Network Security Final Project Report

Sakshi Singh, Nischal Vangipuram, Hamza Sheikh

May 2023

1 Introduction

In today's era, with the exponential increase in the amount of data being generated, processed, and analyzed, there is a great need for secure and efficient methods to perform data processing. However, many scenarios exist where data owners hesitate to share their data with others, even with cloud providers, due to data privacy and security concerns. One such scenario is where Bob wants to perform complex computations on his personal data while taking advantage of the computing power of the cloud. [6]

Homomorphic encryption was proposed as a solution to that. This kind of encryption exposes underlying data characteristics without revealing private information. In the previous scenario, Bob can send its data to an untrusted vendor and analyze their machines without sending the raw data.

There are a lot of libraries out there that perform this kind of encryption, but not a clear analysis of their performance and limitations.

1.0.1 Libraries Used

We decided to use three libraries to benchmark and analyze the results and performance. Each library is written and used in a different language to broaden our analysis over multiple languages.

- C++ library **Microsoft SEAL** [1]
- Java library **Homomorphic Encryption** [2]
- Python Lang library **python-paillier** [3]

2 Related Work

Homomorphic encryption is a type of encryption technique that allows computation to be performed on encrypted data without needing to decrypt it first. In other words, it allows for computations to be done on data while it is still encrypted, which maintains the privacy of the data.

With homomorphic encryption, data is encrypted using a special algorithm, and the ciphertext can be used in computations that produce an encrypted result. This encrypted result can then be decrypted to obtain the result of the computation.

2.1 BFV

The BFV (Brakerski-Fan-Vercauteren) scheme is a homomorphic encryption scheme that supports both partially homomorphic encryption (PHE) and fully homomorphic encryption (FHE) on integers. It was first introduced in 2012 by Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan.

The BFV scheme is based on the learning with errors (LWE) problem, which is a hard mathematical problem used in many lattice-based cryptography schemes. It transforms plaintext messages into polynomials with integer coefficients, which are then encrypted using a unique key. The ciphertexts can be added and multiplied to perform homomorphic operations on the encrypted data.

2.2 CKKS

The CKKS (Cheon-Kim-Kim-Song) scheme is a homomorphic encryption scheme that supports fully homomorphic encryption (FHE) on real numbers. It was first introduced in 2017 by Jung Hee Cheon, Jinsu Kim, Yongsoo Song, and Hyunsoo Yoon.

The CKKS scheme is based on the Ring Learning with Errors (RLWE) problem, which is a variant of the LWE problem used in many lattice-based cryptography schemes. It encodes plaintext messages as complex vectors with real number coefficients, which are then encrypted using a special key. The ciphertexts can be added and multiplied to perform homomorphic operations on the encrypted data.

2.3 Paillier

The popular RSA technique and the Paillier encryption system entail exponentiating a very large base with a large exponent. In the Paillier cryptosystem, an integer serves as the fundamental plaintext (m) unit. Large integers are a good approximation for real numbers. Paillier Library is implementing the Paillier Partially Homomorphic Encryption. The ciphertext (for example, M1, M2,...) can undergo the following processes using the Paillier encryption scheme:

- Addition of a ciphertext with another plaintext: M1 + n1 = M2
- Multiplication of a ciphertext with another plaintext: M1 * n1 = M3
- Addition on two or more ciphertexts: M1 + M4 = M5

2.4 DGK

The DGK scheme is a homomorphic encryption scheme that was proposed by Damgard, Geisler, and Kroigaard in 2010. It is based on the idea of encoding bits as polynomials over a finite field. This scheme allows for efficient homomorphic operations on ciphertexts, specifically addition and multiplication, while maintaining the privacy of the plaintext.

3 Adversary Model

As mentioned above, Homomorphic encryption is used to allow the private data transfer to an untrusted third party/cloud provider. So the main adversary that needs to be protected against is the party in which the data is transferred to. The security is directly matched with the theory and use of the encryption schemes, and is proved in the literature. Other adversaries can be presented, and in this section we expose some of security issues, additional adversaries that can arise.

3.1 Security Caveats in Paillair

- **Information leakage:** Unencrypted numbers do not have unencrypted exponents. For floating point numbers, this results in information leakage about the size of the encrypted value by default. This leakage can be avoided by settling on a fixed value for each exponent as part of the protocol; after that, *decrease_exponent_to*() can be called for each EncryptedNumber before sharing. This exponent ought to serve as a lower constraint for any exponent that might organically develop.

- **Alternative Base for EncodedNumber:** Information about the underlying plaintext values can leak if the encryption exponents are not carefully managed. For example, if the encryption exponents are chosen randomly for each ciphertext, an attacker who knows the plaintext value of one ciphertext can subtract its encryption exponent from the encryption exponent of the result to obtain the encryption exponent of the other ciphertext. This effectively reveals information about the underlying plaintext value of the other ciphertext.

  This problem is exacerbated when smaller bases are used because the range of possible encryption exponents is smaller. This means that an attacker has a higher probability of guessing the encryption exponent and recovering information about the underlying plaintext value.

3.2 BFV encryption scheme Security Example:

This scheme, introduced before, can introduce a person-in-the-middle attack if not careful. As we mentioned repeatedly in class, even when using the most advanced cryptography, knowing how to use it is also very important. The scheme introduces a noise budget, which dictates how many computations can be performed on the value before it needs to be re-encrypted or relinearized. Keep in mind that both re-encryption or re-linearization are expensive to perform. A simple example of the attack is as follows:

User sends encrypted data in non-encrypted message, asking the service provider to multiply the value by 3.

person-in-the-middle intercepts the message, and change the computation, adding a multiply by 3 and multiply by 1/3. This can result in three scenarios:

1. **Noise budget is not reached:** The person-in-the-middle might be extra weary of being caught, so the noise budget is just decreased, leading the user for an expensive relinearization operation

2. **Noise budget reaches 0:** If the noise budget is reached, the result can be random, so the whole interaction is jeopardized. We test this in SEAL, by squaring an encrypted number multiple times, until it reaches its noise budget. The following code does exactly that.

```
uint64_t x = 6;
Plaintext x_plain(uint64_to_hex_string(x));
Ciphertext x_encrypted;
encryptor.encrypt(x_plain, x_encrypted);
Ciphertext x_plus_one_sq;
evaluator.add_plain(x_encrypted, plain_one
                    , x_plus_one_sq);

for (int i = 0; i < 3; i++){
    evaluator.square_inplace(x_plus_one_sq);
}
```

   The result of this code should be 5764801, but the actual result varies since the noise budget is reached. We ran this experiment 3 times, and the results are 32, 129, and 1.

3. **Noise budget used fully:** The SEAL library returns an error, deeming the operation invalid between the user and third party *libc++abi: terminating with uncaught exception of type std* :: *invalid_argument* : *invalidsize*

3.3 DGK Encryption Scheme Adversary Model:

In the DGK scheme, the size of the plaintext space is fixed, and any computation on the ciphertexts should preserve the size of the plaintext space. If the result of an operation on the ciphertexts exceeds the size of the plaintext space, it will be reduced to modulo N, which may result in incorrect results. For example, if the plaintext space is from 0 to 127, and the sum of two ciphertexts is 200, the result will be reduced modulo 128, resulting in 72 instead of the expected 200. Hence, the DGK scheme can be vulnerable to a man-in-the-middle attack where an adversary can exploit it by choosing a large number to add to a ciphertext, causing the result to exceed the size of the plaintext space and thus yield incorrect results.

4 Methodology

Our methodology first compares different libraries widely used for Homomorphic Encryption, in addition to comparing different machine learning use cases and expose their performance, overheads, advantages, and disadvantages.

4.1 Libraries Limitations

4.1.1 Python-Paillier

Paillier encryption is only defined for non-negative integers less than PaillierPublicKey.n.

We cannot multiply two encrypted numbers together. This is the limit of the Paillier cryptosystem.

4.1.2 DGK Scheme(Java)

Limited plaintext space: The size of the plaintext space is fixed, and any computation on the ciphertexts should preserve the size of the plaintext space. If the result of an operation on the ciphertexts exceeds the size of the plaintext space, it will be reduced modulo N, which may result in incorrect results. **Limited homomorphic operations:** The DGK scheme does not support homomorphic multiplication of two ciphertexts

4.1.3 Microsoft SEAL (C++)

There is no scheme for creating encryption decryption on custom data. BFV is an encryption model for integers only

4.2 Libraries Performance Analysis

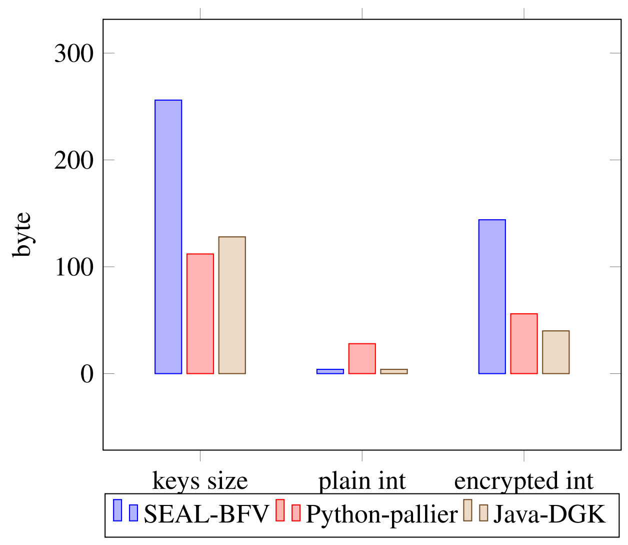
4.2.1 Ease of Use

We found that the python library is easy to use and learn. On the other hand, the SEAL library have a steeper learning curve and needs extensive c++ knowledge to sort out dependencies, but offers a high customizable library with different schemes and parameter tuning. The java library sits in between, with less customization options.

4.2.2 Time

Screen Shot 2023-05-01 at 9.56.44 AM.png
The above bar chart shows the time it takes to perform operations using the three different libraries. The y axis is a log scale in microseconds, as the time taken between libraries is substantially different.

4.2.3 Memory Usage



5 Machine Learning Use case Implementation and Comparison

5.1 Send encrypted model to the User

In this scenario, a third party uses their data to train the model, then delivers the users the encrypted model. The trained model is signed by a third party using their private key, and they send the model and public key along with it so that the user may use the public key to confirm the signature. The user then applies their dataset to the encrypted trained model and receives the encrypted output. The following user sends the obtained encrypted data to the third party; the third party uses the private key to decrypt the data and send the decrypted data to the user.

Advantages:

- **No encryption on the User side:** The third-party trains the model on their dataset, encrypts it and sends it to the user. The user uses their own data set and gets the encrypted output.
- **Third party does not know the data:** Users use an Encrypted model on their dataset and generate the encrypted result. And third party receives the encrypted result, not the raw data.
- **Lesser Memory uses:** The size of plain and encrypted data is much smaller. In our example, it took 28 bytes for the plain data and 56 bytes for encrypted data.

Disadvantages:

- **More Computational time:** It took 209360000 ms to encrypt the model on the third-party side and another 77780000 ms to decrypt the user scores. On the user side, it took 201710000 ms to get the score from the encrypted model.
- **Prediction known by Third Party:** The user will generate the encrypted prediction from the model. To decrypt the prediction, it has to share the encrypted prediction with a third party. By this prediction will be known to the third party and it can learn something from the predicted result.

5.2 Send Encrypted Values for Prediction to Third Party

We model this use case using Microsoft SEAL library [1] The third-party trains the model on its own data. In our example, the model is a simple linear regression model on 2-dimensional data. The user then encrypts its data using homomorphic encryption. In our example, the encryption is made using the CKKS encryption model. After that, the user sends its encrypted data to the third party, which performs the prediction on the encrypted data and returns the encrypted results to the user, where the user decrypts them using their private key and obtains the actual results.

Advantages:

- **The overhead on the third party is minimal:** The third party is not affected by the encryption scheme, as the model is trained on its plain data, and prediction costs are similar to plain. In our example: it took **1.67 seconds** to train a logistic regression model of size 25^2.
- **Third party is not aware of the result:** Since the prediction is done on encrypted data, the third party cannot guess what the prediction is. The user's data stays private, assuming best practices are enforced.

Disadvantages:

- **User incurs overhead** of encryption and decryption of the data used, although this can't be circumvented as the user requires privacy. It took 100 ms to generate the keys, another 100 ms to encrypt the data, and 73.6 ms to decrypt the results sent from the third party.
- **User trust third party:** The user must trust that the third party performs the right prediction/use the correct model on the data.
- **Heavier network cost** as the size of encrypted data is greater than that of encrypted data in homomorphic encryption. The input vector size is 2048 bytes, whereas the resulting encrypted file is 410561 bytes, **a x200 increase** in the size of transferred data.

5.3 Create machine learning model on encrypted data

In this use case, the user wants to train a machine learning model on their data, but they do not want to share the data with the third party who will be training the model. The user can use homomorphic encryption to encrypt their data before sending it to the third party. The third party can then train the model on the encrypted data without being able to see the underlying data. Once the model is trained, the third party can send the encrypted model back to the user. The user can then decrypt the model and use it to make predictions on their own data.

Advantages:

- **Encrypted user data:** The user data is encrypted and then sent to the third party. This allows the user to keep their data private while still being able to perform predictions using the model

Disadvantages:

- **Expensive Computational time:** Since training the model is done on the encrypted data and predictions performed are encrypted, it needs more amount of computational time. Training on plain data took 0.01359 sec, and on encrypted data took 519.3278 sec (x51900).

6 Conclusion

6.1 Libraries

We conducted a comparative study on three different homomorphic encryption libraries to assess their performance based on time and memory usage. Our findings revealed that the SEAL library with BFV encryption took less time for key generation, encryption, decryption, addition and multiplication operations, but consumed more memory space for keys and encrypted values. In contrast, the Python Pallier library had moderate performance in terms of time and memory, while the Java-DGK was more time-consuming but used less memory.

6.2 Machine learning usecases

To further evaluate the performance of homomorphic encryption, we considered three different machine learning use cases. In the first use case, where the user sends encrypted values to the third party for prediction, we observed that the third party incurred no overhead for privacy, except for the heavier network traffic, where the user incurred the whole overhead of securing its own data, as it took 200 times more memory for the size of the encrypted data. The second use case, where the third party sends the encrypted model to the user, threw the overhead of privacy $(e.g. encrypting the model)$ on the third party. This allowed the user to refrain from sending its data, encrypted or not, on the network. However, there exists information leakage as the third party needs to decrypt the predictions, giving information about the data. Finally, the third use case, where the model is trained on encrypted data, was the most time-consuming, but it had the advantage of being trained on the user's data, resulting in more accurate predictions.

References

[1] `https://github.com/microsoft/SEAL`

[2] `https://github.com/AndrewQuijano/Homomorphic_Encryption`

[3] `https://github.com/data61/python-paillier`

[4] https://python-paillier.readthedocs.io/en/develop/index.html

[5] https://github.com/OpenMined/TenSEAL/tree/main/tutorials

[6] Dan Boneh, Amit Sahai, Brent Waters Functional Encryption: Definitions and Challenges *Theory of Cryptography - 8th Theory of Cryptography Conference*, 2011.