Exploring Homomorphic Encryption with Python

Abhishek Tiwari

Citation: *A. Tiwari*, ”Exploring Homomorphic Encryption with Python”, Abhishek Tiwari, 2024. doi:10.59350/vr6dm-7r102

Published on: October 19, 2024

Homomorphic encryption is a powerful cryptographic technique that allows computations to be performed on encrypted data without decrypting it first. This blog post will introduce the concept of homomorphic encryption and demonstrate implementations using Python.

What is Homomorphic Encryption?

Homomorphic encryption is a form of encryption that allows specific types of computations to be carried out on ciphertext. The result is encrypted data that, when decrypted, matches the result of operations performed on the plaintext.

How Does Homomorphic Encryption Work?

To understand how homomorphic encryption works, let's break it down into key concepts:

Traditional Encryption vs. Homomorphic Encryption In traditional encryption, data is scrambled in such a way that it can only be unscrambled with the correct key. Once encrypted, the data can't be meaningfully manipulated without first decrypting it.

Homomorphic encryption allows for meaningful manipulation of the encrypted data. The magic lies in creating an encryption scheme where operations on the encrypted data correspond to operations on the original data.

Mathematical Foundations Homomorphic encryption schemes are built on complex mathematical structures and problems that are believed to be computationally hard to solve. These often involve concepts from number theory and abstract algebra.

Homomorphic Properties The key to homomorphic encryption is that certain mathematical operations on encrypted values correspond to operations on the plaintext values. In general:

- For addition: $E(a) \oplus E(b) = E(a + b)$
- For multiplication: $E(a) \otimes E(b) = E(a * b)$

Where $E()$ represents the encryption function, and $\oplus$ and $\otimes$ represent operations on encrypted values.

Types of Homomorphic Encryption There are three main types of homomorphic encryption:

1. Partially Homomorphic Encryption (PHE): Supports either addition or multiplication, but not both.
2. Somewhat Homomorphic Encryption (SHE): Supports both addition and multiplication, but only for a limited number of operations.
3. Fully Homomorphic Encryption (FHE): Supports an unlimited number of both addition and multiplication operations.

A Simple Example

Let's start with a simple example using the Paillier cryptosystem, which is a partially homomorphic encryption (PHE) scheme that supports addition operations on encrypted data. We'll use the phe library in Python.

First, install the required library:

```
pip install phe
```

Now, let's write some Python code to demonstrate homomorphic encryption:

Listing 1: Example using Python phe package.

```python
from phe import paillier

def phe_example():
    # Generate public and private keys
    public_key, private_key = paillier.generate_paillier_keypair()

    # Encrypt two numbers
    a, b = 5, 3
    encrypted_a = public_key.encrypt(a)
    encrypted_b = public_key.encrypt(b)

    # Perform homomorphic addition
    encrypted_sum = encrypted_a + encrypted_b
    decrypted_sum = private_key.decrypt(encrypted_sum)

    # Perform homomorphic multiplication by a scalar
    scalar = 2
    encrypted_product = encrypted_a * scalar
    decrypted_product = private_key.decrypt(encrypted_product)

    print("PHE Results:")
    print(f"Original numbers: {a} and {b}")
    print(f"Encrypted sum: {decrypted_sum}")
    print(f"Encrypted product with scalar {scalar}: {decrypted_product}")
```

```
phe_example()
```

This example demonstrates: key generation, Encryption of user-input numbers, Homomorphic addition on encrypted values, and multiplication of one of the encrypted inputs by a scalar. Multiplication of two encrypted values is not supported by PHE. Output from above script,

Listing 2: Example output using Python phe package.

```
PHE Results:
Original numbers: 5 and 3
Encrypted sum: 8
Encrypted product with scalar 2: 10
```

Advanced Example

Now, let's explore a more powerful approach using Fully Homomorphic Encryption (FHE). We'll use the Concrete library, which is based on the TFHE (Fast Fully Homomorphic Encryption over the Torus) scheme.

First, install the required library:

```
pip install concrete-python
```

Now, let's create a simple example that performs basic operations on encrypted data:

Listing 3: Example using Python concrete package.

```
from concrete import fhe

def fhe_example():
    # Define a function to be executed homomorphically
    @fhe.compiler({"x": "encrypted", "y": "encrypted"})
    def add_and_multiply(x, y):
        return x + y, x * y

    inputset = [(2, 3), (0, 0), (1, 6), (7, 7), (7, 1), (3, 2), (6, 1),
       (1, 7), (4, 5), (5, 4)]

    # Compile the function
    circuit = add_and_multiply.compile(inputset)
    circuit.keygen()
    # Example values
    a, b = 5, 3

    # Perform homomorphic computation
    result = circuit.encrypt_run_decrypt(a, b)
    print("\nFHE Results:")
    print(f"Original numbers: {a} and {b}")
    print(f"Encrypted sum and product: {result}")

fhe_example()
```

This FHE example declares a function definition for addition and multiplication with the **fhe.compiler?** decorator, then compiles for specific input set, generates the key, and finally run operations on the encrypted user inputs. The Concrete library handles the complex task of translating our Python function into FHE operations. This abstraction makes it much easier to work with FHE. The encrypt_run_decrypt method handles the entire process of encryption, computation, and decryption.

Listing 4: Example output using Python concrete package.

```
FHE Results:
Original numbers: 5 and 3
Encrypted sum and product: (8, 15)
```

Subtraction and Division

While addition and multiplication are the fundamental operations in homomorphic encryption, subtraction and division can also be performed, albeit with some caveats. Let's explore how these operations are handled in both PHE (using the phe library) and FHE (using the concrete library).

With Paillier

In the Paillier cryptosystem, subtraction is straightforward, but division is more complex and limited. Subtraction is basically performed by adding the negative of the second encrypted number. Division is not directly supported so we use a logarithm-based approximation, which requires decrypting values. This approach is not secure for real-world applications and is shown here for demonstration purposes only.

Listing 5: Subtraction and division using phe

```
from phe import paillier
import numpy as np

def phe_subtraction_and_division():
    # Generate public and private keys
    public_key, private_key = paillier.generate_paillier_keypair()
    # Encrypt two numbers
    a, b = 10, 3
    encrypted_a = public_key.encrypt(a)
    encrypted_b = public_key.encrypt(b)

    # Subtraction
    encrypted_diff = encrypted_a - encrypted_b
    decrypted_diff = private_key.decrypt(encrypted_diff)

    # Division (approximation using logarithms)
    def encrypted_divide(encrypted_a, encrypted_b, public_key, private_key
        , precision=1000):
        log_a = np.log(private_key.decrypt(encrypted_a))
        log_b = np.log(private_key.decrypt(encrypted_b))
        encrypted_log_a = public_key.encrypt(log_a * precision)
        encrypted_log_b = public_key.encrypt(log_b * precision)
        encrypted_log_ratio = encrypted_log_a - encrypted_log_b
        return encrypted_log_ratio, precision

    encrypted_quotient, precision = encrypted_divide(encrypted_a,
        encrypted_b, public_key, private_key)
    decrypted_log_quotient = private_key.decrypt(encrypted_quotient)
    decrypted_quotient = np.exp(decrypted_log_quotient / precision)

    print("PHE Results:")
    print(f"Original numbers: {a} and {b}")
    print(f"Encrypted difference: {decrypted_diff}")
    print(f"Encrypted approximate quotient: {decrypted_quotient:.4f}")

phe_subtraction_and_division()
```

Listing 6: Output for subtraction and division using phe

```
PHE Results:
Original numbers: 10 and 3
Encrypted difference: 7
Encrypted approximate quotient: 3.3333
```

With Concrete

In FHE, we can perform both subtraction and division, but with some limitations due to the integer-based nature of most FHE schemes. Subtraction is directly supported in FHE.

Listing 7: Subtraction using concrete

```
from concrete import fhe

def fhe_subtraction():
    # Define functions to be executed homomorphically
    @fhe.compiler({"x": "encrypted", "y": "encrypted"})
    def subtract(x, y):
        return x - y

    # Compile the functions
    inputset = [(2, 3), (0, 0), (1, 6), (7, 7), (7, 1), (3, 2), (6, 1),
       (1, 7), (4, 5), (5, 4)]
    circuit = subtract.compile(inputset)
    circuit.keygen()

    # Example values
    a, b = 3, 9

    # Perform computations
    result = circuit.encrypt_run_decrypt(a, b)

    print("\nFHE Results:")
    print(f"Original numbers: {a} and {b}")
    print(f"Encrypted difference: {result}")

fhe_subtraction()
```

Listing 8: Output from subtraction using concrete

```
FHE Results:
Original numbers: 3 and 9
Encrypted difference: -6
```

Division in FHE is typically integer division (floor division) due to the integer-based nature of most FHE schemes. More complex operations like floating-point division would require additional techniques and approximations. In following example, we are using fhe.multivariate which allows defining functions with multiple encrypted inputs, which is crucial for operations like division.

Listing 9: Floor division using concrete

```
from concrete import fhe
import numpy

def fhe_div():
    # Define functions to be executed homomorphically
    @fhe.compiler({"x": "encrypted", "y": "encrypted"})
    def div(x, y):
        return fhe.multivariate(lambda x, y: x // y)(x, y)

    # Compile the functions
    w = 8
    inputset = [(numpy.random.randint(1, 2**w), numpy.random.randint(1,
       2**w)) for _ in range(100)]
    circuit = div.compile(inputset)
    circuit.keygen()

    # Example values
    a, b = 13, 3

    # Perform computations
    result = circuit.encrypt_run_decrypt(a, b)

    print("\nFHE Results:")
    print(f"Original numbers: {a} and {b}")
    print(f"Encrypted quotient: {result}")

fhe_div()
```

Listing 10: Output from floor division using concrete.

```
FHE Results:
Original numbers: 13 and 3
Encrypted quotient: 4
```

Applicability

PHE are useful for specific applications where only one type of operation is needed such secure voting systems where only addition is required or privacy-preserving data aggregation. FHE is applicable to a wide range of complex computation problems, including machine learning on encrypted data, secure multi-party computation, privacy-preserving measurement, etc.

Conclusion

Homomorphic encryption opens up exciting possibilities for privacy-preserving computation and data analysis. We've explored two different approaches: the partially homomorphic Paillier system and a fully homomorphic encryption system using the Concrete library.