PROJECT REPORT

Partial and Fully Homomorphic Encryption Algorithms for Privacy-Preserving Machine Learning

SUPERVISOR

Prof. Dr. KK Shukla

TEAM MEMBERS

Lavish Bansal
19124019
Mathematics and Computing (IDD)

Payal Umesh Pote
19095074
Electronics Engineering (B.Tech)

Somnath Sendhil Kumar
19085089
Electrical Engineering (B.Tech)

CONTRIBUTION OF EACH MEMBER:

- **Lavish** - Studied and Implemented the Partial Homomorphic Encryption Algorithm in Python using a partially homomorphic encryption library (phe).
- **Payal** - Implemented the complete Machine Learning pipeline integrating the PHE, FHE algorithms with the ML model.
- **Somnath** - Studied and Implemented the Fully Homomorphic Encryption(FHE) Algorithm in Python using a fully homomorphic encryption library (pyFHE).

More or less, all 3 of us have approximately worked equally on each of the sections combined. We did the project step by step performing each of them one by one.

WHY HOMOMORPHIC ENCRYPTION:

Homomorphic encryption is a form of encryption that allows you to perform mathematical or logical operations on encrypted data. For example, suppose we have two numbers *m1* and *m2* and we encrypt those numbers using some public-key encryption scheme with a public key *pub* and a private key *priv*. If we have a homomorphic encryption scheme that enables addition, there will be a function add which anyone can perform on $c1 = E_{pub}(m_1)$ and $c2 = E_{pub}(m_1)$ such that the result, $add_{pub}(c1,c2)$ will decrypt to the sum of m_1 and m_2.

$$D_{priv}(add(E_{pub}(m_1), E_{pub}(m_2)) = m_1 + m_2$$

Note that the *add* function will not necessarily be a literal addition, just whichever function plays the role described above.

Normally, encryption aims to make all encrypted numbers indistinguishable from random numbers for anyone who does not have the private key required for decryption.

Previously, if an application had to perform some computation on data that was encrypted, this application would necessarily need to decrypt the data first, perform the desired computations on the clear data, and then re-encrypt the data. Homomorphic Encryption, on the other hand, simply removes the need for these decryption-encryption steps by the application, all at once.

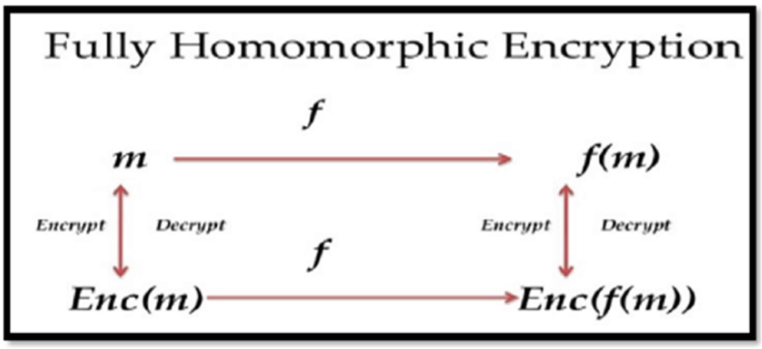
- **Partial Homomorphic Encryption(PHE):**

Partial Homomorphic Encryption schemes have been available for some time, but they support a limited number of computations, such as either only addition or only multiplication.

In 1999, Paillier Pascal invented the Paillier cryptosystem which allows us to compute both addition and multiplication operations on the encrypted data.

- **Fully Homomorphic Encryption(FHE):**

Fully Homomorphic Encryption (FHE) is still an emerging cryptographic technique that allows developers to perform computations on encrypted data.



Basically, FHE allows for arbitrary computations on encrypted data. Computing on encrypted data means that if a user has a function f and wants to obtain $f(m_1, \ldots, m_n)$ for some inputs $m_1, \ldots, m_n$, it is possible to instead compute on encryptions of these inputs, $c_1, \ldots, c_n$, obtaining a result which decrypts to $f(m_1, \ldots, m_n)$.

ADVANTAGES OF USING FHE:

- Data remains secure and private in untrusted environments, like public clouds or external parties. The data stays encrypted at all times, which minimizes the likelihood that sensitive information ever gets compromised.
- There is no need to mask or drop any features in order to preserve the privacy of data. All features may be used in an analysis, without compromising privacy.
- Fully homomorphic encryption schemes are resilient against quantum attacks.

PROJECT WORKFLOW :

```
(base) C:\Users\asus\Desktop\Fully-Homo>python Client.py
The prediction file has not been recieved from the server.

(base) C:\Users\asus\Desktop\Fully-Homo>python Server.py
Prediction file has been sent to the client.

(base) C:\Users\asus\Desktop\Fully-Homo>python Client.py
Predicted Result after FHE: 72802556.29864131
Original Predicted Result: 72802556.2986413
```

PHE: We have used Linear Regression model to predict the prices of Cars Dataset. First, the Client encrypts the

data using its public key and sends it to Server for prediction. Then the server makes the prediction on the received encrypted data and sends it back to the client by encrypting the data using the client's public key. Then the client decrypts the data using its private key.

```
"""Encrypting Client data with public key"""
def Data_serialization(data, public_key):
    encrypted_data_list = [public_key.encrypt(x) for x in data]
    encrypted_data={}
    encrypted_data['public_key'] = {'n': public_key.n}
    encrypted_data['values'] = [(str(x.ciphertext()), x.exponent) for x in encrypted_data_list]
    serialized = json.dumps(encrypted_data)
    return serialized

## Sample data point to be predicted
data = [2012, 25000, 1, 1, 1, 0]
datafile = Data_serialization(data, pub_key)
with open('Client_data.json', 'w') as file:
    json.dump(datafile, file)
```

This is how data is encrypted by the client using its public key, and then sends it to the server.

```
def getPredictions():
    ## Load the encrypted data from the file
    with open('Client_data.json', 'r') as file:
        d = json.load(file)
    data = json.loads(d)

    ##Load the weights of trained ML model
    coeff = Linear_Regressor().train_and_get_weights()
    pk = data['public_key']
    public_key = ph.PaillierPublicKey(n=int(pk['n']))
    enc_nums_rec = [ph.EncryptedNumber(public_key, int(x[0], int(x[1]))) for x in data['values']]
    results = sum([coeff[i] * enc_nums_rec[i] for i in range(len(coeff))])
    return results, public_key
```

Then the server takes the encrypted data and predicts it using the trained ML model weights, and returns the results.

```
if not os.path.exists('prediction.json'):
    print("The prediction file has not been recieved from the server.")
else:
    pub_key, priv_key = Fetch_keys()

    ##Load the result file returned by the server
    with open('prediction.json', 'r') as file:
        ans=json.load(file)
        answer_file=json.loads(ans)

    answer_key = ph.PaillierPublicKey(n=int(answer_file['pubkey']['n']))
    answer = ph.EncryptedNumber(answer_key, int(answer_file['values'][0]), int(answer_file['values'][1]))
    if (answer_key == pub_key):
        print("Predicted Result after FHE:", priv_key.decrypt(answer))

    """ Verifying the result of Encryption Algorithm with original prediction."""
    data = [2012, 25000, 1, 1, 1, 0]
    Regressor_coeff = Linear_Regressor().train_and_get_weights()
    pred = sum([data[i]*Regressor_coeff[i] for i in range(len(data))])
    print("Original Predicted Result:", pred)
```

After receiving the predictions from the server, the client decrypts the predictions using its private key and also verifies the result with the original prediction i.e. without applying the encryption algorithm.

FHE:

```
degree = ENC_SIZE

plain_modulus = ENC_SIZE*2 + 1

ciph_modulus = 8000000000000
params = BFVParameters(poly_degree=degree,
                       plain_modulus=plain_modulus,
                       ciph_modulus=ciph_modulus)
key_generator = BFVKeyGenerator(params)
public_key = key_generator.public_key
secret_key = key_generator.secret_key
relin_key = key_generator.relin_key
encoder = BatchEncoder(params)
encryptor = BFVEncryptor(params, public_key)
decryptor = BFVDecryptor(params, secret_key)
evaluator = BFVEvaluator(params)
```

Setting Parameters about the Fully Homophobic encryption using BFV. Here the Degree is 128 equal to the feature vector of the Image. Also, we generate Public, Private and Relin key which are needed for the operations.

Encrypting individual Images for processing and Predicting the Class.

```
for img in X_d:
    plain1 = encoder.encode(img)
    ciph1 = encryptor.encrypt(plain1)
```

```
for img in X_d:
    plain1 = encoder.encode(img)
    ciph1 = encryptor.encrypt(plain1)
    class_dist = []
    for i in range(N_CLASSES):
        plain2 = encoder.encode(lin_matrx[:, i])
        ciph2 = encryptor.encrypt(plain2)
        ciph_prod = evaluator.multiply(ciph1, ciph2, relin_key)
        decrypted_prod = decryptor.decrypt(ciph_prod)
        decoded_prod = encoder.decode(decrypted_prod)
        class_dist.append(sum(decoded_prod))
    pred.append(np.argmax(class_dist))
```

Running the computation separately for each class as we are only limited to dot product of two messages hence accumulating the class distribution.

SOURCE CODE LINK:

https://github.com/hex-plex/PnFHE_Privacy_Preserving_ML