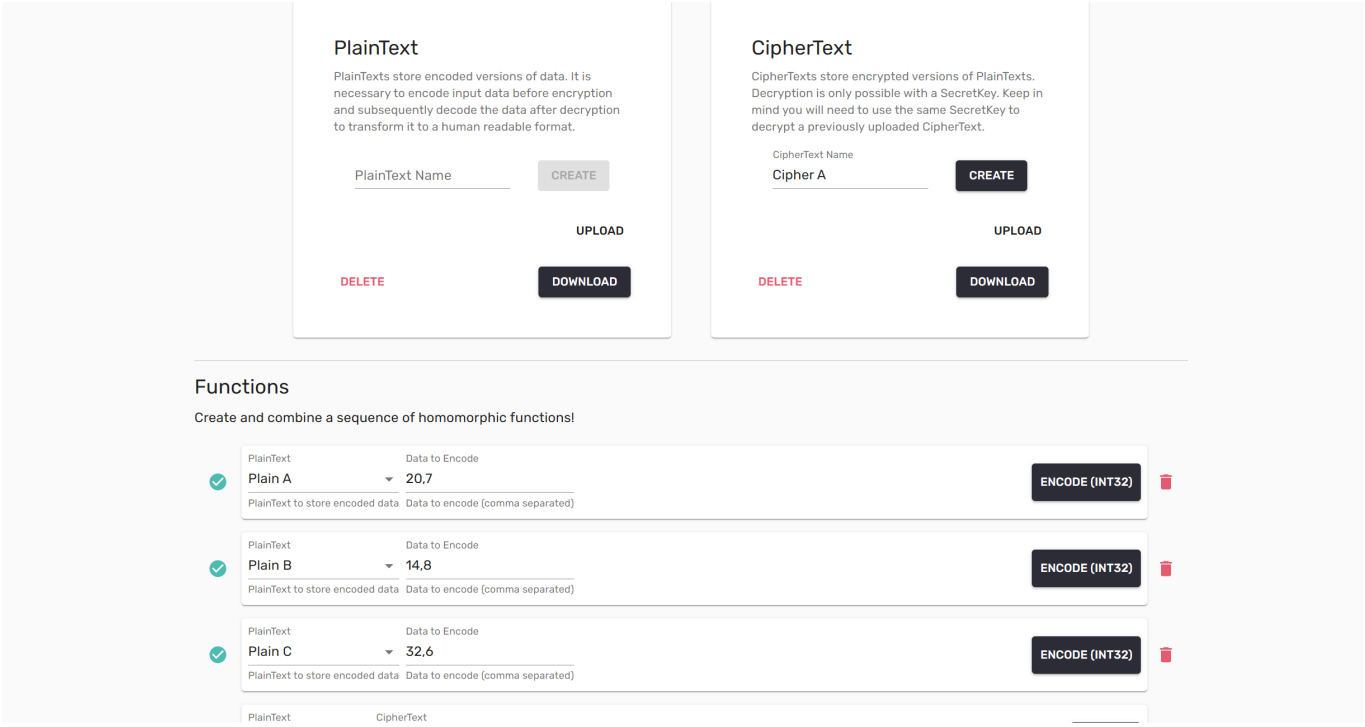
Polymodulus Degree
4096 Bits
Higher values allow for more computations at the cost of more CPU and memory resources

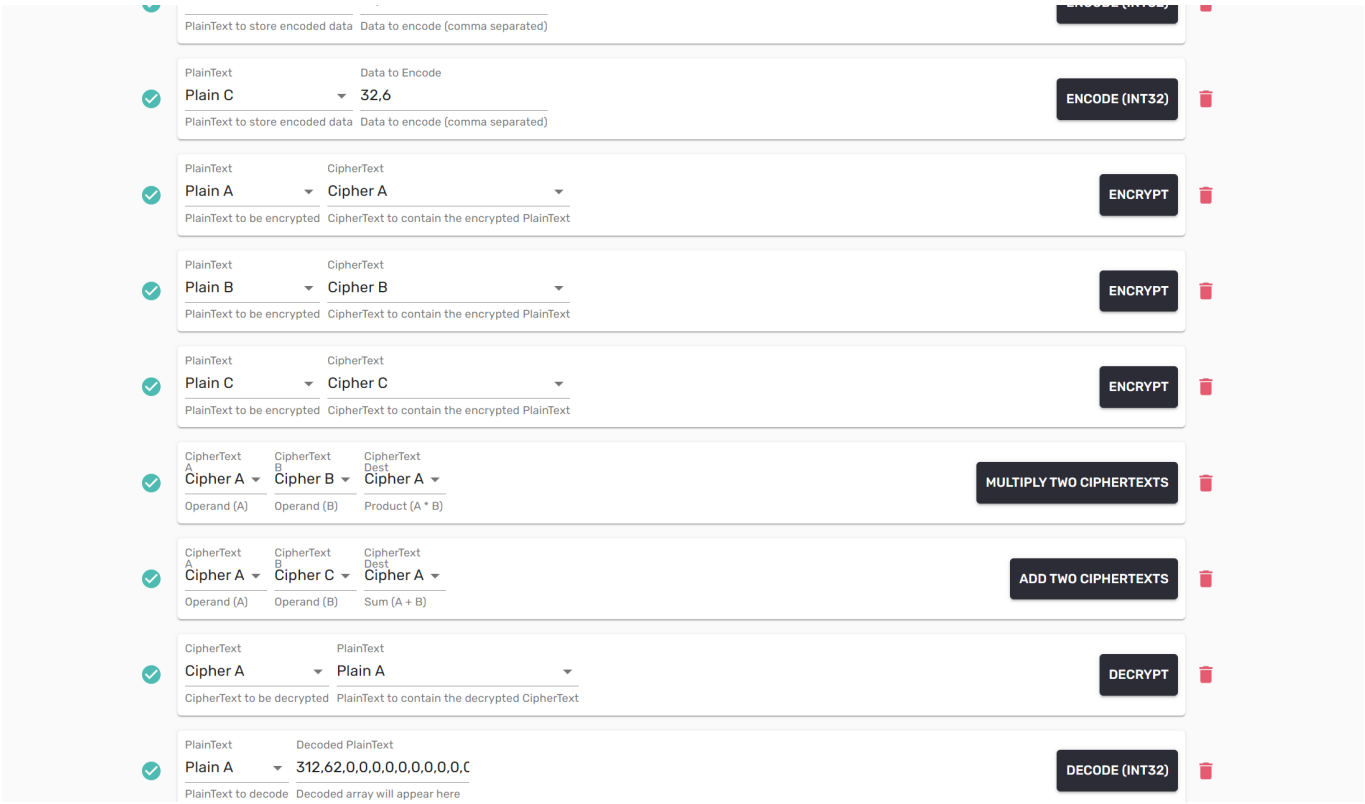
Coefficient Modulus (Bit Sizes)
36,36,37
The bit-lengths of the primes to be generated for the Coefficient Modulus (comma separated). Max 60 bits.

Plain Modulus (Bit Size)
20
The bit-length of the prime to be generated for the Plain Modulus that will enable Batching. Max 60 bits.

☑ Expand Modulus Chain


Keys
Keys are specific to the Context in which they are generated. If you change the Encryption Parameters, you will need to create new keys to work under the new Context. Start by creating a public/secret key pair. Don't forget to download your keys.
Secret Key
A SecretKey allows you to decrypt CipherTexts to PlainTexts.
Key Pair name
Create a new Secret and Public key pair
CREATE
Active SecretKey
Secret key (Keypair A)
Set the active SecretKey to be used for Decryption
UPLOAD
DELETE
DOWNLOAD
Public Key
A PublicKey allows you to encrypt PlainTexts to CipherTexts.
PublicKey name
Create a new PublicKey from the active SecretKey
CREATE
Active PublicKey
Public key (Keypair A)
Set the active PublicKey to be used for Encryption
UPLOAD
DELETE
DOWNLOAD

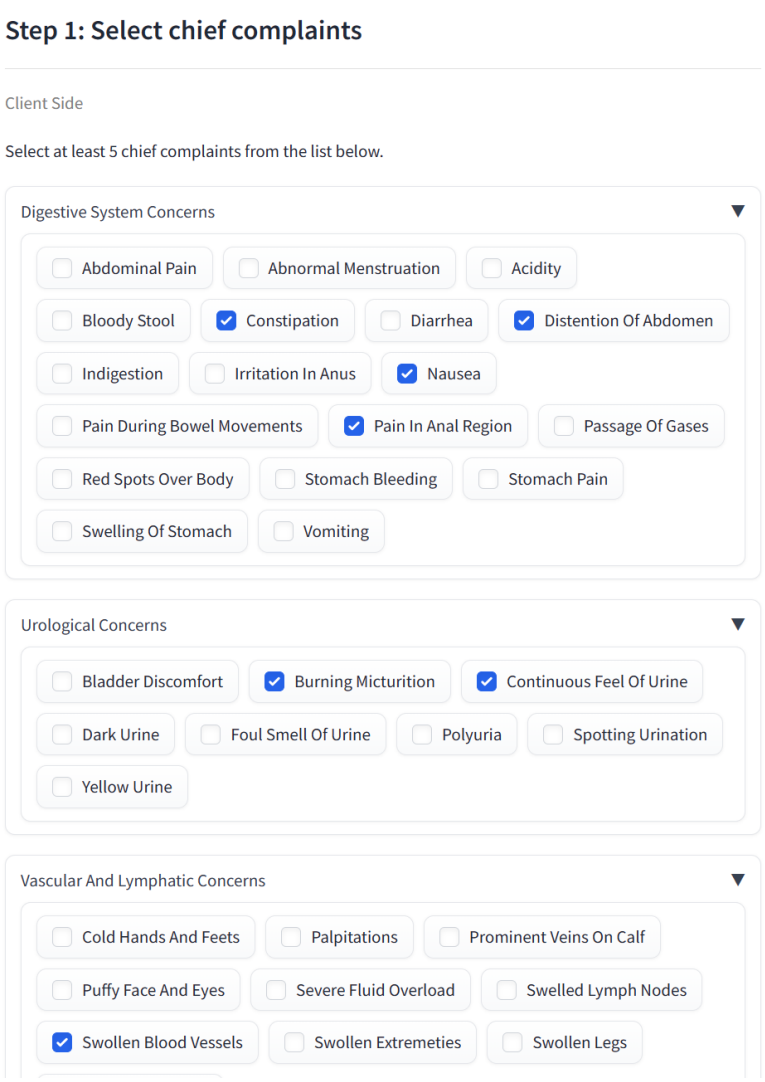

PlainText
PlainTexts store encoded versions of data. It is necessary to encode input data before encryption and subsequently decode the data after decryption to transform it to a human readable format.
PlainText Name
CREATE
UPLOAD
DELETE
DOWNLOAD
CipherText
CipherTexts store encrypted versions of PlainTexts. Decryption is only possible with a SecretKey. Keep in mind you will need to use the same SecretKey to decrypt a previously uploaded CipherText.
CipherText Name
Cipher A
CREATE
UPLOAD
DELETE
DOWNLOAD
Functions
Create and combine a sequence of homomorphic functions!
PlainText
Plain A
PlainText to store encoded data
Data to Encode
20,7
Data to encode (comma separated)
ENCODE (INT32)
PlainText
Plain B
PlainText to store encoded data
Data to Encode
14,8
Data to encode (comma separated)
ENCODE (INT32)
PlainText
Plain C
PlainText to store encoded data
Data to Encode
32,6
Data to encode (comma separated)
ENCODE (INT32)
PlainText
CipherText



3.4 ZAMA Demonstration of Secure Medical Diagnosis:

The following example demonstrates how fully homomorphic encryption can compute a medical diagnosis based off of reported symptoms while keeping the reported symptoms encrypted. The application has the patient check off a list of symptoms, creates a binary-valued vector representing the symptoms, encrypts this data, and sends it to the server. The server uses machine learning and fully homomorphic encryption to compute a diagnosis only using the encrypted representation of the symptoms. The server sends back the encrypted diagnosis, allowing the patient to decrypt and view the diagnosis. Thus, the patient's inputted symptoms remain secure, even while on the server.

- The evaluation key is a public key that the server needs to process encrypted data.



Step 2: Encrypt data

Client Side

Key Generation

In FHE schemes, a secret (enc/dec)ryption keys are generated for encrypting and decrypting data owned by the client.

Additionally, a public evaluation key is generated, enabling external entities to perform homomorphic operations on encrypted data, without the need to decrypt them.

The evaluation key will be transmitted to the server for further processing.

Keys have been generated ✅

Encrypt the data

Encrypt the data using the private secret key

User Symptoms Vector:

```
[[0 0 0 0 0 0 0 0 0 1 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 1 0
  0 0 1 0 0 0 0 1 1 0 0 0 1 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 1 0 1 0 0 0
  0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0
  0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0]]
```

Encrypted Vector:

```
010000001cd203000300000000000000000000000000030008000000000001000c480f00010001001
c480f000100010001000000e0000000100000000280f4005bc239b78ea2af8e0d0de8be065a38bd07
97495d07747e0918444dc637e27116122dbdd323644d5b7cef5f5c6ae82a8040616818ffc42ba95b7
4bd9195b572f94502a1ceced886e0f2a11852cac5d2ca33b5d466472105e2e4ade111efe28843bcbb
4ae5108ed62d471936907762f0b20437479d9ce43ad60ac6db143afbd4014773ebf2793fab60
```

Send the encrypted data to the Server Side

Send data

☑ Data Sent

Step 3: Run the FHE evaluation

Server Side

Once the server receives the encrypted data, it can process and compute the output without ever decrypting the data just as it would on clear data.

This server employs a Logistic Regression model that has been trained on this data-set.

Run the FHE evaluation

Total FHE Execution Time:

0.30 seconds

Step 4: Decrypt the data

Client Side

Get the encrypted data from the Server Side

Get data

Data Received

Decrypt the output

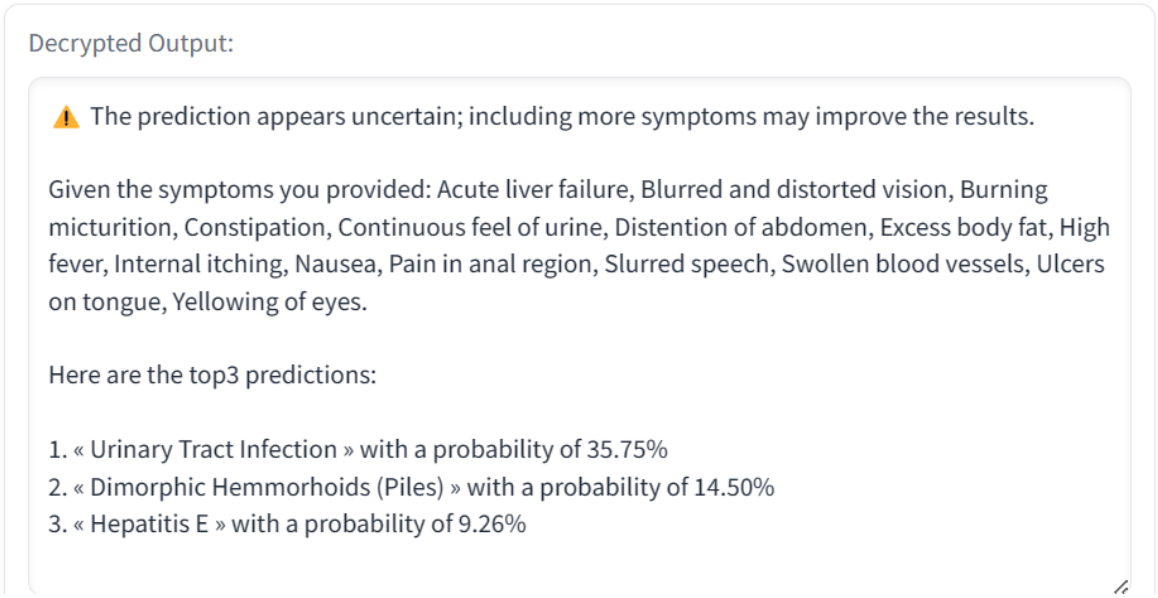
Decrypt the output using the private secret key

Decrypted Output:

⚠️ The prediction appears uncertain; including more symptoms may improve the results.

Given the symptoms you provided: Acute liver failure, Blurred and distorted vision, Burning micturition, Constipation, Continuous feel of urine, Distention of abdomen, Excess body fat, High fever, Internal itching, Nausea, Pain in anal region, Slurred speech, Swollen blood vessels, Ulcers on tongue, Yellowing of eyes.

Here are the top3 predictions:



4 Conclusion:

In conclusion, fully homomorphic encryption can increase data security in numerous applications if encryption schemes become efficient enough to implement. The complex math utlilized to build these schemes make them difficult for non-mathematicians to fully understand, but the potential applications are clear and could create a revolution in data security. As mathematicians and computer scientists continue improving these encryption schemes, we may soon find that the dream of wide-scale homomorphic encryption becomes a reality.

5 References:

5.1 Git Repositories:

LightPHE https://github.com/serengil/LightPHE

awesome-he https://github.com/jonaschn/awesome-he.git

5.2 Web Applications:

Morfix https://s0l0ist.github.io/seal-sandbox/

ZAMA-fhe https://huggingface.co/zama-fhe

5.3 Papers/Websites:

Wikipedia - Homomorphic Encryption https://en.wikipedia.org/wiki/Homomorphic$_e ncryption$

IEEE Digital Privacy Homomorphic Encryption Use Cases https://digitalprivacy.ieee.org/publications/topics/homomor

encryption-use-cases

IEEE Digital Privacy What is Homomorphic Encryption https://digitalprivacy.ieee.org/publications/topics/what-is-homomorphic-encryption: :text=Homomorphic

Medium Homomorphic Encryption for Beginners: A Practical Guide (Part 1) https://medium.com/privacy-preserving-natural-language-processing/homomorphic-encryption-for-beginners-a-practical-guide-part-1-b8f26d03a98a

Brilliant.org Homomorphic Encryption https://brilliant.org/wiki/homomorphic-encryption/

Gentry, C. (2009). Fully homomorphic encryption using ideal lattices. In Proceedings of the forty-first annual ACM symposium on theory of computing (pp. 169–178).

van Dijk, M., Gentry, C., Halevi, S., Vaikuntanathan, V. (2010). Fully homomorphic encryption using ideal lattices. In Proceedings of the forty-second ACM symposium on Theory of computing (pp. 169-178). ACM.