Analysis and strategy development for improving the performance of the Paillier cryptosystem algorithm in medical data processing

Volodymyr Semchyshyn [a] , Dmytro Mykhalyk [a]

[1] *Ternopil Ivan Puluj National Technical University 1, Ruska str, 56, Ternopil, 46001, Ukraine*

Abstract

The Paillier cryptosystem, known for its additive homomorphic properties, plays a crucial role in preserving the confidentiality of computations with encrypted medical data. However, its computational complexity poses significant challenges when applied to large-scale medical datasets, affecting both performance and efficiency. This study focuses on analyzing the performance limitations of the Paillier cryptosystem and developing strategies to improve its performance for medical data processing.

This study provides a comprehensive framework for improving the performance of the Paillier cryptosystem, contributing to its effectiveness in the secure processing of medical data and paving the way for future advances in privacy-preserving cryptographic techniques.

Keywords

Paillier Cryptosystem, Medical Data Processing, Data Confidentiality, Cloud Computing, Encryption Algorithms

1. Introduction

The appearance of digital health technologies has transformed the management of medical records, making it possible to have more efficient storage, retrieval and analysis of patient information. The rise in the use of electronic health records (EHRs) and telemedicine platforms poses a significant threat in medical data privacy and integrity. Protection involves encrypting sensitive health information in order to prevent unauthorized access or breaches.

Out of diverse cryptographic techniques, Paillier cryptosystem distinguishes itself by its additive homomorphic properties. In other words, computations can be done on encrypted data without decrypting it first. This is especially useful in medical data processing where privacy-preserving computations may make tasks such as statistical analysis and data mining easier while keeping the underlying data secure[1].

[1] *ITTAP'2024: 4th International Workshop on Information Technologies: Theoretical and Applied Problems, October 23-25, 2024, Ternopil, Ukraine, Opole, Poland*

* Corresponding author.

† These authors contributed equally.

vmsemchyshyn@gmail.com (V. Semchyshyn); dmykhalyk@gnail.com (D. Mykhalyk)

0009-0008-9206-8657 (V. Semchyshyn); 0000-0001-9032-695X (D. Mykhalyk);

While traditional encryption algorithms like RSA and AES operate efficiently in standard scenarios, advanced algorithms that support operations on encrypted data, such as Paillier's homomorphic encryption, can benefit from the high-performance computing techniques. Implementing these techniques could lead to more efficient encryption and decryption processes, especially for large datasets or in cloud-based environments.

By applying high-performance computing strategies, we can address some of the limitations associated with the computational complexity of advanced encryption algorithms. This approach not only enhances the practicality of such algorithms but also opens up new possibilities for their application in secure data processing[2].

Review of encryption algorithms

There are several alternative encryption algorithms that can be used instead of or in addition to the Paillier algorithm, depending on your security and performance needs. Here are some of them:

1. RSA (Rivest-Shamir-Adleman)

Type: Asymmetric encryption

Features: RSA is one of the most common algorithms for asymmetric encryption and digital signatures. It is based on the complexity of factorization of large numbers.

Advantages: Widely supported, used for secure key exchange.

Disadvantages: Slower compared to symmetric algorithms; large key sizes for a high level of security.

2. ElGamal

Type: Asymmetric encryption

Features: Based on the complexity of calculating the discrete logarithm in finite fields. It provides privacy and is used to create digital signatures.

Benefits: Provides privacy and authentication.

Disadvantages: Requires larger ciphertext size than other algorithms.

3. ECC (Elliptic Curve Cryptography)

Type: Asymmetric encryption

Features: Uses properties of elliptic curves to provide security with smaller key sizes. This makes ECC particularly effective for resource-intensive environments.

Advantages: High level of security with smaller key sizes.

Disadvantages: More complex implementation compared to other algorithms.

4. AES (Advanced Encryption Standard)

Type: Symmetric encryption

Features: Encryption standard to protect confidential data. Uses keys of 128, 192, or 256 bits.

Advantages: Very fast and efficient encryption for large amounts of data.

Disadvantages: Requires secure key exchange, as one key is used for encryption and decryption.

5. Homomorphic Encryption

Type: Specialized encryption

Features: Allows you to perform calculations on encrypted data without the need to decrypt it. Examples include Gentry and CKKS (Cheon-Kim-Kim-Song) schemes.

Advantages: Ideal for cloud computing where sensitive data needs to be processed.

Disadvantages: Very high computational complexity and slow processing speed[3].

The Paillier algorithm is a cryptographic algorithm that provides homomorphic encryption. It allows you to perform certain calculations on encrypted data without decrypting it, making it useful for secure computing and data storage.

2. Description of the Paillier algorithm

The main components of the Paillier algorithm

Public key: (n, g), where n is the product of two large prime numbers p and q, and g is a number that satisfies certain conditions.

Private key: (λ, μ), where λ is the Least Common Multiple (LCM) of (p-1) and (q-1), and μ is the modular inverse of n for λ.

The main stages of the algorithm

Key generation:

1) Choose two large prime numbers p and q.
2) Calculate n = p * q and n^2 = n * n.
3) Define g, often g = n + 1.
4) Calculate λ = LCM(p-1, q-1).
5) Calculate μ = λ^(-1) mod n.

Encryption:

1) Choose a random number r from {1, ..., n-1}.
2) Compute c1 = g^m mod n^2, where m is the plaintext.
3) Calculate c2 = r^n mod n^2.
4) Encrypted message c = c1 * c2 mod n^2.

Decryption:

1) Calculate x1 = c^λ mod n^2 - 1.
2) Calculate x2 = x1 / n mod n.
3) Decrypted message m = x2 * μ mod n.

3. Advantages and disadvantages of the Paillier algorithm

Advantages:

1) Homomorphic properties and data analysis. Paillier allows you to perform addition operations on encrypted data, which is particularly useful for statistical analyzes and calculations on encrypted data. Calculations can be performed without data decryption, which preserves data confidentiality during processing.

2) Security level: Based on the complexity of solving the problem of factorization of large numbers. Keys can be large for increased security.

3) Privacy:

Can provide a high level of privacy for data by storing data in encrypted form during processing.

4) Flexibility:

Application: Suitable for scenarios where you need to perform analytics on encrypted data without decrypting it.

Disadvantages:

1) Productivity:

Encryption and decryption with Paillier is significantly slower than symmetric encryption algorithms such as AES. This can be a problem when working with large amounts of data. High computational cost for encryption and decryption, which can require significant resources.

2) Storage capacity:

The size of the encrypted data is often much larger than the size of the original data. This can lead to high storage costs.

3) Difficulty of implementation:

The implementation of homomorphic algorithms can be more complex compared to symmetric algorithms, requiring additional knowledge and skills for proper implementation.

4) Scaling:

The efficiency of the algorithm may decrease when scaling to large volumes of data due to computational costs and overhead.

The Paillier algorithm is a powerful tool for processing data with high privacy requirements, where computations must be performed on encrypted data. However, its performance and storage capacity can be a problem for large data sets or scenarios where speed is critical. For most applications that require fast encryption and decryption, symmetric algorithms such as AES may be more appropriate.

4. Comparison of Paillier Algorithm and AES

1) Encryption type:

Paillier: A homomorphic cipher that allows you to perform arithmetic operations on encrypted data. This can be useful in scenarios where data needs to be processed without decryption[3].

AES: A symmetric cipher that provides a high level of security for data but does not support operations on encrypted data.

2) Basic function:

Paillier: Allows to perform arithmetic operations on encrypted data.

AES: Encrypts data but does not allow to perform operations on encrypted data.

3) Key size:

Paillier: The size of the key is determined by the length of the prime numbers p and q, which are used to create the module n. larger keys provide a higher level of security but increase processing time.

AES: Supports standard key sizes of 128-bit, 192-bit and 256-bit, providing a good balance between speed and security.

4) Execution speed:

Paillier: Speed may be slower due to complex math operations, especially with large key sizes. Optimization and parallelization can help but can still be slower compared to AES[4].

AES: Extremely fast due to simple math operations and parallelization capabilities. Suitable for large volumes of data and high speeds.

5) Storage volume on AWS:

Paillier: Encrypted data can be larger due to encryption overhead. The volume may increase depending on the size of the key and data.

AES: Usually stores data more compactly, due to lower overhead. AES encryption does not increase data volume as much as it can with Paillier.

6) Resource requirements:

Paillier: High computational resources due to the complexity of mathematical operations.

AES: Low computing resources, fast encryption and decryption.

7) Parallelization:

Paillier: Can be parallelized, but the complexity of the math operations may limit efficiency[4,5].

AES: Parallelizes quickly and supports SIMD instructions, making it very efficient on modern processors.

8) Security:

Paillier: Provides a high level of security for scenarios where computations over encrypted data are required. However, security may depend on implementation parameters and may be less effective for large amounts of data[4,5].

AES: High security with many proven attacks. A good choice for general data encryption.

Homomorphic encryption allows to perform calculations on encrypted data, which can be useful in certain scenarios, such as processing data without decrypting it. However, these algorithms typically have higher computational overhead and a larger volume of encrypted data, which can lead to increased cloud costs. The performance of algorithms such as Paillier is often lower due to the complexity of the mathematical calculations.

For general medical data storage where data processing without decryption is not a primary priority, AES provides an efficient balance between speed, security and resources.

If the system requires data processing without decryption, homomorphic algorithms can be considered as an additional solution but considering possible resource costs[6].

5. Implementation of the optimized Paillier algorithm

An encryption and decryption algorithm based on the Paillier homomorphic encryption scheme was implemented.

Input parameters for encryption (Fig 1):

textData: This is the data to be encrypted. In this case, they are represented as a big number (BigInteger).

publicKey: A public key that contains parameters for encryption (values of n, g, nSquared).

Encryption:

A random number r is generated, which is very important for ensuring the randomness of the ciphertext.

The Paillier algorithm uses several mathematical operations based on the values from the public key:

publicKey.g.modPow(textData, nSquared) is the exponentiation of g^textData mod nSquared.

r.modPow(n, nSquared) is the exponentiation of a random number r^n mod nSquared.

The results of these two operations are multiplied, and then the modulus of nSquared is taken.

This is the encrypted text.

Input parameters for decryption(Fig 1):

encryptedData: This is encrypted data that needs to be decrypted.

publicKey: The public key that contains the parameters for decryption.

privateKey: private key containing parameters λ (lambda) and μ (mu).

Decryption:

Calculates the value of u as encryptedData.modPow(privateKey.lambda, nSquared). This is the operation of raising the encrypted data to the power of λ, followed by taking the module by nSquared.

```
// Encryption
Rename usages
public static BigInteger encrypt(BigInteger textData, PublicKey publicKey) {  3 usages
    SecureRandom random = new SecureRandom();
    BigInteger n = publicKey.n;
    BigInteger nSquared = publicKey.nSquared;

    BigInteger r = new BigInteger(n.bitLength(), random).mod(n);

    return publicKey.g.modPow(textData, nSquared)
            .multiply(r.modPow(n, nSquared))
            .mod(nSquared);
}

// Decryption
public static BigInteger decrypt(BigInteger encryptedData, PublicKey publicKey, PrivateKey privateKey) {  2 usages
    BigInteger n = publicKey.n;
    BigInteger nSquared = publicKey.nSquared;

    BigInteger u = encryptedData.modPow(privateKey.lambda, nSquared);
    return L(u, n).multiply(privateKey.mu).mod(n);
}
```

Figure 1: Sequential encryption and decryption algorithm based on the Paillier homomorphic encryption

The function L is calculated as L(u, n) = (u - 1) / n.

The decrypted text is calculated by multiplying the result L(u, n) by μ from the private key, and then taking the modulus of n. It is used during decryption to correctly obtain the original text from the ciphertext.

Basic concepts of Paillier:

Public key: n, g, and nSquared (where nSquared = n^2).

Private key: λ and μ. They are used to decrypt the message.

Homomorphic: Paillier is a homomorphic encryption scheme, meaning that it allows operations on encrypted data (such as addition) without decrypting it[7].

An optimized algorithm was implemented to increase productivity

Input parameters (Fig 2):

textData: List of data (in BigInteger format) to be encrypted.

publicKey: public key for encryption using the Paillier algorithm.

Potential exceptions: The method may throw an InterruptedException or an ExecutionException if there are multithreading issues.

An ExecutorService is used, which creates a thread pool. The number of threads depends on the number of available processors in the system (Runtime.getRuntime().availableProcessors()), which allows for the most efficient use of system resources.

For each element in the textData list, a separate encryption task is created, which is started in a new thread using executor.submit(). EncryptTask is a class that implements the encryption of each individual item (this class is probably a separate implementation where the encryption process is described). The submit method returns a Future object that represents the result of an asynchronous operation (in this case, encryption). All these Future objects are stored in the futures list.

```
public static List<BigInteger> parallelEncrypt(List<BigInteger> textData, Paillier.PublicKey publicKey) throws InterruptedException, ExecutionException {
    long startTime = System.nanoTime();

    ExecutorService executor = Executors.newFixedThreadPool(Runtime.getRuntime().availableProcessors());
    List<Future<BigInteger>> futures = new ArrayList<>();

    //Distribution of encryption tasks between threads
    for (BigInteger plaintext : textData) {
        Future<BigInteger> future = executor.submit(new EncryptTask(plaintext, publicKey));
        futures.add(future);
    }

    List<BigInteger> ciphertexts = new ArrayList<>();
    for (Future<BigInteger> future : futures) {
        ciphertexts.add(future.get());
    }

    executor.shutdown();
    long endTime = System.nanoTime();
    System.out.println("Encrypt Execution time in nano sec: " + (endTime - startTime)+ " ns");
    return ciphertexts;
}
```

Figure 2: Parallel algorithm for encryption

Input parameters(Fig 3):

encryptedText: list of encrypted data (in BigInteger format) to be decrypted.

publicKey: The public key used in the Paillier algorithm.

privateKey: private key for decrypting messages.

Potential exceptions: The method may throw an InterruptedException or an ExecutionException if there are multithreading issues.

```
public static List<BigInteger> parallelDecrypt(List<BigInteger> encryptedText, Paillier.PublicKey publicKey, Paillier.PrivateKey privateKey) throws
    long startTime = System.nanoTime();
    ExecutorService executor = Executors.newFixedThreadPool(Runtime.getRuntime().availableProcessors());
    List<Future<BigInteger>> futures = new ArrayList<>();

    // Distribution of decryption tasks between threads
    for (BigInteger ciphertext : encryptedText) {
        Future<BigInteger> future = executor.submit(new DecryptTask(ciphertext, publicKey, privateKey));
        futures.add(future);
    }

    List<BigInteger> textData = new ArrayList<>();
    for (Future<BigInteger> future : futures) {
        textData.add(future.get());
    }

    executor.shutdown();
    long endTime = System.nanoTime();
    System.out.println("Decrypt Execution time in nano sec: " + (endTime - startTime)+ " ns");
    return textData;
}
```

Figure 3: Parallel algorithm for decryption

An ExecutorService is created that manages the thread pool. The number of threads is determined by the number of available processors (Runtime.getRuntime().availableProcessors()). This allows efficient use of system hardware resources for parallel execution of decryption. For each item in the encryptedText list (encrypted data), a separate decryption task is created, which is passed to the execution of a new thread via executor.submit(). The DecryptTask class performs the decryption operation for each

encrypted block. The submit method returns a Future object that represents the result of asynchronous execution. All these Future objects are added to the futures list.

To check algorithm was created test data(Fig 4).

According to results encryption and decryption is correct. However, performance for small data set is better for sequential realization(Fig 5).

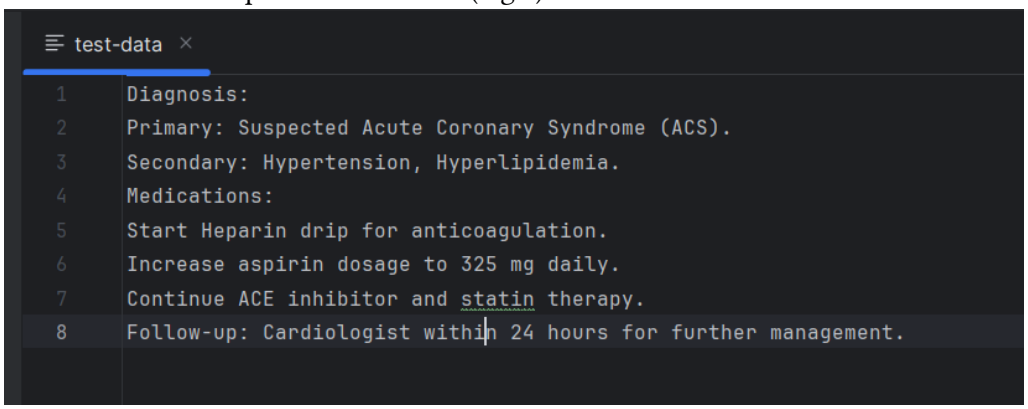

Figure 4: Test data for encryption and decryption

Figure 5: Sequential and parallel results

Analysis of the results of the Paillier algorithm and the optimized Paillier algorithm

According to results of research for small amount of data (309 bytes), parallel methods are not very efficient due to the overhead of thread management. For 3.45 MB, parallel approaches show significant speed advantages for both encryption and decryption compared to sequential methods. For large data amount (50 MB), parallel methods provide significant processing speed advantages for both encryption and decryption. Parallel decryption is much faster than serial decryption, but it requires more powerful computing resources(Table 1).

Table 1

Analysis of the researched results

Data amount	Parallel encryption(ns)	Sequential encryption(ns)	Parallel decryption(ns)	Sequential decryption(ns)
309 bytes	17506900	4927600	5008500	663200
3.45 MB	8277496400	14101378500	3112705100	5291598670
50 MB	11638800000	20276400000	42220000000	7174400000
	0	0		0

Parallel methods are significantly more efficient for processing large amounts of data, both for encryption and for decryption. This is especially noticeable when processing data with a volume of 50 MB, where the advantages of parallelization are most pronounced. For small data amounts (309 bytes), the thread management overhead of parallel methods outweighs the benefits, making them less efficient than the sequential approach. As data amounts increase, the advantages of parallel methods become more apparent. Parallel encryption and decryption demonstrate a significant reduction in processing time for large amounts of data compared to sequential methods.

Conclusions

In this work analyzed the Paillier algorithm, which provides additive homomorphic properties for processing encrypted medical data. The main emphasis was placed on studying the performance of the algorithm and developing strategies for its optimization, which will allow more efficient processing of large volumes of data. The conducted analysis showed that the Paillier algorithm has significant advantages in preserving data confidentiality due to its homomorphic properties, which allow performing calculations on encrypted data without decrypting it. This is particularly useful for medical data where privacy protection is critical.

However, the high computational complexity of the algorithm creates performance problems, especially when processing large amounts of data. To improve processing speed, parallel encryption and decryption methods were implemented, which demonstrated significant advantages in the processing speed of large data sets compared to sequential methods.

The results of the work confirm the expediency of using the Paillier algorithm for tasks that require data processing without decoding them, and suggest effective optimization strategies to improve the performance of this algorithm in the conditions of large medical data sets. In the future, this may contribute to the further development of cryptographic methods to ensure the confidentiality and security of medical data.

References

[1] P. Paillier, "Public-Key Cryptosystems Based on Composite Degree Residuosity Classes," in Advances in Cryptology — EUROCRYPT'99, Lecture Notes in Computer Science, vol. 1592, Springer, Berlin, Heidelberg, 1999, pp. 223-238, doi: 10.1007/3-540-48910-X_16.

[2] Petryk, M.R., Boyko, I.V., Khimich, O.M. et al. High-Performance Supercomputer Technologies of Simulation and Identification of Nanoporous Systems with Feedback for n-Component Competitive Adsorption. Cybern Syst Anal 57, 316–328 (2021).

[3] V. Vaikuntanathan, "Computing Blindfolded: New Developments in Fully Homomorphic Encryption," in Proceedings of the 2011 IEEE 52nd Annual Symposium on Foundations of Computer Science (FOCS '11), Palm Springs, CA, USA, 2011, pp. 5-16, doi: 10.1109/FOCS.2011.45.

[4] X. Wu, H. Hu, and F. Bao, "Efficient Homomorphic Encryption Protocol for Scalable Encrypted Data Processing in Health Systems," IEEE Access, vol. 7, pp. 143746-143758, 2019, doi: 10.1109/ACCESS.2019.2945273.

[5] M. Kim, K. Lauter, and M. Naehrig, "Private Predictive Analysis on Encrypted Medical Data," Journal of Biomedical Informatics, vol. 46, no. 4, pp. 675-685, Aug. 2013, doi: 10.1016/j.jbi.2012.09.006.

[6] M. Naehrig, K. Lauter, and V. Vaikuntanathan, "Can Homomorphic Encryption be Practical?" in Proceedings of the 3rd ACM Workshop on Cloud Computing Security Workshop (CCSW '11), Chicago, IL, USA, 2011, pp. 113-124, doi: 10.1145/2046660.2046682.

[7] X. Yi, R. Paulet, and E. Bertino, Homomorphic Encryption, SpringerBriefs in Computer Science, Springer, New York, NY, USA, 2014, doi: 10.1007/978-1-4614-7411-2.