Homomorphic Encryption

Cryptography can keep secrets safe while sharing important information. Imagine you want to send a picture of a new traffic sign to your computer friend who can recognize traffic signs. You can put the picture inside a special envelope only your friend can open. It is like putting a secret code lock on the envelope. This way, even if someone tries to peek at the picture while it travels through the internet, they will not understand it because it is all coded up.

Now, let us talk about Homomorphic Encryption (HE) like in the figure. Your friend may want to learn something new from all the pictures it receives but still wants to keep everything secret. HE is like teaching the computer to do calculations on the coded-up pictures without actually decoding them.

1. Introduction to Cryptography

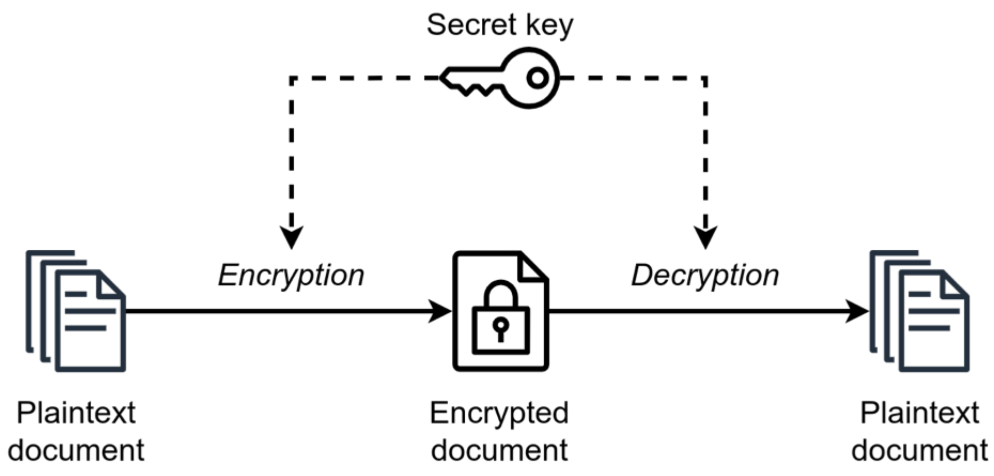
Cryptography is the practice of creating secure communications by transforming plaintext into ciphertext using mathematical techniques. These techniques are used to protect the confidentiality, integrity, and authenticity of information.

1.1 Basics of Encryption and Decryption

Encryption is the process of transforming plaintext into ciphertext. This is achieved using a secret key or a public key. The two main types of encryption are symmetric encryption and asymmetric encryption.

1.1.1 Symmetric Cryptography

Symmetric encryption is a type of encryption where a single secret key is used for both encryption and decryption of messages. This key must be shared between the sender and the recipient to enable secure communication. The encryption process can be represented by the mathematical expression $(c = E_k(p))$, where (k) represents the secret key and (p) is the plaintext message to be encrypted. The encryption function (E_k) is typically a permutation or substitution cipher, which operates on blocks of fixed length. The decryption function, on the other hand, is expressed as $(p = D_k(c))$, where (k) is the same secret key used for encryption and (c) is the ciphertext obtained after encryption.



The security of symmetric encryption algorithms depends on

- the key length
- the quality of the key generation process
- the mathematical operations used for encryption and decryption.

The key length determines the number of possible keys that can be used for encryption and decryption, and a longer key length increases the security of the algorithm. The quality of the key generation process is crucial, as a weak key can be easily guessed or brute-forced, leading to a compromise of the encrypted message.

Some of the widely used symmetric encryption algorithms are

- Advanced Encryption Standard (AES)
- Data Encryption Standard (DES)
- Blowfish.

These algorithms use a series of mathematical operations to scramble the plaintext into an unreadable form, making it difficult for unauthorized parties to access the original message.

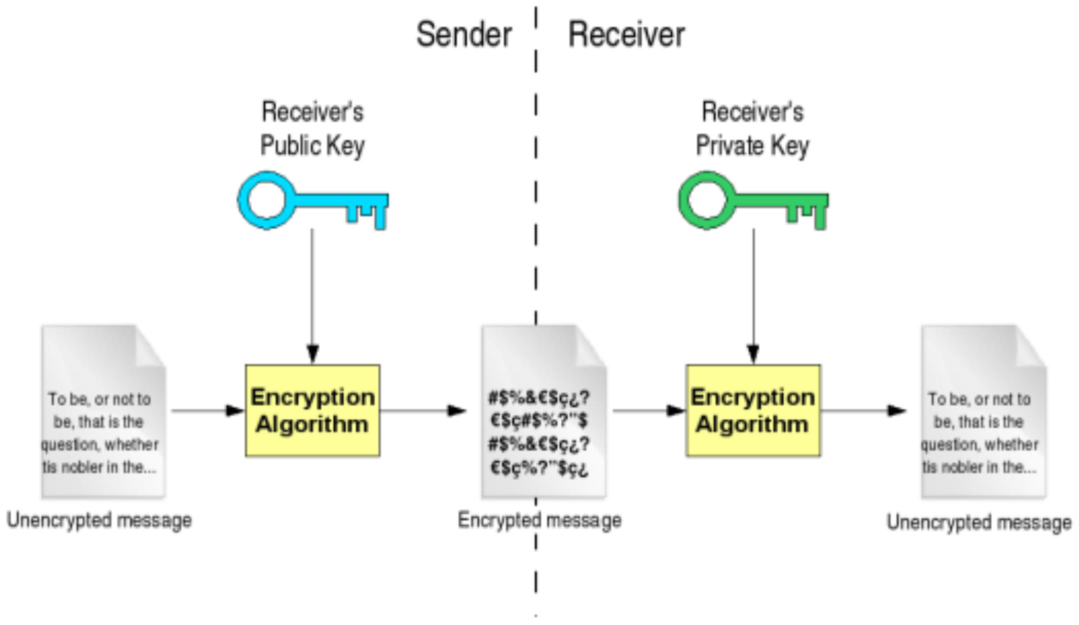
AES is considered the most secure symmetric encryption algorithm and is widely used in various applications, including online banking and government communications. AES operates on fixed-length blocks of 128 bits and uses a key of 128, 192, or 256 bits. The AES encryption and decryption functions are based on substitution and permutation operations, as well as a key expansion algorithm.

DES, on the other hand, has been deprecated due to its vulnerabilities and is no longer considered a secure option for encryption. DES uses a 56-bit key and operates on 64-bit blocks. The algorithm consists of 16 rounds of substitution and permutation operations, and it is vulnerable to brute-force attacks due to its short key length.

Blowfish is a widely-used symmetric encryption algorithm that is known for its fast processing speed and strong security. Blowfish operates on 64-bit blocks and uses a key of variable length, up to a maximum of 448 bits. The algorithm consists of 16 rounds of substitution and permutation operations, and it is resistant to brute-force attacks due to its key length and the complex nature of the encryption and decryption functions.

1.1.2 Asymmetric Cryptography (Public-key cryptography)

Asymmetric encryption is a type of encryption that uses two different keys for encryption and decryption. These keys are known as the public key and the private key. The public key is used for encryption and is openly shared, while the private key is kept secret and used for decryption. The encryption process can be represented by the mathematical expression $(c = E_{pk}(p))$, where (pk) is the public key and (p) is the plaintext message to be encrypted. The decryption function, on the other hand, is expressed as $(p = D_{sk}(c))$, where (sk) is the private key used for decryption.


Sender
Receiver
Receiver's Public Key
Receiver's Private Key
To be, or not to be, that is the question, whether tis nobler in the...
Encryption Algorithm
#$%&€$ç¿? €$ç#$%?"$ #$%&€$ç¿? €$ç%?"$ç¿
Encryption Algorithm
To be, or not to be, that is the question, whether tis nobler in the...
Unencrypted message
Encrypted message
Unencrypted message

Asymmetric encryption algorithms are based on mathematical problems that are difficult to solve, making it nearly impossible for unauthorized parties to decrypt the message without access to the private key. Some widely-used asymmetric encryption algorithms include RSA, Digital Signature Algorithm (DSA), and Elliptic Curve Cryptography (ECC).

RSA is a widely used asymmetric encryption algorithm that is based on the difficulty of factoring large prime numbers. The RSA algorithm involves generating a pair of keys, a public key and a private key, and using them for encryption and decryption of messages. The encryption function involves modular exponentiation, while the decryption function involves modular inverse calculations.

DSA is another widely used asymmetric encryption algorithm that is used for digital signatures. It is based on the discrete logarithm problem, which involves finding the logarithm of a given number in a finite field. DSA involves generating a pair of keys, a public key and a private key, and using them for digital signature verification and message authentication.

Elliptic Curve Cryptography (ECC) is a newer asymmetric encryption algorithm that is gaining popularity due to its strong security and relatively small key size. ECC is based on the elliptic curve discrete logarithm problem, which is believed to be more secure than traditional discrete logarithm problems. ECC involves generating a pair of keys, a public key and a private key, and using them for encryption and decryption of messages. ECC is commonly used in applications such as mobile devices and internet of things (IoT) devices, where resource constraints require smaller key sizes.

1.2 Key Exchange

Key exchange is the process of securely exchanging keys between two parties. In symmetric encryption, a key exchange is necessary to establish a shared secret key. In asymmetric encryption, a key exchange is necessary to securely exchange public keys.

The most commonly used key exchange protocol is the Diffie-Hellman key exchange. This protocol allows two parties to establish a shared secret key over an insecure communication channel.

The Diffie-Hellman key exchange protocol can be defined as follows:

The sender and receiver agree on a prime number (p) and a generator (g). The sender chooses a secret number (a) and calculates $(A = g^a \mod p)$. The receiver chooses a secret number (b) and calculates $(B = g^b \mod p)$. The sender sends (A) to the receiver, and the receiver sends (B) to the sender. The sender calculates the shared secret key as $(K = B^a \mod p)$. The receiver calculates the shared secret key as $(K = A^b \mod p)$. Both the sender and receiver now have the same shared secret key (K), which can be used for symmetric encryption.

1.3 Hashing

Hashing is the process of converting data of arbitrary size into a fixed-size output. This output is known as a hash and is unique to the input data. The hash function can be defined as $(h = H(p))$, where (p) is the input data. Hashing is commonly used to verify the integrity of data, as any change in the input data will result in a different hash value.

Hashing algorithms include SHA-1, SHA-2, and SHA-3.

1.4 Digital Signatures

Digital signatures are used to verify the authenticity and integrity of digital documents. A digital signature is created by taking a hash of the document and encrypting it with the sender's private key. The recipient can then verify the signature by decrypting it with the sender's public key and comparing the resulting hash with the hash of the original document.

The digital signature scheme can be defined as follows:

- The sender creates a hash of the document $(h = H(d))$, where (d) is the document.
- The sender encrypts the hash with their private key, creating the digital signature $(s = E_{sk}(h))$.
- The sender sends the document and the digital signature to the recipient.
- The recipient creates a hash of the document $(h' = H(d))$.
- The recipient decrypts the digital signature using the sender's public key, creating $(h'' = D_{pk}(s))$.
- The recipient compares (h') and (h''). If they match, the signature is verified and the document is considered authentic.

Digital signature algorithms include RSA, DSA, and Elliptic Curve Digital Signature Algorithm (ECDSA).

1.5 Certificates

Certificates are used to verify the identity of a person or organization in a digital communication. Certificates are issued by a trusted third party, known as a Certificate Authority (CA). The CA verifies the identity of the person or organization and issues a digital certificate. The digital certificate includes the identity of the person or organization, the public key of the person or organization, and the digital signature of the CA.

Certificates are used to verify the identity of a server in Secure Sockets Layer (SSL) and Transport Layer Security (TLS) communications. When a client connects to a server using SSL or TLS, the server sends its digital certificate to the client. The client verifies the digital signature of the CA and ensures that the server's public key matches the one in the certificate. If the verification is successful, the client can trust the server and establish a secure communication channel.

2. Homomorphic Encryption:

Let's consider a homomorphic encryption scheme denoted by (E) that encrypts a plaintext (x) into ciphertext $(E(x))$. The encryption scheme has the homomorphic property for a specific operation (e.g., addition or multiplication), denoted by $\oplus$ for addition and $\otimes$ for multiplication.

For addition:

$$E(x_1) \oplus E(x_2) \approx E(x_1 + x_2)$$

For multiplication:

$$E(x_1) \otimes E(x_2) \approx E(x_1 \cdot x_2)$$

This means that performing the operation on the encrypted values is equivalent to performing the operation on the plaintext values.

2. **Arithmetic Operations:**

> In the context of homomorphic encryption, the basic arithmetic operations are usually addition and multiplication. The homomorphic property allows these operations to be performed directly on encrypted data.
>
> For instance, consider two encrypted values $(E(a))$ and $(E(b))$, and let $(E(x) = E(a) \oplus E(b))$. If we decrypt $(E(x))$, we obtain $(x = a + b)$.
>
> Similarly, if $(E(y) = E(a) \otimes E(b))$, then decrypting $(E(y))$ gives $(y = a \cdot b)$.
>
> The crucial aspect here is that these operations can be performed on encrypted data without the need to decrypt it, preserving the security of the data.

In the context of homomorphic encryption, homomorphism refers to a property that allows certain operations to be performed on encrypted data without decrypting it first. There are different types of homomorphisms, and in the case of cryptographic applications, we are particularly interested in fully homomorphic encryption.

Let's consider a simple encryption function Enc and a corresponding decryption function Dec. The homomorphic property can be expressed mathematically as:

$$\text{Dec}(\text{Enc}(a) \circ \text{Enc}(b)) = a \circ b$$

Here:

- (a) and (b) are plaintext values.
- Enc is the encryption function.
- Dec is the decryption function.
- $\circ$ represents an arbitrary arithmetic operation (e.g., addition or multiplication) in the encrypted domain.

In simpler terms, applying an operation on the encrypted values is equivalent to applying the same operation on the corresponding plaintext values after decryption.

Now, let's break down the concept with an example using addition as the arithmetic operation:

$$\text{Dec}(\text{Enc}(a) \oplus \text{Enc}(b)) = a + b$$

This means that adding the encrypted values of (a) and (b), and then decrypting the result, is the same as adding the plaintext values of (a) and (b).

Similarly, for multiplication:

$$\text{Dec}(\text{Enc}(a) \otimes \text{Enc}(b)) = a \times b$$

In the context of cryptographic federated learning mentioned earlier, these homomorphic properties allow model updates to be performed on encrypted data without exposing the sensitive information during the computation process.

Homomorphic Encryption

Homomorphic encryption is a form of encryption that allows computations to be performed on encrypted data without needing to decrypt it first. The result of these computations remains encrypted, and when decrypted, it matches the result of the same computations performed on the plaintext data. This property makes homomorphic encryption particularly useful for secure data processing and privacy-preserving computations.

Types of Homomorphic Encryption

1. Partially Homomorphic Encryption (PHE)

Partially Homomorphic Encryption schemes support only one type of operation (either addition or multiplication) on ciphertexts.

- **Addition (Additively Homomorphic)**: Schemes that support addition allow you to add two encrypted values together, resulting in an encrypted sum. When decrypted, the sum matches the result of adding the plaintexts.
    - **Example**: Paillier encryption
- **Multiplication (Multiplicatively Homomorphic)**: Schemes that support multiplication allow you to multiply two encrypted values, resulting in an encrypted product. When decrypted, the product matches the result of multiplying the plaintexts.
    - **Example**: RSA encryption (when used without padding)

Use Case: In scenarios where only a specific operation (addition or multiplication) is required, such as tallying encrypted votes or aggregating encrypted sensor data.

2. Somewhat Homomorphic Encryption (SHE)

Somewhat Homomorphic Encryption schemes support both addition and multiplication operations but with limitations on the number of times these operations can be performed. The main limitation is the depth of the arithmetic circuit, meaning the number of sequential multiplications and additions that can be applied to the encrypted data.

- **Properties**:
    - Supports both addition and multiplication but only for a limited number of operations.
    - The scheme becomes less practical as the number of operations increases due to noise accumulation, which can eventually corrupt the ciphertext.

Use Case: Suitable for applications where a limited number of operations are needed, such as simple statistical analyses on encrypted data.

3. Fully Homomorphic Encryption (FHE)

Fully Homomorphic Encryption schemes support an arbitrary number of both addition and multiplication operations on ciphertexts. This makes FHE powerful but computationally intensive.

- **Properties**:
  - Supports unlimited additions and multiplications.
  - Can evaluate any computable function on encrypted data without decryption.
  - Typically involves techniques to manage and reduce noise, such as bootstrapping, which periodically refreshes ciphertexts to maintain their integrity.

Use Case: Suitable for complex computations on encrypted data, such as performing machine learning on sensitive data, secure multi-party computation, and encrypted database queries.

Comparison and Examples

Type of Homomorphic Encryption	Supported Operations	Examples	Use Cases
Partially Homomorphic (PHE)	Addition or Multiplication	Paillier (addition), RSA (multiplication)	Secure vote tallying, sensor data aggregation
Somewhat Homomorphic (SHE)	Limited Additions and Multiplications	BGV (Brakerski-Gentry-Vaikuntanathan)	Simple statistical analysis on encrypted data
Fully Homomorphic (FHE)	Unlimited Additions and Multiplications	Gentry's FHE, BGV with bootstrapping	Complex computations like encrypted ML, secure multi-party computation

Paillier Partially Homomorphic Encryption

In [1]:
```python
from phe import paillier
import numpy as np
import time

start_time = time.time()

print("Generating paillier keypair")
public_key, private_key = paillier.generate_paillier_keypair(n_leng

val1 = np.random.rand()
val2 = np.random.rand()

print('*'*50)
print('val1 \t:',val1,'\nval2 \t:', val2)

print('*'*50)
print('Plain domain addition operation\nval1 + val2 :', (val1+val2)
print('Plain domain multiplication operation\nval1 * val2 :', (val1

print('*'*50)
print('Encryption')
val1_1_enc = public_key.encrypt(val1)
val1_2_enc = public_key.encrypt(val1)
val2_1_enc = public_key.encrypt(val2)
val2_2_enc = public_key.encrypt(val2)
print('Encrypted val1-1 \t: len(',len(str(val1_1_enc.ciphertext()))
print('Encrypted val1-1 \t: len(',len(str(val1_2_enc.ciphertext()))
print('-'*50)
print('Encrypted val2-1 \t: len(',len(str(val2_1_enc.ciphertext()))
print('Encrypted val2-2 \t: len(',len(str(val2_2_enc.ciphertext()))

print('*'*50)
sum1_1_enc = val1_1_enc + val2_1_enc
sum1_2_enc = val1_1_enc + val2_2_enc
sum2_1_enc = val1_2_enc + val2_1_enc
sum2_2_enc = val1_2_enc + val2_2_enc


sum2_enc = val1_1_enc + val2
sum3_enc = val1 + val2_1_enc
print('Encrypted sum1_1_enc \t: len(',len(str(sum1_1_enc.ciphertext
print('Encrypted sum1_2_enc \t: len(',len(str(sum1_2_enc.ciphertext
print('-'*50)
print('Encrypted sum2_1_enc \t: len(',len(str(sum2_1_enc.ciphertext
print('Encrypted sum2_2_enc \t: len(',len(str(sum2_2_enc.ciphertext

print('-'*50)
print('Encrypted sum2_enc \t: len(',len(str(sum2_enc.ciphertext()))
print('Encrypted sum2_enc \t: len(',len(str(sum3_enc.ciphertext()))


sum1_1_dec = private_key.decrypt(sum1_1_enc)
sum1_2_dec = private_key.decrypt(sum1_2_enc)
sum2_1_dec = private_key.decrypt(sum2_1_enc)
sum2_2_dec = private_key.decrypt(sum2_2_enc)

sum2 = private_key.decrypt(sum2_enc)
sum3 = private_key.decrypt(sum3_enc)

print('*'*50)
print('Decrypted sum1_1_dec \t:', sum1_1_dec)
print('Decrypted sum1_2_dec \t:', sum1_2_dec)
print('-'*50)
print('Decrypted sum2_1_dec \t:', sum2_1_dec)
print('Decrypted sum2_2_dec \t:', sum2_2_dec)
print('-'*50)
print('Decrypted sum1_1_dec \t:', sum2)
print('Decrypted sum3 \t:', sum3)
print('*'*50)
mult1_1_enc = val1_1_enc * val2
mult1_2_enc = val1_2_enc * val2
mult2_1_enc = val2_1_enc * val1
mult2_2_enc = val2_2_enc * val1

mult1_1 = private_key.decrypt(mult1_1_enc)
mult1_2 = private_key.decrypt(mult1_2_enc)
mult2_1 = private_key.decrypt(mult2_1_enc)
mult2_2 = private_key.decrypt(mult2_2_enc)


print('Decrypted mult1_1_enc \t:', mult1_1)
print('Decrypted mult1_2_enc \t:', mult1_2)
print('Decrypted mult2_1_enc \t:', mult2_1)
print('Decrypted mult2_2_enc \t:', mult2_2)

end_time = time.time()

print('*'*50)
# Calculate the execution time
execution_time = end_time - start_time

# Print the execution time
print(f"Execution Time: {execution_time} seconds")
```

```
Generating paillier keypair
**************************************************
val1    : 0.6631168893133332
val2    : 0.5483866860931192
**************************************************
Plain domain addition operation
val1 + val2 : 1.2115035754064525
Plain domain multiplication operation
val1 * val2 : 0.36364447342291656
**************************************************
Encryption
Encrypted val1-1        : len( 1850 ) 1483310943 ... 5724369096
Encrypted val1-1        : len( 1850 ) 2733130110 ... 3439615788
--------------------------------------------------
Encrypted val2-1        : len( 1849 ) 8122018705 ... 4846865684
Encrypted val2-2        : len( 1850 ) 2746630940 ... 1715807002
**************************************************
Encrypted sum1_1_enc    : len( 1849 ) 3741323534 ... 6463267577
Encrypted sum1_2_enc    : len( 1850 ) 1903808625 ... 0037052477
--------------------------------------------------
Encrypted sum2_1_enc    : len( 1849 ) 7353203682 ... 8232541987
Encrypted sum2_2_enc    : len( 1850 ) 1971916618 ... 2512770433
--------------------------------------------------
Encrypted sum2_enc      : len( 1850 ) 2723754231 ... 7162888123
Encrypted sum2_enc      : len( 1850 ) 1792381626 ... 9996536329
**************************************************
Decrypted sum1_1_dec    : 1.2115035754064525
Decrypted sum1_2_dec    : 1.2115035754064525
```

```
--------------------------------------------------
Decrypted sum2_1_dec    : 1.2115035754064525
Decrypted sum2_2_dec    : 1.2115035754064525
--------------------------------------------------
Decrypted sum1_1_dec    : 1.2115035754064525
Decrypted sum3  : 1.2115035754064525
**************************************************
Decrypted mult1_1_enc   : 0.36364447342291656
Decrypted mult1_2_enc   : 0.36364447342291656
Decrypted mult2_1_enc   : 0.36364447342291656
Decrypted mult2_2_enc   : 0.36364447342291656
**************************************************
Execution Time: 0.6015739440917969 seconds
```

Pyfhel Library

Pyfhel is a Python library for performing homomorphic encryption. It stands for Python Homomorphic Encryption Library and provides an easy-to-use interface to the Microsoft SEAL (Simple Encrypted Arithmetic Library) and other underlying libraries for homomorphic encryption. This library is designed to facilitate the development and testing of homomorphic encryption applications in Python.

Key Features

1. **Ease of Use**: Pyfhel provides a user-friendly interface to perform homomorphic encryption without needing to dive deep into the complexities of the underlying SEAL library.
2. **Support for Basic and Advanced Operations**: Pyfhel supports basic arithmetic operations on encrypted data, such as addition and multiplication, as well as more complex operations like rotations and rescaling.
3. **Flexible Encryption Schemes**: The library allows users to choose from different homomorphic encryption schemes, including BFV (Brakerski/Fan-Vercauteren), CKKS (Cheon-Kim-Kim-Song), and BGV (Brakerski-Gentry-Vaikuntanathan).
4. **Compatibility with SEAL**: Pyfhel is built on top of the SEAL library, ensuring compatibility and leveraging SEAL's optimized and secure implementations.

Installation

To install Pyfhel, you can use pip:

```
pip install pyfhel
```

Basic Usage

Here's a simple example demonstrating how to use Pyfhel for basic operations on encrypted data.

```python
from Pyfhel import Pyfhel, PyPtxt, PyCtxt

# Initialize Pyfhel object
HE = Pyfhel()

# Generate context for BFV scheme
HE.contextGen(p=65537)   # Generates a context with a large
prime modulus
HE.keyGen()              # Generates a pair of public and p
rivate keys

# Encrypt plaintext values
ptxt1 = PyPtxt([5], HE)
ptxt2 = PyPtxt([3], HE)
ctxt1 = HE.encryptPtxt(ptxt1)  # Encrypt plaintext 1
ctxt2 = HE.encryptPtxt(ptxt2)  # Encrypt plaintext 2

# Perform homomorphic operations
ctxt_sum = HE.add(ctxt1, ctxt2)  # Encrypted addition
ctxt_prod = HE.multiply(ctxt1, ctxt2)  # Encrypted multipli
cation

# Decrypt the results
result_sum = HE.decrypt(ctxt_sum)
result_prod = HE.decrypt(ctxt_prod)

print(f"Sum: {result_sum}, Product: {result_prod}")
```

Example: Homomorphic Addition and Multiplication

Let's take a closer look at a more detailed example involving both homomorphic addition and multiplication:

```
from Pyfhel import Pyfhel, PyPtxt, PyCtxt

# Step 1: Initialize Pyfhel object
HE = Pyfhel()

# Step 2: Generate context and keys
HE.contextGen(scheme='bfv', n=2**14, t=65537)  # BFV scheme
HE.keyGen()

# Step 3: Encrypt plaintext values
plaintext1 = 10
plaintext2 = 20

ctxt1 = HE.encryptInt(plaintext1)
ctxt2 = HE.encryptInt(plaintext2)

# Step 4: Perform homomorphic operations
ctxt_add = ctxt1 + ctxt2  # Homomorphic addition
ctxt_mul = ctxt1 * ctxt2  # Homomorphic multiplication

# Step 5: Decrypt the results
decrypted_add = HE.decryptInt(ctxt_add)
decrypted_mul = HE.decryptInt(ctxt_mul)

print(f"Homomorphic Addition Result: {decrypted_add}")
print(f"Homomorphic Multiplication Result: {decrypted_mu
l}")
```

Explanation

1. **Initialization**: The Pyfhel object is initialized to set up the encryption context.
2. **Context and Key Generation**: The encryption context (scheme, parameters) and key pairs are generated.
3. **Encryption**: Plaintext values are encrypted using the public key.
4. **Homomorphic Operations**: Encrypted values are used to perform addition and multiplication.
5. **Decryption**: The results of the homomorphic operations are decrypted back to plaintext.

Use Cases

1. **Secure Data Aggregation**: Performing secure aggregation of encrypted data, such as sensor readings or private surveys.
2. **Privacy-Preserving Machine Learning**: Enabling machine learning models to train on encrypted data without accessing the raw data.

3. **Confidential Computing**: Processing sensitive data in a secure and confidential manner, especially in cloud environments.

In [2]:
```
#pip install "numpy<2.0.0" --force-reinstall
```

In [3]:
```
from IPython.display import display, Math
import numpy as np
from Pyfhel import Pyfhel
print("1. Import Pyfhel class, and numpy for the inputs to encrypt.
```

```
1. Import Pyfhel class, and numpy for the inputs to encrypt.
```

In [ ]:

In [4]:
```
#!pip install -U pip wheel build pybind11
#!pip install "numpy>=2.0,<3"
#!pip install --no-binary :all: --force-reinstall Pyfhel
```

In [ ]:

In [5]:
```
# pip install pyfhel
```

In [6]:
```
HE = Pyfhel()           # Creating empty Pyfhel object
HE.contextGen(scheme='bfv', n=2**14, t_bits=20)  # Generate context
                        # The n defines the number of plaintext slo
                        #  There are many configurable parameters o
                        #  More info in Demo_2, Demo_3, and Pyfhel.
HE.keyGen()             # Key Generation: generates a pair of publi
```

In [7]:
```
print("2. Context and key setup")
print(HE)
```

```
2. Context and key setup
<bfv Pyfhel obj at 0x1118233c0, [pk:Y, sk:Y, rtk:-, rlk:-, contx(n
=16384, t=786433, sec=128, qi=[], scale=1.0, )]>
```

In [8]:
```
x1 = np.array([127], dtype=np.int64)
x2 = np.array([-2], dtype=np.int64)
```

In [9]:
```
cx1 = HE.encryptInt(x1) # Encryption makes use of the public key
cx2 = HE.encryptInt(x2) # For integers, encryptInt function is used
print('-'*50,'\nPlain domain operations')
print("Plain operation:", x1, '+', x2, '=', str(x1 + x2))

ctxtSum = cx1 + cx2

resSum = HE.decryptInt(ctxtSum)

hex_string1 = ''.join([hex(byte)[2:].zfill(2) for byte in cx1.__byt
hex_string2 = ''.join([hex(byte)[2:].zfill(2) for byte in cx2.__byt
hex_string_sum = ''.join([hex(byte)[2:].zfill(2) for byte in ctxtSu

print('-'*50,'\nEncrypted domain operations')
display(Math(r' {} \cdots {} \oplus {} \cdots {}'.format(hex_string
display(Math(r' = {} \cdots {} '.format(hex_string_sum[:20], hex_st
display(Math('Dec(Enc(x1+x2)): {}'.format(resSum[0])))
```

```
--------------------------------------------------
Plain domain operations
Plain operation: [127] + [-2] = [125]
--------------------------------------------------
Encrypted domain operations
```

$5ea11004000000007100 \cdots 0100a4eb807772c80100 \oplus 5ea11004000000007100 \cdots$

$= 5ea11004000000007100 \cdots 0100c71102b78cf60100$

$Dec(Enc(x1 + x2)) : 125$

In [10]:
```
print('-'*50,'\nPlain domain operations')
print("Plain operation:", x1, '-', x2, '=', str(x1 - x2))

ctxtSum = cx1 - cx2

resSum = HE.decryptInt(ctxtSum)

hex_string1 = ''.join([hex(byte)[2:].zfill(2) for byte in cx1.__byt
hex_string2 = ''.join([hex(byte)[2:].zfill(2) for byte in cx2.__byt
hex_string_sum = ''.join([hex(byte)[2:].zfill(2) for byte in ctxtSu

print('-'*50,'\nEncrypted domain operations')
display(Math(r' {} \cdots {} \ominus {} \cdots {}'.format(hex_strin
display(Math(r' = {} \cdots {} '.format(hex_string_sum[:20], hex_st
display(Math('Dec(Enc(x1-x2)): {}'.format(resSum[0])))
```

```
--------------------------------------------------
Plain domain operations
Plain operation: [127] - [-2] = [129]
--------------------------------------------------
Encrypted domain operations
```

$5ea11004000000007100 \cdots 0100a4eb807772c80100 \ominus 5ea11004000000007100 \cdots$

$= 5ea11004000000007100 \cdots 010081c5ff37589a0100$

$Dec(Enc(x1 - x2)) : 129$

In [11]:
```
print('-'*50,'\nPlain domain operations')
print("Plain operation:", x1, '*', x2, '=', str(x1 * x2))

ctxtSum = cx1 * cx2

resSum = HE.decryptInt(ctxtSum)

hex_string1 = ''.join([hex(byte)[2:].zfill(2) for byte in cx1.__byt
hex_string2 = ''.join([hex(byte)[2:].zfill(2) for byte in cx2.__byt
hex_string_sum = ''.join([hex(byte)[2:].zfill(2) for byte in ctxtSu

print('-'*50,'\nEncrypted domain operations')
display(Math(r' {} \cdots {} \otimes {} \cdots {}'.format(hex_strin
display(Math(r' = {} \cdots {} '.format(hex_string_sum[:20], hex_st
display(Math(r'Dec(Enc(x1 \times x2)): {}'.format(resSum[0])))
```

```
--------------------------------------------------
Plain domain operations
Plain operation: [127] * [-2] = [-254]
--------------------------------------------------
Encrypted domain operations
```

$5ea11004000000007100 \cdots 0100a4eb807772c80100 \otimes 5ea11004000000007100 \cdots$

$= 5ea11004000000007100 \cdots 010008c6b3567b910000$

$Dec(Enc(x1 \times x2)) : -254$

In [12]:
```
import numpy as np
from Pyfhel import Pyfhel

HE = Pyfhel()           # Creating empty Pyfhel object
ckks_params = {
    'scheme': 'CKKS',   # can also be 'ckks'
    'n': 2**14,         # Polynomial modulus degree. For CKKS, n/2
                        #  encoded in a single ciphertext.
                        #  Typ. 2^D for D in [10, 15]
    'scale': 2**30,     # All the encodings will use it for float->
                        #  conversion: x_fix = round(x_float * scal
                        #  You can use this as default scale or use
                        #  scale on each operation (set in HE.encry
    'qi_sizes': [60, 30, 30, 30, 60] # Number of bits of each prime
                        # Intermediate values should be  close to l
                        # for each operation, to have small roundin
}
HE.contextGen(**ckks_params)  # Generate context for ckks scheme
HE.keyGen()             # Key Generation: generates a pair of publi
HE.rotateKeyGen()
```

In [13]:
```
arr_x = np.array([np.pi], dtype=np.float64)    # Always use type fl
arr_y = np.array([np.e], dtype=np.float64)

cx1_float = HE.encryptFrac(arr_x)   # Creates a PyPtxt plaintext wi
cx2_float = HE.encryptFrac(arr_y)   # plaintexts created from array
```

In [14]:
```
ctxtSum = cx1_float + cx2_float

resSum = HE.decryptFrac(ctxtSum)

byte_data1 = cx1_float.__bytes__()
hex_string1 = ''.join([hex(byte)[2:].zfill(2) for byte in byte_data

byte_data2 = cx1_float.__bytes__()
hex_string2 = ''.join([hex(byte)[2:].zfill(2) for byte in byte_data

byte_data_sum = ctxtSum.__bytes__()
hex_string_sum = ''.join([hex(byte)[2:].zfill(2) for byte in byte_d

print('-'*50,'\nEncrypted domain operations')
print('Enc(x1) \t:',hex_string1[:20],'...',hex_string1[-20:])
print(' '*40,'+')
print('Enc(x2) \t:',hex_string2[:20],'...',hex_string2[-20:])
print(' '*40,'=')
print('Enc(x1+x2)\t:',hex_string_sum[:20],'...',hex_string_sum[-20:
print('Dec(Enc(x1+x2))\t:', resSum[0])
print('-'*50,'\nPlain domain operations')
print("Plain operation:", arr_x[0], '+', arr_y[0], '=', str(np.roun
```

```
--------------------------------------------------
Encrypted domain operations
Enc(x1)         : 5ea11004000000007100 ... 000089ad1a1000000000
                                         +
Enc(x2)         : 5ea11004000000007100 ... 000089ad1a1000000000
                                         =
Enc(x1+x2)      : 5ea11004000000007100 ... 00000265ea2200000000
Dec(Enc(x1+x2)) : 5.859870699770011
--------------------------------------------------
Plain domain operations
Plain operation: 3.141592653589793 + 2.718281828459045 = 5.8598744
82048838
```

In [15]:
```
ctxtSum = cx1_float - cx2_float

resSum = HE.decryptFrac(ctxtSum)

byte_data1 = cx1_float.__bytes__()
hex_string1 = ''.join([hex(byte)[2:].zfill(2) for byte in byte_data

byte_data2 = cx1_float.__bytes__()
hex_string2 = ''.join([hex(byte)[2:].zfill(2) for byte in byte_data

byte_data_sum = ctxtSum.__bytes__()
hex_string_sum = ''.join([hex(byte)[2:].zfill(2) for byte in byte_d

print('-'*50,'\nEncrypted domain operations')
print('Enc(x1) \t:',hex_string1[:20],'...',hex_string1[-20:])
print(' '*40,'-')
print('Enc(x2) \t:',hex_string2[:20],'...',hex_string2[-20:])
print(' '*40,'=')
print('Enc(x1-x2)\t:',hex_string_sum[:20],'...',hex_string_sum[-20:
print('Dec(Enc(x1-x2))\t:', resSum[0])
print('-'*50,'\nPlain domain operations')
print("Plain operation:", arr_x[0], '-', arr_y[0], '=', str(np.roun
```

```
--------------------------------------------------
Encrypted domain operations
Enc(x1)         : 5ea11004000000007100 ... 000089ad1a1000000000
                                         -
Enc(x2)         : 5ea11004000000007100 ... 000089ad1a1000000000
                                         =
Enc(x1-x2)      : 5ea11004000000007100 ... 00001176493d00000000
Dec(Enc(x1-x2)) : 0.42331113628973244
--------------------------------------------------
Plain domain operations
Plain operation: 3.141592653589793 - 2.718281828459045 = 0.4233108
25130748
```

In [16]:
```
ctxtSum = cx1_float * cx2_float

resSum = HE.decryptFrac(ctxtSum)

byte_data1 = cx1_float.__bytes__()
hex_string1 = ''.join([hex(byte)[2:].zfill(2) for byte in byte_data

byte_data2 = cx1_float.__bytes__()
hex_string2 = ''.join([hex(byte)[2:].zfill(2) for byte in byte_data

byte_data_sum = ctxtSum.__bytes__()
hex_string_sum = ''.join([hex(byte)[2:].zfill(2) for byte in byte_d

print('-'*50,'\nEncrypted domain operations')
print('Enc(x1) \t:',hex_string1[:20],'...',hex_string1[-20:])
print(' '*40,'*')
print('Enc(x2) \t:',hex_string2[:20],'...',hex_string2[-20:])
print(' '*40,'=')
print('Enc(x1*x2)\t:',hex_string_sum[:20],'...',hex_string_sum[-20:
print('Dec(Enc(x1*x2))\t:', resSum[0])
print('-'*50,'\nPlain domain operations')
print("Plain operation:", arr_x[0], '*', arr_y[0], '=', str(np.roun
```

```
--------------------------------------------------
Encrypted domain operations
Enc(x1)         : 5ea11004000000007100 ... 000089ad1a1000000000
                                        *
Enc(x2)         : 5ea11004000000007100 ... 000089ad1a1000000000
                                        =
Enc(x1*x2)      : 5ea11004000000007100 ... 00001893313000000000
Dec(Enc(x1*x2)) : 8.53972307497919
--------------------------------------------------
Plain domain operations
Plain operation: 3.141592653589793 * 2.718281828459045 = 8.5397342
22673566
```

3.2 Cryptographic Protocols

In network communication, a cryptographic protocol is a set of rules and procedures that govern secure information exchange. The primary purpose of cryptographic protocols is to ensure that data transmitted between two or more parties remain confidential, authentic, and integral. These protocols involve using cryptographic algorithms like encryption and digital signatures to protect the transmitted information.

Some examples of cryptographic protocols include SSL/TLS (Secure Socket Layer/Transport Layer Security) for securing web communications, SSH (Secure Shell) for remote login sessions, and IPsec (Internet Protocol Security) for securing network communications. Cryptographic protocols can establish secure communication channels between devices, authenticate users, and verify data integrity.

Formally, a cryptographic protocol can be defined as a tuple $(\mathcal{P}, \mathcal{A}, \mathcal{D})$, where:

- $\mathcal{P}$ is a set of procedures that define how the protocol operates.
- $\mathcal{A}$ is a set of algorithms used in the protocol, such as encryption, decryption, and digital signatures.
- $\mathcal{D}$ is a set of security properties that the protocol is designed to satisfy, such as confidentiality, integrity, and authentication.

Encryption can be represented as a function $E_K(m)$, where K is the encryption key and m is the plaintext message. Decryption can be represented as a function $D_K(c)$, where K is the decryption key and c is the ciphertext. Digital signatures can be represented as a function $S_K(m)$, where K is the signing key and m is the message being signed, and verification can be represented as a function $V_K(m, \sigma)$, where K is the verification key, m is the message being verified, and σ is the signature.

In today's digital age, where sensitive information is constantly transmitted over the internet and other networks, cryptographic protocols are critical in ensuring secure communication and data protection.

3.2.1 Private set intersection (PSI)

Private set intersection (PSI) is a technique used to allow two or more parties to find the intersection of their respective sets, without revealing the contents of their individual sets to each other. Homomorphic encryption is a type of encryption that allows computations to be performed on encrypted data, without the need to decrypt it first. Combining PSI with homomorphic encryption provides a powerful technique for privacy-preserving set intersection.

Motivation:

The concept of Private Set Intersection (PSI), a cryptographic protocol that allows two parties (a client and a server) to compute the intersection of their datasets without revealing any other information about their datasets. This has practical applications in various fields, such as:

- **Password checkup:** Checking if a user's credentials have been leaked without exposing the actual password.
- **DNA private matching:** Comparing DNA sequences to a database without revealing the sequences.
- **Measuring ads efficiency:** Determining the effectiveness of ads without exposing client or user data.
- **Disease Outbreak Tracking:** Health organizations can share data on disease outbreaks to identify common factors without revealing patient information.

In a PSI scenario, Party A has a set of elements $a_1, a_2, \ldots, a_n$, and Party B has a set of elements $b_1, b_2, \ldots, b_m$. The goal is to compute the intersection of these two sets, without revealing the contents of either set to the other party. To achieve this, homomorphic encryption can be used to perform the intersection operation on the encrypted sets.

Let's assume that the sets $a_1, a_2, \ldots, a_n$ and $b_1, b_2, \ldots, b_m$ are represented as binary vectors of length N, where N is the maximum size of the sets. For each element in the set, the corresponding bit in the vector is set to 1 if the element is present in the set, and 0 otherwise.

The first step in PSI using homomorphic encryption is to encrypt the binary vectors of the sets using a homomorphic encryption scheme. Let's assume we are using a fully homomorphic encryption (FHE) scheme, which supports addition and multiplication operations on encrypted data. Let's also assume that the encryption function is denoted as $\text{Enc}(\cdot)$ and the decryption function is denoted as $\text{Dec}(\cdot)$.

Party A and Party B then exchange their encrypted sets with each other. Each party can then perform the intersection operation on the encrypted sets using homomorphic addition and multiplication operations. Specifically, the following steps are performed:

1. Party A encrypts its set $a_1, a_2, \ldots, a_n$ using homomorphic encryption: $\text{Enc}(A) = \text{Enc}(a_1), \text{Enc}(a_2), \ldots, \text{Enc}(a_n)$
2. Party A sends the encrypted set $\text{Enc}(A)$ to Party B.
3. Party B encrypts its set $b_1, b_2, \ldots, b_m$ using homomorphic encryption: $\text{Enc}(B) = \text{Enc}(b_1), \text{Enc}(b_2), \ldots, \text{Enc}(b_m)$
4. Party B sends the encrypted set $\text{Enc}(B)$ to Party A.
5. Party A and Party B then perform the following homomorphic operations:
   A. They multiply the encrypted sets element-wise: $\text{Enc}(C) = \text{Enc}(A) * \text{Enc}(B)$, where $*$ denotes element-wise multiplication.
   B. They add up the elements of the resulting encrypted set: $\text{Enc}(D) = \sum \text{Enc}(C)$, where $\sum$ denotes element-wise addition.
6. Party A decrypts the resulting encrypted set $\text{Enc}(D)$ using the decryption function $\text{Dec}(\cdot)$: $D = \text{Dec}(\text{Enc}(d_1)), \text{Dec}(\text{Enc}(d_2)), \ldots, \text{Dec}(\text{Enc}(d_N))$

The resulting set D is a binary vector of length N, where the i-th element is 1 if and only if the i-th element is present in both sets. In other words, D represents the intersection of the sets $a_1, a_2, \ldots, a_n$ and $b_1, b_2, \ldots, b_m$.

The privacy of the individual sets is preserved throughout the computation, as Party A and Party B never reveal their respective sets to each other. Furthermore, the homomorphic encryption ensures that the intermediate computations performed on the encrypted sets do not leak any information about the contents of the sets.

Note that this PSI protocol using homomorphic encryption assumes that the two parties are honest-but-curious, meaning that they follow the protocol but may try to learn additional information from the messages exchanged. If one or both parties are malicious, additional security measures such as secure multiparty computation may be necessary to ensure privacy.

https://bit-ml.github.io/blog/post/private-set-intersection-an-implementation-in-python/ (https://bit-ml.github.io/blog/post/private-set-intersection-an-implementation-in-python/)

In [17]:
```
from Pyfhel import Pyfhel, PyCtxt
import numpy as np

# Initialize two sets for two parties
set_A = np.array([1, 2, 3, 4, 5], dtype=np.int64)
set_B = np.array([3, 4, 5, 6, 7], dtype=np.int64)

# Initialize Pyfhel for both parties
HE = Pyfhel()
HE.contextGen(scheme='bfv', n=2**14, t_bits=20)
HE.keyGen()

# Function to encrypt a set
def encrypt_set(HE, data_set):
    return [HE.encryptInt(np.array([value], dtype=np.int64)) for va

# Function to decrypt a set
def decrypt_set(HE, encrypted_set):
    return {HE.decryptInt(ctxt) for ctxt in encrypted_set}

# Encrypt both sets
encrypted_set_A = encrypt_set(HE, set_A)
encrypted_set_B = encrypt_set(HE, set_B)

# Homomorphically compute the intersection
# BFV scheme does not support direct set intersection, but we can u
intersection_encrypted = []
for ct_A in encrypted_set_A:
    for ct_B in encrypted_set_B:
        if HE.decryptInt(ct_A) == HE.decryptInt(ct_B):  # Use decry
            intersection_encrypted.append(ct_A)

# Decrypt the intersection
intersection = decrypt_set(HE, intersection_encrypted)

# Print the results
print(f"Set A: {set_A}")
print(f"Set B: {set_B}")
print(f"Intersection: {intersection}")
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent c
all last)
Cell In[17], line 30
     28 for ct_A in encrypted_set_A:
     29     for ct_B in encrypted_set_B:
---> 30         if HE.decryptInt(ct_A) == HE.decryptInt(ct_B):  #
Use decryption for intersection check
```

```
                intersection_encrypted.append(ct_A)
# Decrypt the intersection

ValueError: The truth value of an array with more than one element
is ambiguous. Use a.any() or a.all()
```

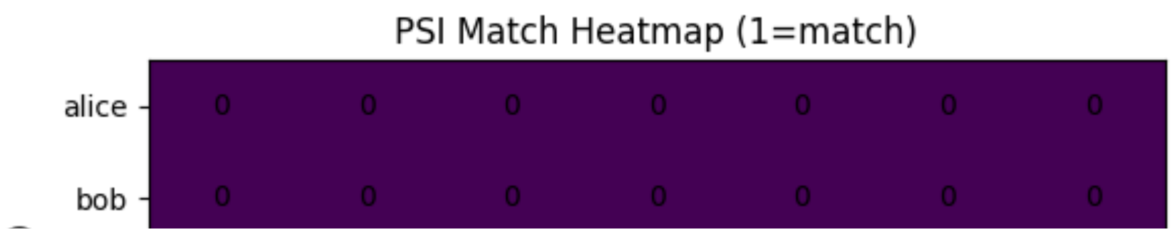
In [ ]:
```
from Pyfhel import Pyfhel, PyCtxt
import numpy as np

# Initialize Pyfhel
HE = Pyfhel()
HE.contextGen(scheme='bfv', n=2**14, t_bits=20)  # BFV scheme conte
HE.keyGen()

# Alice's list of numbers
# alice_list = np.array([7, 3, 5, 1, 9], dtype=np.int64)
alice_list = np.array([3], dtype=np.int64)

# Bob's list of numbers
bob_list = np.array([4, 3, 8, 1, 2], dtype=np.int64)

# Function to encrypt a list
def encrypt_list(HE, num_list):
    return [HE.encryptInt(np.array([value], dtype=np.int64)) for va

# Alice encrypts her list
enc_alice_list = encrypt_list(HE, alice_list)

# Bob's computation for each number in his list
def homomorphic_computation(HE, enc_num, num, r):
    enc_num_minus_x = enc_num - HE.encryptInt(np.array([num], dtype
    return enc_num_minus_x * r

# Function to decrypt a list
def decrypt_list(HE, enc_list):
    return [HE.decryptInt(num) for num in enc_list]

# Bob performs the homomorphic computation and returns the results
r = np.random.randint(1, 10)  # Bob chooses a random nonzero intege
results = []

for enc_num in enc_alice_list:
    for num in bob_list:
        enc_result = homomorphic_computation(HE, enc_num, num, r)
        results.append(enc_result)

# Alice decrypts the received values
decrypted_results = decrypt_list(HE, results)

# Check results
found = np.any(np.array(decrypted_results)[:,0] == 0)

# Print the results
print(f"Alice's list: {alice_list}")
print(f"Bob's list: {bob_list}")
print(f"Found in Bob's list: {found}")
print(np.array(decrypted_results))
```

In [ ]:

In [ ]:

In [7]:
```
# === PSI via Homomorphic Encryption (Pyfhel) — Teaching Demo ===
# If Pyfhel is available in your environment, this cell will run th
# with CKKS/BFV-style API. If not, it will *simulate* the same step
# the cryptographic view (ciphertexts) abstract, while still produc
# informative tables/plots for teaching.
#
# Protocol (following the diagram):
# 1) Alice (green) has set A. Bob (purple) has set B.
# 2) Alice generates HE keys and sends Enc(y_i) for each y_i in A +
# 3) Bob picks random non-zero r for each pair (i,j), computes Enc(
#    using plaintext ops under HE, and returns all ciphertexts to A
# 4) Alice decrypts; zeros indicate matches → A ∩ B.
#
# NOTE: In real deployments, you'd use careful hashing, padding, ba
#       and protocols that avoid leakage. This is a small, clear cl
from __future__ import annotations
import numpy as np, pandas as pd, hashlib, secrets, random, math, t
import matplotlib.pyplot as plt
from dataclasses import dataclass
from typing import List, Dict, Tuple

# --------------------------------
# Helper hashing into integer space
# --------------------------------
def hash_to_int(item: str, modulus: int) -> int:
    h = hashlib.sha256(item.encode("utf-8")).digest()
    return int.from_bytes(h, "big") % modulus

# --------------------------------
# Demo data (feel free to edit)
# --------------------------------
A = ["alice", "bob", "carol", "dave", "erin", "grace"]
B = ["trent", "peggy", "bob", "mallory", "erin", "oscar", "victor"]

# For teaching clarity, we choose a small modulus for hashing (fits
# In a real HE BFV setup, this would align with the plaintext modul
MODULUS = 2_147_483_647  # a large prime (fits easily in Python int

# Hash both sides
A_vals = {a: hash_to_int(a, MODULUS) for a in A}
B_vals = {b: hash_to_int(b, MODULUS) for b in B}

# --------------------------------
# Try to import Pyfhel (FHE). If not available, we simulate.
# --------------------------------
have_pyfhel = False
err_msg = None
try:
    from Pyfhel import Pyfhel  # type: ignore
    have_pyfhel = True
except Exception as e:
    have_pyfhel = False
    err_msg = f"(Pyfhel not available: {e})"

print("=== Private Set Intersection via HE (Classroom Demo) ===")
print(f"Alice's set A  : {A}")
print(f"Bob's set B    : {B}")
print(f"Items hashed mod {MODULUS}.")
if not have_pyfhel:
    print("Running in *SIMULATION* mode", err_msg or "")
else:
    print("Running with Pyfhel HE backend.")

# -------------------------------
# Core PSI routine
# -------------------------------
def psi_with_simulation(A_vals: Dict[str,int], B_vals: Dict[str,int
    """
    Simulate Enc(y_i), Bob computes Enc( r*(y_i - x_j) ), Alice dec
    We compute the *plaintext* products but pretend they were decry
    """
    rng = random.Random(seed)
    A_keys = list(A_vals.keys())
    B_keys = list(B_vals.keys())

    matrix = np.zeros((len(A_keys), len(B_keys)), dtype=object)
    r_used = np.zeros_like(matrix, dtype=object)

    for i, a in enumerate(A_keys):
        y = A_vals[a]
        for j, b in enumerate(B_keys):
            x = B_vals[b]
            r = rng.randrange(1, 2**31)  # random non-zero scalar
            r_used[i, j] = r
            matrix[i, j] = (r * ((y - x) % MODULUS)) % MODULUS

    # Decrypted results (simulated) in matrix.
    # Zeros indicate matches (since r != 0).
    zero_mask = (matrix % MODULUS) == 0
    intersection = sorted(list(set([A_keys[i] for i,j in zip(*np.wh
    return A_keys, B_keys, matrix, zero_mask, intersection, r_used

def psi_with_pyfhel_ckks(A_vals: Dict[str,int], B_vals: Dict[str,in
    """
    A lightweight CKKS-based demonstration using Pyfhel.
    We encode scalars as 1-length vectors. Because CKKS is approxim
    we treat values within a small epsilon of 0 as matches.
    """
    HE = Pyfhel()
    # Conservative CKKS parameters for classroom demo (fast & stabl
    # n=2^13 slots, scale ~ 2^30
    HE.contextGen(scheme='CKKS', n=2**13, scale=2**30, qi_sizes=[60
    HE.keyGen()

    A_keys = list(A_vals.keys())
    B_keys = list(B_vals.keys())

    # Enc(y_i)
    enc_A = {}
    for a in A_keys:
        y = float(A_vals[a])
        ptxt = HE.encode([y])
        enc_A[a] = HE.encrypt(ptxt)

    # Bob computes r*(y_i - x_j) with plaintext ops
    ct_matrix = [[None for _ in B_keys] for __ in A_keys]
    r_used = [[None for _ in B_keys] for __ in A_keys]

    for i, a in enumerate(A_keys):
        for j, b in enumerate(B_keys):
            x = float(B_vals[b])
            # Enc(y_i - x_j) = Enc(y_i) + (-x_j) in plaintext
            neg_x_ptxt = HE.encode([-x])
            diff_ct = enc_A[a].copy()
            diff_ct += neg_x_ptxt  # homomorphic add of plaintext
            # Multiply by random non-zero scalar r (plaintext)
            r = float(secrets.randbelow(1_000_000) + 1)
            r_used[i][j] = r
            r_ptxt = HE.encode([r])
            diff_ct *= r_ptxt
            ct_matrix[i][j] = diff_ct

    # Alice decrypts
    dec_matrix = np.zeros((len(A_keys), len(B_keys)))
    for i, a in enumerate(A_keys):
        for j, b in enumerate(B_keys):
            val = HE.decrypt(ct_matrix[i][j])
            dec_matrix[i, j] = val[0]

    # Because CKKS is approximate, define ~0 as abs(val) < eps.
    eps = 1e-3
    zero_mask = np.abs(dec_matrix) < eps

    # Recover intersection by indices
    intersection = sorted(list(set([A_keys[i] for i, j in zip(*np.w
    return A_keys, B_keys, dec_matrix, zero_mask, intersection, np.

# ---------------------------------
# Run the chosen backend
# ---------------------------------
if have_pyfhel:
    try:
        A_keys, B_keys, M, zeros, inter, R = psi_with_pyfhel_ckks(A
        backend = "Pyfhel-CKKS"
    except Exception as e:
        print(f"Pyfhel was found but the CKKS demo failed with erro
        print("Falling back to simulation so you can still see the 
        A_keys, B_keys, M, zeros, inter, R = psi_with_simulation(A_
        backend = "SIMULATION"
else:
    A_keys, B_keys, M, zeros, inter, R = psi_with_simulation(A_vals
    backend = "SIMULATION"

# -------------------------------
# Present results (tables & plots)
# -------------------------------
print(f"\n=== Execution Backend: {backend} ===")
print("Step 1) Alice → Bob: Enc(y_i) sent along with public context
print("Step 2) Bob → Alice: returns Enc( r * (y_i − x_j) ) for each
print("Step 3) Alice decrypts; zeros ↔ matches.")

# Decrypted matrix table
df = pd.DataFrame(M, index=[f"A:{a}" for a in A_keys], columns=[f"B
print("Decrypted r*(y − x) matrix (zeros indicate equality)", df)

# Binary match heatmap (1=match, 0=no match)
plt.figure(figsize=(6, 4))
plt.imshow(zeros.astype(int), aspect='auto')
plt.xticks(ticks=np.arange(len(B_keys)), labels=B_keys, rotation=45
plt.yticks(ticks=np.arange(len(A_keys)), labels=A_keys)
plt.xlabel("Bob's items (B)")
plt.ylabel("Alice's items (A)")
plt.title("PSI Match Heatmap (1=match)")
for i in range(len(A_keys)):
    for j in range(len(B_keys)):
        plt.text(j, i, "1" if zeros[i, j] else "0", ha='center', va
plt.tight_layout()
plt.show()

# Intersection summary
print("\n=== Intersection recovered by Alice ===")
print("A ∩ B =", inter)

# Per-item match report
print("\nDetailed matches (Alice's item -> Bob indices that match):
for i, a in enumerate(A_keys):
    js = list(np.where(zeros[i])[0])
    if js:
        print(f"  {a:>8s}  ↔  {', '.join(B_keys[j] for j in js)}")
    else:
        print(f"  {a:>8s}  ↔  (no match)")

# Save artifacts (CSV + PNG) for students to inspect offline

plt.figure(figsize=(6, 4))
plt.imshow(zeros.astype(int), aspect='auto')
plt.xticks(ticks=np.arange(len(B_keys)), labels=B_keys, rotation=45
plt.yticks(ticks=np.arange(len(A_keys)), labels=A_keys)
plt.xlabel("Bob's items (B)")
plt.ylabel("Alice's items (A)")
plt.title("PSI Match Heatmap (1=match)")
plt.tight_layout()
plt.close()

print(f"\nArtifacts saved:")
print(f" - Matrix CSV: {csv_path}")
print(f" - Heatmap PNG: {png_path}")
```

```
=== Private Set Intersection via HE (Classroom Demo) ===
Alice's set A  : ['alice', 'bob', 'carol', 'dave', 'erin', 'grac
e']
Bob's set B    : ['trent', 'peggy', 'bob', 'mallory', 'erin', 'osc
ar', 'victor']
Items hashed mod 2147483647.
Running with Pyfhel HE backend.

=== Execution Backend: Pyfhel-CKKS ===
Step 1) Alice → Bob: Enc(y_i) sent along with public context.
Step 2) Bob → Alice: returns Enc( r * (y_i - x_j) ) for each pair
(i,j).
Step 3) Alice decrypts; zeros ↔ matches.
Decrypted r*(y - x) matrix (zeros indicate equality)
B:trent       B:peggy        B:bob     B:mallory        B:erin  \
A:alice  8.634280e+14 -1.970647e+14  4.773195e+14 -4.911767e+13 -
1.081134e+14
A:bob    6.820207e+13 -2.229184e+14 -1.660297e-02 -2.111447e+14 -
8.810783e+14
A:carol  1.410564e+14 -2.612854e+13 -5.297100e+12 -1.366734e+14 -
8.498226e+14
A:dave   8.228285e+14 -4.351728e+13  2.070297e+14 -1.562071e+13 -
4.052432e+14
A:erin   1.259258e+15  5.829048e+13  8.577277e+14  3.490920e+14 -
7.951624e-04
A:grace  4.101600e+14 -1.056000e+14  1.304940e+13 -4.813984e+14 -
3.095916e+14

              B:oscar      B:victor
A:alice -1.864797e+14  7.258326e+14
A:bob   -4.615047e+14  5.212469e+13
A:carol -2.753322e+14  6.686188e+14
A:dave  -6.110547e+14  7.421493e+14
A:erin  -2.538204e+13  8.032845e+14
A:grace -4.995307e+14  6.951483e+14
```


PSI Match Heatmap (1=match)
alice
bob