UTS UNIVERSITY OF TECHNOLOGY SYDNEY

Task 02

HOMOMORPHIC ENCRYPTION IN NETWORK TRAFFIC for NEURAL NETWORKS

Present by Group 6

OVERVIEW

1. Introduction (abstract + research quesion)
2. Background and context
3. Methods
4. Benchmark data
5. Evaluation
6. Reference

HOMOMORPHIC ENCRYPTION

*Homomorphic encryption allows **computations** to be performed on encrypted data **without decryption***

First Implemented

Craig Gentry (2009)
Stanford University
Palto Alto, USA

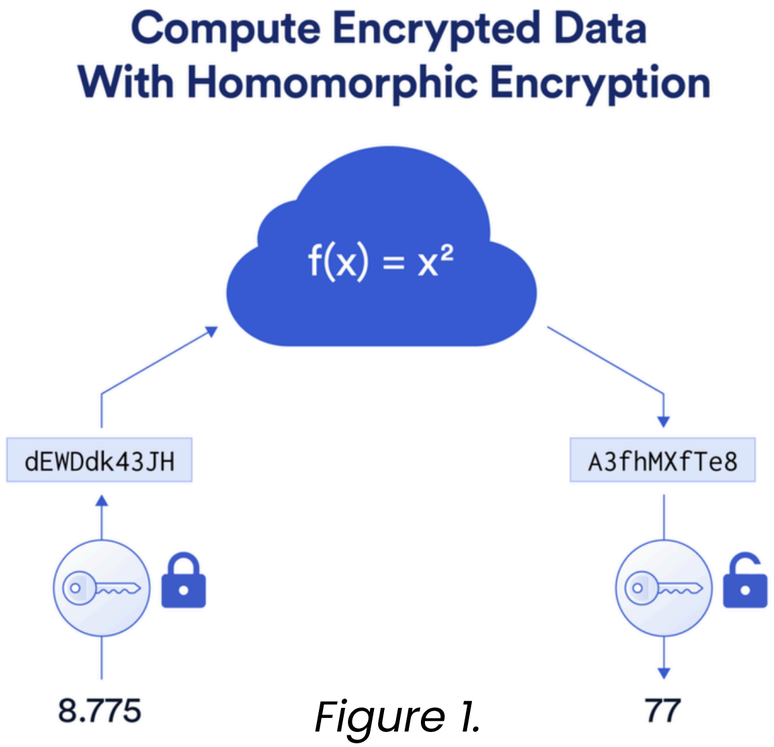


Figure 1.

Importance of data privacy and security in machine learning:
In the modern age of machine learning and AI driven models, data privacy is a huge challenge in the use of up and coming technologies.

Sensitive data such as healthcare information, financial data or personal identifying data poses a huge risk to individuals and also **limits companies ability** to operate on data and use it in meaningful ways due to the risk of data breaches, data misuse or other malicious usage.

RESEARCH QUESTION

How can we use Paillier Crytpography for Partial Homomorphic Encryption systems for neural networks?

Paillier Cryptography

- Exploring the usage of Paillier Cryptography to encrypt data using asymmetric public-private keys

Pascal Pallier (1999)

Partially Homomorphic Systems

- Showcase the idea of a partial homomorphic system that doesnt require human intervention

Neural Networks

- Exploring the use of neural network models to perform encryption and decryption tasks

Objectives

- Assessing the suitability of Paillier Crytpographic encryption in regards to performance, privacy and security
- Implementing a proof of concept model that demonstrates the benefits of homomorphic encryption systems using neural networks.
- Demonstrate the feasibility of adaptation of homomorphic encryption in other models or highlight potential in other domain areas.

PROPOSED SOLUTION

Implement a public key encryption scheme using Python that supports homomorphic encryption. This will be further integrated into neural networks and will be usable without human interaction.

SOLUTION OVERVIEW

- Pallier Encryption Implementation
- Public-Private key generation
- Partial homomorphic encryption
- Neural Network Model implementation

EXPECTED OUTCOMES

- High model accuracy on selected datasets
- Better Computational performance
- Increased security/privacy through data obscuration

POTENTIAL USE CASES

MULTI-PARTY COLLABORATION

- Homomorphic encryption could allow the obscuration of individual input data in cases of sensitive data that requires collaboration of third parties whilst offering protection to individuals.

SECURE COMPUTATIONAL OUTSOURCING

- Using homomorphic encryption neural network systems could allow autonomous outsourcing of computation power that completely preservers individual privacy and security. This could be used in areas such as outsourcing data analytics to a 3rd party company without revealing sensitive data.

MACHINE LEARNING PRIVACY PRESERVATION

- Homomorphic encryption could solve a significant issue in training machine learning models in that user data is often used unknowingly as training data for large scale machine learning. Homomorphic encryption could allow the training of machine learning models without revealing private data.

SECURE DATA ANALYTICS IN E-COMMERCE/RETAIL

- Homomorphic encryption could also be used to gain valuable insight into customer behaviours and trends without requiring invasive data tracking and information on individual customers

BACKGROUND AND CONTEXT

Encryption techniques are methods used to keep information secret. Traditional encryption techniques refer to algorithms used to generate secret information before the advent of computers. However, these algorithms became easily breakable with the advancement of computation. This moved us in a new era of computing, leading to the development of new encryption techniques known as modern encryption methods.

Compare and contrast the **benefits** and **changes** from traditional methods to modern encryption methods, as well as **homomorphic encryption**.

Traditional Methods

- Symmetric Encryption
- Concepts/Methods before computer era
- Requires decryption before data can be used
- Involves simple mathematical operations, faster and more efficient

Modern Methods; Homomorphic Enc

- Asymmetric Encryption
- Concepts/Methods before + after computer era
- Computations on encrypted data
- Supports various types of computation

Timeline

1943-1945

First Computer Created

1976

First public key idea proposed

1999

Paillier Crypto developed (semi-homomorphic)

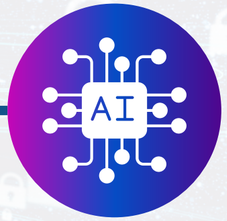
2000

AES(Advanced encryption standard) Developed

2009

Full Homomorphic Encryption is invented

CURRENT



Moving to AI era

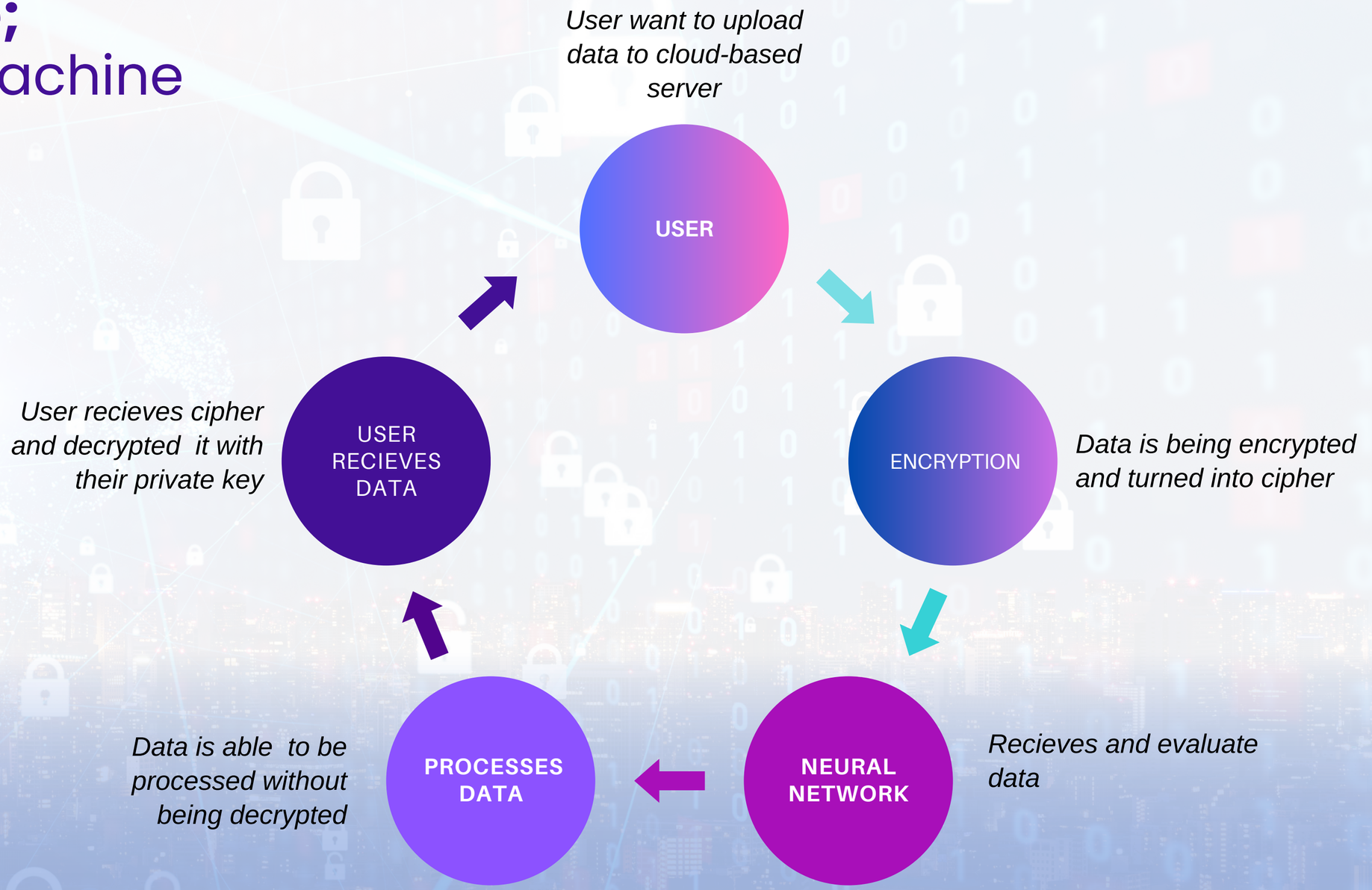
Pre-Computer Era

Information Age

AI Era

Use Case Example; Homomorphic in Machine Learning

"Secure computation outsourcing to untrusted servers or third-party service providers without exposing the underlying data."



Encryption

Encoding/decoding to suitable form

- Exponent extraction by determining data type (int or float)
- Finding the correct integer representation by finding the necessary precision from the exponenet
- Adding the integer representation with modulus from public key in case of negative scalars

Encryption

- Generating pseudo-random prime values of specific length
- From there you can generate public key and private key of a specific length to encrypt your values with homomorphic properties (Paillier, 1999)

Method

Encryption

Encrypting data with homomorphic properties using Paillier Scheme

Machine learning

Preprocessing the data, encrypting it, then passing it onto machine learning algorithms

Paillier Encryption Analyse

```python
import numpy as np
import random
import math
import libnum

key_length = 256

class PaillierKeyGen:
    """ Object to Generate Public/Private Key pair """
    def __init__(self):
        length = key_length/2
        p = prime_generator(length)
        q = p
        while p == q:
           q = prime_generator(length)
        self.n = p*q
        self.glambda = math.lcm(p-1,q-1)
        self.g = random.choice(multiplicative(self.n))
        l = (pow(self.g, self.glambda, self.n*self.n)-1)//self.n
        self.gMu = libnum.invmod(l,self.n)

    def publickey(self):
        """ Returns PublicKeyO object, can be used to encrypt data """
        return PublicKey(self.n, self.g)

    def privatekey(self, public_key):
        """ Returns PrivateKey object, can be used to encrypt data """
        return PrivateKey(self.glambda,self.gMu, self.n, public_key)
```

Initializes key generation with specified key length.

Encrypts data ensuring that only someone with the private key can decrypt it.

Decrypts data that was encrypted with the associated public key.

Paillier Scheme algorithm (Paillier 1999)

```python
class PublicKey:
    def __init__(self, n, g):
        self.n = n
        self.g = g
        self.n_squared = n*n
        self.max_int = n//3 -1
        self.r = random.randint(0, self.n_squared)

    def raw_encrypt(self, m, r):
          k1 = pow(self.g, m, self.n_squared)
          k2 = pow(r, self.n, self.n_squared)
          cipher = (k1* k2) % (self.n_squared)
          return cipher

    def encrypt(self, encoding, precision=None):
      if isinstance(encoding, EncodedNumber):
          cipher = self.raw_encrypt(encoding.encoding, self.r)
      else:
          encoding = EncodedNumber.encode(self, encoding, precision)
          cipher = self.raw_encrypt(encoding.encoding, self.r)
      return EncryptedNumber(self, cipher, encoding.exponent)

    def raw_add(self, cipher1, cipher2):
        return (cipher1 * cipher2) % (self.n_squared)

    def raw_mul(self, cipher, plaintext):
        if not isinstance(cipher, int):
            raise TypeError('Expected ciphertext to be int, not %s' %
                type(cipher))

        if plaintext < 0 or plaintext >= self.n:
            raise ValueError('Scalar out of bounds: %i' % plaintext)

        if self.n - self.max_int <= plaintext:
            # Very large plaintext, play a sneaky trick using inverses
            neg_c = libnum.invmod(cipher, self.n_squared)
            neg_scalar = self.n - plaintext
            return powmod(neg_c, neg_scalar, self.n_squared)
        else:
            return powmod(cipher, plaintext, self.n_squared)
```

Paillier Scheme algorithm (Paillier 1999)

INITIALIZE

Establishes public key parameters for subsequent encryption processes.

↓

RAW ENCRYPT

Directly encrypts numbers, serving as the core encryption mechanism.

↓

ENCRYPT

Processes and encrypts data, making it ready for secure handling.

↓

ADDING ENCRYPTED VALUES

Combines two encrypted values, enabling secure addition operations.

↓

MULTIPLY FUNCTION

Performs multiplication between an encrypted and a plain number, extending encrypted calculations.

```python
class PrivateKey:
    def __init__(self,glambda, gMu, n, public_key):
        self.glambda = glambda
        self.gMu= gMu
        self.public_key = public_key
        self.n = n

    def repr(self):
        pub_repr = repr(self.public_key)
        return f"[{self.__class.__name}{pub_repr}]"

    def raw_decrypt(self, cipher):
        l = (pow(cipher, self.glambda, self.n*self.n)-1) // self.n
        mess= (l * self.gMu) % self.n
        return mess

    def decrypt(self, cipher):
        """Return the decrypted & decoded plaintext of *encrypted_number*. """

        encoded = self.decrypt_encoded(cipher)
        return encoded.decode()

    def decrypt_encoded(self, cipher, Encoding=None):
        if not isinstance(cipher, EncryptedNumber):
            raise TypeError('Expected encrypted_number to be an EncryptedNumber'
                            ' not: %s' % type(cipher))

        if self.public_key != cipher.public_key:
            raise ValueError('encrypted_number was encrypted against a '
                             'different key!')

        if Encoding is None:
            Encoding = EncodedNumber

        encoded = self.raw_decrypt(cipher.ciphertext())
        return Encoding(self.public_key, encoded,
                        cipher.exponent)
```

Paillier Scheme algorithm (Paillier 1999)

INITIALIZATION FOR PRIVATE KEY

Configures the private key using the core parameters and the corresponding public key.

↓

PUBLIC KEY REPRESENTATION

Offers a string representation of the associated public key for reference.

↓

RAW DECRYPTION

Executes the primary decryption process on an encrypted numerical value.

↓

DECRYPTION

Decodes and decrypts an encrypted number to its original form.

↓

DECRYPTION WITH ENCODING

Decodes and decrypts an encrypted number with encoding to its original value, ensuring correct data interpretation.

```python
[5]  class EncryptedNumber(object):
         """Represents the Paillier encryption of a float or int."""

         def __init__(self, public_key, ciphertext, exponent=0):
             self.public_key = public_key
             self.__ciphertext = ciphertext
             self.exponent = exponent
             if isinstance(self.ciphertext, EncryptedNumber):
                 raise TypeError('ciphertext should be an integer')
             if not isinstance(self.public_key, PublicKey):
                 raise TypeError('public_key should be a PublicKey')

         def __add__(self, other):
             """Add an int, float, `EncryptedNumber` or `EncodedNumber`."""
             if isinstance(other, EncryptedNumber):
                 return self._add_encrypted(other)
             elif isinstance(other, EncodedNumber):
                 return self._add_encoded(other)
             else:
                 return self._add_scalar(other)

         def __radd__(self, other):
             """Called when Python evaluates `34 + <EncryptedNumber>`
             Required for builtin `sum` to work.
             """
             return self.__add__(other)

         def __mul__(self, other):
             if isinstance(other, EncryptedNumber):
               raise ValueError("Not possible to multiply ciphertexts")
             if isinstance(other, EncodedNumber):
                 encoding = other
             else:
                 encoding = EncodedNumber.encode(self.public_key, other)
             product = self.public_key.raw_mul(self.ciphertext(), encoding.encoding)
             exponent = self.exponent + encoding.exponent
```

INITIALIZE

Creates an encrypted number using the public key and the ciphertext, with an adjustable exponent for precision.

ADDITION (__ADD__)

Adds an encrypted number with another encrypted number, an encoded number, or a plain scalar

REVERSE ADDITION (__RADD__)

Ensures compatibility with Python's sum function for adding a list of encrypted numbers.

MULTIPLICATION (__MUL__)

Multiplies an encrypted number by a plain scalar, but disallows multiplying two encrypted numbers.

```python
class EncodedNumber(object):
    BASE = 16
    """Base to use when exponentiating. Larger `BASE` means
    that :attr:`exponent` leaks less information. If you vary this,
    you'll have to manually inform anyone decoding your numbers.
    """
    LOG2_BASE = math.log(BASE, 2)
    FLOAT_MANTISSA_BITS = sys.float_info.mant_dig

    def __init__(self, public_key, encoding, exponent):
        self.public_key = public_key
        self.encoding = encoding
        self.exponent = exponent

    @classmethod
    def encode(cls, public_key, scalar, precision=None, max_exponent=None):
        """Return an encoding of an int or float."""

        # Calculate the maximum exponent for desired precision
        if precision is None:
            if isinstance(scalar, int):
                prec_exponent = 0
            elif isinstance(scalar, float):
                # Encode with *at least* as much precision as the python float
                bin_flt_exponent = math.frexp(scalar)[1] #Base 2 exponent

                # The least significant bit has value 2 ** bin_lsb_exponent
                bin_lsb_exponent = bin_flt_exponent - cls.FLOAT_MANTISSA_BITS

                # What's the corresponding base BASE exponent? Round that down.
                prec_exponent = math.floor(bin_lsb_exponent / cls.LOG2_BASE)
            else:
                raise TypeError("Don't know the precision of type %s."
                                % type(scalar))
        else:
            prec_exponent = math.floor(math.log(precision, cls.BASE))

        # Exponents are negative for numbers < 1.
        # If we're going to store numbers with a more negative exponent than dema
        if max_exponent is None:
            exponent = prec_exponent
```

(data61, 2023)

INITIALIZATION

Constructs an encoded number with a given public key and encoding parameters, allowing for precision adjustments.

CLASS METHOD ENCODE

Transforms a scalar into an encoded number, adjusting the exponent for desired precision levels.

BASE CALCULATION

Determines the exponent base on the precision of the scalar, ensuring minimal loss of information during encryption.

```
    # Exponents are negative for numbers < 1.
    # If we're going to store numbers with a more negative exponent than der
    if max_exponent is None:
        exponent = prec_exponent
    else:
        exponent = min(max_exponent, prec_exponent)

    # Use rationals instead of floats to avoid overflow.
    int_rep = round(fractions.Fraction(scalar)
                    * fractions.Fraction(cls.BASE) ** -exponent)

    if abs(int_rep) > public_key.max_int:
        raise ValueError('Integer needs to be within +/- %d but got %d'
                         % (public_key.max_int, int_rep))

    # Wrap negative numbers by adding n
    return cls(public_key, int_rep % public_key.n, exponent)
```

(data61, 2023)

```
def decode(self):
    """Decode plaintext and return the result. """
    if self.encoding >= self.public_key.n:
        # Should be mod n
        raise ValueError('Attempted to decode corrupted number')
    elif self.encoding <= self.public_key.max_int:
        # Positive
        mantissa = self.encoding
    elif self.encoding >= self.public_key.n - self.public_key.max_int:
        # Negative
        mantissa = self.encoding - self.public_key.n
    else:
        raise OverflowError('Overflow detected in decrypted number')

    if self.exponent >= 0:
        # Integer multiplication. This is exact.
        return mantissa * self.BASE ** self.exponent
    else:
        # BASE ** -e is an integer, so below is a division of ints.
        # Not coercing mantissa to float prevents some overflows.
        try:
            return mantissa / self.BASE ** -self.exponent
        except OverflowError as e:
            raise OverflowError(
                'decoded result too large for a float') from e
```

(data61, 2023)

```python
class Dataset(object):
    def __init__(self, segments, labels, one_hot = False, reshape = True):
          """Construct a Dataset
          one_hot arg is used only if fake_data is True. 'dtype' can be either unit9 or float32
          """
          '''
          dtype = dtypes.as_dtype(dtype).base_dtype
          if dtype not in (dtypes.uint8, dtypes.float32):
              raise TypeError('Invalid')
          '''
          self._num_examples = len(segments)
          self._segments = segments
          self._labels = labels
          self._epochs_completed = 0
          self._index_in_epoch = 0

    @property
    def segments(self):
        return self._segments

    @property
    def labels(self):
        return self._labels

    @property
    def num_examples(self):
        return self._num_examples

    @property
    def epochs_completed(self):
        return self._epochs_completed

    def next_batch(self, batch_size):
        """Return the next batch-size examples from this dataset"""

        start = self._index_in_epoch
        self._index_in_epoch += batch_size
        if self._index_in_epoch > self._num_examples:
            self._epochs_completed +=1

            perm = np.arange(self._num_examples)
```

Data Preprocessing

```
dir_path = "/drive/MyDrive"

filename1 = "/content/drive/MyDrive/datasets/kdd/kddcup.data_10_percent_corrected"
filename2 = "/content/drive/MyDrive/datasets/kdd/corrected"
dataset1 = read_data(filename1)
dataset2 = read_data(filename2)
nomial(dataset1, dataset2)

dataset1['label'] = initlabel(dataset1)
dataset2['label'] = initlabel(dataset2)

num_features = ["duration","protocol_type","service","flag", "src_bytes", "dst_bytes","land","wrong_fragment","urgent","hot","num_failed_logins",
    "logged_in","num_compromised","root_shell","su_attempted","num_root",
    "num_file_creations","num_shells","num_access_files","num_outbound_cmds",
    "is_host_login","is_guest_login","count","srv_count","serror_rate",
    "srv_serror_rate","rerror_rate","srv_rerror_rate","same_srv_rate",
    "diff_srv_rate","srv_diff_host_rate","dst_host_count","dst_host_srv_count",
    "dst_host_same_srv_rate","dst_host_diff_srv_rate","dst_host_same_srv_port_rate",
    "dst_host_srv_diff_host_rate","dst_host_serror_rate","dst_host_srv_serror_rate",
    "dst_host_rerror_rate","dst_host_srv_rerror_rate"
    ]
dataset1[num_features] = dataset1[num_features].astype(float)
dataset1[num_features] = MinMaxScaler().fit_transform(dataset1[num_features].values)
dataset2[num_features] = dataset2[num_features].astype(float)
dataset2[num_features] = MinMaxScaler().fit_transform(dataset2[num_features].values)

print(dataset1.describe())

print(dataset1['label'].value_counts())

labels1 = dataset1['label'].copy()
print(labels1.unique())

labels1[labels1 == 'normal.'] = 0
labels1[labels1 == 'dos'] = 1
labels1[labels1 == 'u2r'] = 2
labels1[labels1 == 'r2l'] = 3
labels1[labels1 == 'probe'] = 4
dataset1['label'] = labels1
```

```
[56] np.random.seed(42)

     indices = np.arange(len(train._segments))
     np.random.shuffle(indices)
     shuffled_features = np.array(train._segments)[indices]
     shuffled_labels = np.array(train._labels)[indices]
     test._segments = np.array(test._segments)

     shuffled_features = shuffled_features.reshape(shuffled_features.shape[0], 41)
     shuffled_features = shuffled_features[:10000]
     test._segments = test._segments.reshape(test._segments.shape[0], 41)
```

```
     key_gen = PaillierKeyGen()
     public_key = key_gen.publickey()
     private_key = key_gen.privatekey(public_key)
     encrypted_train = encrypt_vector(shuffled_features.tolist(), 0, len(shuffled_features[:10000]), public_key)
     encrypted_test = encrypt_vector(test._segments[:1000].tolist(), 0, len(test._segments[:1000]), public_key)
```

```
[58] train._labels = shuffled_labels[:10000].tolist()
     test._labels = test._labels[:1000]
```

Neural Network

```python
class NeuralNetwork:
    def __init__(self, input_size, hidden_size, output_size, private_key):
        self.input_size = input_size
        self.hidden_size = hidden_size
        self.output_size = output_size
        self.private_key = private_key

        # Initialize weights and biases
        self.W1 = [[random.uniform(-1, 1) for _ in range(hidden_size)] for _ in range(input_size)]
        self.b1 = [0 for _ in range(hidden_size)]
        self.W2 = [[random.uniform(-1, 1) for _ in range(output_size)] for _ in range(hidden_size)]
        self.b2 = [0 for _ in range(output_size)]

    def sigmoid(self, x):
      return [1 / (1 + math.exp(-i)) for i in x]

    def forward(self, X):
    # Forward pass through the network without activation functions
        self.z1 = [sum((X[i] * self.W1[i][j] for i in range(self.input_size))) + self.b1[j] for j in range(self.hidden_size)]
        self.a1 = self.z1  # No activation function
        self.z2 = [sum((self.a1[i] * self.W2[i][j] for i in range(self.hidden_size))) + self.b2[j] for j in range(self.output_size)]
        self.a2 = decrypt_vector(private_key, self.z2)
        self.a2 = self.sigmoid(self.a2)
        return self.a2
```

```python
def backward(self, y, output, learning_rate):
    # Backpropagation to update weights and biases
    self.loss = mean_absolute_error(output, y)

    # Compute delta_z2
    delta_z2 = [output[i] - y[i] for i in range(self.output_size)]
    self.a1 = decrypt_vector(private_key, self.a1)
    # Update weights and biases for output layer (W2 and b2)
    delta_W2 = [[self.a1[j] * delta_z2[i] for i in range(self.output_size)] for j in range(self.hidden_size)]
    delta_b2 = delta_z2

    delta_z1 = [sum(delta_z2[i] * self.W2[j][i] for i in range(self.output_size)) for j in range(self.hidden_size)]

    # Update weights and biases for hidden layer (W1 and b1) excluding inputs
    delta_W1 = [[delta_z1[j] for _ in range(self.input_size)] for j in range(self.hidden_size)]
    delta_b1 = delta_z1



    # Update weights and biases using the computed deltas and learning rate
    self.W1 = [[self.W1[j][k] - learning_rate * delta_W1[j][k] for k in range(self.input_size)] for j in range(self.hidden_size)]
    self.b1 = [self.b1[j] - learning_rate * delta_b1[j] for j in range(self.hidden_size)]
    self.W2 = [[self.W2[j][k] - learning_rate * delta_W2[j][k] for k in range(self.output_size)] for j in range(self.hidden_size)]
    self.b2 = [self.b2[j] - learning_rate * delta_b2[j] for j in range(self.output_size)]
```

```
[62] def main(model = [], private_key = None):
        # Define the neural network architecture
        output_size = len(train._labels[0])
        learning_rate = 0.01

        if private_key != None :
          input_size = len(encrypted_train[0])
          hidden_size = len(encrypted_train[0])
          X_train = encrypted_train
          X_test = encrypted_test
        else:
          input_size = len(shuffled_features[0])
          hidden_size = len(shuffled_features[0])
          X_train = shuffled_features
          X_test = test._segments[:1000]
        y_train = train._labels
        y_test = test._labels

        # Create an instance of the neural network
        nn = NeuralNetwork(input_size, hidden_size, output_size, private_key)
        if len(model) != 0:
            nn.W1 = model[0]
            nn.W2 = model[1]
            nn.b1 = model[2]
            nn.b2 = model[3]

        # Training loop
        epochs = 1000
        lowest_loss = float('inf')
        print("Beginning training")
        for epoch in range(epochs):
            total_loss = 0
            total_acc = 0
            correct_train = 0
            correct_test = 0
            with tqdm(total=len(X_train), desc=f"Epoch {epoch}", unit=" samples") as pbar:
                for i in range(len(X_train)):
                    # Forward pass
                    output = nn.forward(X_train[i])
                    # Backward pass
                    nn.backward(y_train[i], output, learning_rate)
                    # Accumulate loss
```

```
with tqdm(total=len(X_test), desc=f"Epoch {epoch}", unit=" samples") as tbar:
    for i in range(len(X_test)):
        output = nn.forward(X_test[i])
        result = np.argmax(output)
        if result == np.argmax(y_test[i]):
            correct_test += 1
        tbar.update(1)  # Update progress bar
        tbar.set_description(f"Epoch {epoch} - Accuracy: {correct_test/(i+1):.4f}")
    test_acc = (correct_test / len(X_test))*100
    print(f"Epoch {epoch}: Test Accuracy = {test_acc}%")
```

Data Benchmark

KDD-Cup 1999

Contains standard set of data that requires auditing, containing wide variety of intrusions simulated in a military network environment (Huang, n.d.).

Why we choose it?

- Large Volume of Data
- High Feature Diversity
- Extensive Data Variety

Encryption speed (256 bit key)	Input size	Hidden Size	Output size	Learning rate	Training sample size	Testing sample size
~42 encryptions per second	41	41	5	0.01	10000	1000

```
Encrypted : 9960 numbers
Encrypted : 9980 numbers
Time : 235.56600904464722
```

Normal	u2r	dos	r2l	probe
Normal packets	User to Root attacks (Privilege escalation)	Denial of Service attacks	Remote to local attacks	Probing or Scanning the network traffic (e.g. Port scans)

Speed	Encrypted Data	Non-Encrypted data
Samples/s	2-3	150-200

```
main(model = [], private_key = private_key)

Beginning training
Epoch 0 - Accuracy: 0.5000:   0%|          | 26/10000 [00:10<1:00:12,  2.76 samples/s]
```

```
Beginning training
Epoch 0 - Accuracy: 0.9788 - Loss: 0.3342: 100%|██████████| 10000/10000 [00:55<00:00, 178.81 samples/s]
Epoch 0: Train Accuracy = 97.88%
Epoch 0: Average Loss = 0.0156
```

With encrypted data

```
Beginning training
Epoch 0 - Accuracy: 0.9761 - Loss: 0.3110: 100%|          | 10000/10000 [1:04:44<00:00,  2.57 samples/s]
Epoch 0: Train Accuracy = 97.61%
Epoch 0: Average Loss = 0.0193
Epoch 0: 100%|          | 1000/1000 [06:01<00:00,  2.77 samples/s]
Epoch 0: Test Accuracy = 88.6%
Epoch 1 - Accuracy: 0.9851 - Loss: 0.2124: 100%|          | 10000/10000 [1:09:40<00:00,  2.39 samples/s]
Epoch 1: Train Accuracy = 98.50999999999999%
Epoch 1: Average Loss = 0.0120
Epoch 1: 100%|          | 1000/1000 [06:16<00:00,  2.66 samples/s]
Epoch 1: Test Accuracy = 88.7%
Epoch 2 - Accuracy: 0.9861 - Loss: 0.1756: 100%|          | 10000/10000 [1:04:39<00:00,  2.58 samples/s]
Epoch 2: Train Accuracy = 98.61%
Epoch 2: Average Loss = 0.0108
Epoch 2: 100%|          | 1000/1000 [06:08<00:00,  2.72 samples/s]
Epoch 2: Test Accuracy = 88.6%
Epoch 3 - Accuracy: 0.9869 - Loss: 0.1608: 100%|          | 10000/10000 [1:05:58<00:00,  2.53 samples/s]
Epoch 3: Train Accuracy = 98.69%
Epoch 3: Average Loss = 0.0101
Epoch 3: 100%|          | 1000/1000 [06:03<00:00,  2.75 samples/s]
Epoch 3: Test Accuracy = 88.5%
Epoch 4 - Accuracy: 0.9878 - Loss: 0.1491: 100%|          | 10000/10000 [1:07:31<00:00,  2.47 samples/s]
Epoch 4: Train Accuracy = 98.78%
Epoch 4: Average Loss = 0.0096
```

With encrypted data

Epoch	Testing Accuracy	Training Accuracy	Training Loss	Time per epoch
0	88.6%	97.61%	0.0193	1:04:44
1	88.7%	98.51%	0.0120	1:09:40
2	88.6%	98.61%	0.0108	1:04:39
3	88.5%	98.69%	0.0101	1:05:58
4	88.5%	98.78%	0.0096	1:07:31

With non-encrypted data

```
Beginning training
Epoch 0 - Accuracy: 0.9781 - Loss: 0.2057: 100%|██████████| 10000/10000 [00:55<00:00, 181.39 samples/s]
Epoch 0: Train Accuracy = 97.81%
Epoch 0: Average Loss = 0.0172
Epoch 0 - Accuracy: 0.8800: 100%|██████████| 1000/1000 [00:03<00:00, 262.46 samples/s]
Epoch 0: Test Accuracy = 88.0%
Epoch 1 - Accuracy: 0.9860 - Loss: 0.2028: 100%|██████████| 10000/10000 [00:58<00:00, 171.47 samples/s]
Epoch 1: Train Accuracy = 98.6%
Epoch 1: Average Loss = 0.0113
Epoch 1 - Accuracy: 0.8800: 100%|██████████| 1000/1000 [00:03<00:00, 294.47 samples/s]
Epoch 1: Test Accuracy = 88.0%
Epoch 2 - Accuracy: 0.9870 - Loss: 0.1923: 100%|██████████| 10000/10000 [00:55<00:00, 178.85 samples/s]
Epoch 2: Train Accuracy = 98.7%
Epoch 2: Average Loss = 0.0100
Epoch 2 - Accuracy: 0.8790: 100%|██████████| 1000/1000 [00:03<00:00, 263.04 samples/s]
Epoch 2: Test Accuracy = 87.9%
Epoch 3 - Accuracy: 0.9883 - Loss: 0.1787: 100%|██████████| 10000/10000 [01:15<00:00, 131.70 samples/s]
Epoch 3: Train Accuracy = 98.83%
Epoch 3: Average Loss = 0.0093
Epoch 3 - Accuracy: 0.8790: 100%|██████████| 1000/1000 [00:03<00:00, 258.61 samples/s]
Epoch 3: Test Accuracy = 87.9%
Epoch 4 - Accuracy: 0.9891 - Loss: 0.1664: 100%|██████████| 10000/10000 [00:56<00:00, 177.61 samples/s]
Epoch 4: Train Accuracy = 98.91%
Epoch 4: Average Loss = 0.0088
Epoch 4 - Accuracy: 0.8800: 100%|██████████| 1000/1000 [00:03<00:00, 254.71 samples/s]
Epoch 4: Test Accuracy = 88.0%
```

With non-encrypted data

Epoch	Testing Accuracy	Training Accuracy	Training Loss	Time per epoch
0	88.0%	97.81%	0.0172	00:00:55
1	88.0%	98.60%	0.0113	00:00:58
2	87.9%	98.70%	0.0100	00:00:55
3	87.9%	98.83%	0.0093	00:01:15
4	88.0%	98.91%	0.0088	00:00:56

Evaluation

Exploring the integration of the Paillier cryptosystem with neural networks to enhance data privacy in AI, highlighting our achievements, challenges, improvements, and future directions in privacy-preserving machine learning.

Key Achievements & Challenges

Improvements

Future Directions

KEY ACHIVEMENTS

1. Enhanced Data Privacy
2. Homomorphic Encryption Capabilities
3. Compatibility with Federated Learning

CHALLENGES

1. Performance Limitations
2. Key Size Requirements
3. Activation Function Constraints
4. Lack of Library Support

Improvements

Optimization of Computational Processes

We plan to optimize cryptographic operations by refining encryption and decryption routines to reduce computational load and enhance processing speed, potentially bypassing Python's Global Interpreter Lock.

Compatibility with Machine Learning Frameworks

The development of adapters or middleware will enable seamless integration of encrypted operations with popular machine learning frameworks like TensorFlow and PyTorch, allowing developers to use familiar tools and libraries.

Future Directions

Research on Advanced Homomorphic Encryption

The research aims to develop next-generation homomorphic encryption schemes that enable complex functions like sigmoid to be executed on encrypted data, fully supporting deep learning architectures.

Federated Learning Enhancements

We plan to enhance data security in multi-party computations and develop efficient data aggregation methods while maintaining encryption to leverage our federated learning capabilities further.

Thank You

For Your Attention

References

data61. (2023, October 23). data61/python-paillier. GitHub. https://github.com/data61/python-paillier

Huang, S. (n.d.). KDD Cup 1999 Data. Www.kaggle.com. https://www.kaggle.com/datasets/galaxyh/kdd-cup-1999-data

Paillier, P. 1999, 'Public-Key Cryptosystems Based on Composite Degree Residuosity Classes', LNCS, vol. 1592, pp. 223–38, viewed 9 March 2024, <https://link.springer.com/content/pdf/10.1007/3-540-48910-X_16.pdf>.

Chainlink. (2023, December 21). [Diagram of Homomorphic Encryption]. Chainlink. https://chain.link/education-hub/homomorphic-encryption

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