Privacy Through Homomorphic Encryption

Benjamim Moreira 2020261856
José Cunha 2021223719
José Filipe 2021216675

17th December 2024

1 Introduction

Homomorphic encryption emerges as an advanced solution for safeguarding sensitive data, enabling computations to be performed directly on encrypted data without requiring decryption or exposing keys. This approach ensures privacy and security while supporting applications in critical fields such as healthcare and private data analytics [6, 2].

In this project, we explore the use of homomorphic encryption in the healthcare domain, focusing on heart rate data analysis. Monitoring heart rate is crucial for assessing cardiovascular conditions, medical diagnosis, and identifying health patterns. However, the sensitive nature of this data makes its protection paramount in scenarios involving sharing and processing by third parties [11].

The problem we are replicating involves the interaction between two entities: the data holder and the data analyzer. For example, data collected by smartwatches can be provided to a clinic for statistical analysis on encrypted data, ensuring sensitive information remains private throughout the entire process [7]. The data holder encrypts the data before sharing it. The data analyzer performs calculations directly on the encrypted data and returns the results to the holder, who decrypts them to obtain the final output [8].

To implement this system, we leverage the Paillier encryption scheme, a partially additive homomorphic encryption method. This scheme allows the summation of encrypted messages and multiplication by constants while preserving data privacy [8]. Additionally, we evaluate its performance by analyzing the impact of key size on critical operations such as encryption, decryption, and homomorphic addition [9].

This work aims not only to validate the effectiveness of homomorphic encryption in real-world scenarios but also to highlight the opportunities and challenges in its practical deployment [2, 1].

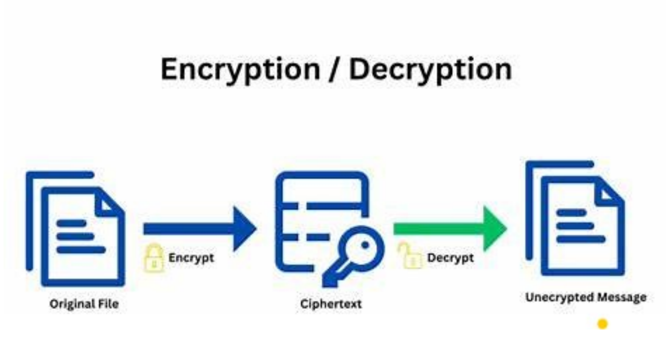


Figure 1: Project Workflow

2 Background

Homomorphism refers to a mapping (a function, i.e., an operation) that preserves algebraic structures (e.g., groups, rings, and vector spaces) between the domain and range of an algebraic set. Homomorphic encryption exhibits similar properties, enabling certain operations to be performed directly on encrypted data without requiring decryption. This property aligns with the mathematical definition of homomorphism, as shown in Equation 1.

$$E(m_1) * E(m_2) = E(m_1 * m_2), \quad \forall m_1, m_2 \in M \tag{1}$$

In this context, E represents an encryption function, M denotes the set of all plaintext messages, and $*$ is an operation supported by the encryption scheme. For instance, given two messages m_1 and m_2, their encrypted sum $E(m_1 + m_2)$ can be computed using $E(m_1)$ and $E(m_2)$ without knowledge of the original plaintexts.

The degree and complexity of a homomorphic encryption scheme depend on the number and types of supported operations. These schemes are broadly classified into the following categories [6, 8, 3].

2.1 Partially Homomorphic Encryption (PHE)

Partially Homomorphic Encryption allows an infinite number of a specific type of operation, such as addition or multiplication. For example, the Paillier cryptosystem supports addition by ensuring that the sum of two ciphertexts equals the encryption of the sum of the corresponding plaintexts. Multiplicative homomorphism can be observed in schemes such as RSA, where the product of two ciphertexts is equivalent to the encryption of the product of their plaintexts [8, 10].

2.2 Somewhat Homomorphic Encryption (SHE)

Somewhat Homomorphic Encryption extends PHE by allowing a limited number of operations, such as a combination of additions and multiplications. While the finite nature of supported operations may seem restrictive, it represents a significant improvement over PHE and demonstrates the increasing difficulty in designing homomorphic cryptographic techniques [6].

2.3 Fully Homomorphic Encryption (FHE)

Fully Homomorphic Encryption supports an unlimited number of both additions and multiplications on ciphertexts, making it the most flexible and powerful type of homomorphic encryption. However, this comes at a significant computational cost, resulting in high implementation complexity and slower operational performance [6].

2.4 Algorithms Under Consideration

This study focuses on three additive, partially homomorphic encryption algorithms: *Paillier*, *Damgård-Jurik*, and *ElGamal*. These algorithms differ in their cryptographic foundations and performance characteristics:

2.4.1 Paillier Cryptosystem

The Paillier cryptosystem is additively homomorphic, supporting ciphertext addition and plaintext multiplication. Its simplicity and robust security make it a popular choice for applications such as electronic voting [8].

Key Generation:

1. Select two large prime numbers p and q such that $\gcd(p \cdot q, (p-1)(q-1)) = 1$.
2. Compute $n = p \cdot q$ and $\lambda = \mathrm{lcm}(p-1, q-1)$.
3. Choose a random $g \in Z_{n^2}^*$.
4. Compute $\mu = (L(g^\lambda \mod n^2))^{-1} \mod n$, where $L(x) = \frac{x-1}{n}$.
5. The public key is (n, g), and the private key is (λ, μ).

Encryption: To encrypt a message $m \in \{0, 1, \ldots, n-1\}$, choose a random $r \in Z_n^*$ and compute:

$$E(m, r) = g^m \cdot r^n \mod n^2 \tag{2}$$

Decryption: Given ciphertext c, compute:

$$m = L(c^\lambda \mod n^2) \cdot \mu \mod n \tag{3}$$

The encryption function $E(m, r)$ has the additively homomorphic property:

$$E(m_1, r_1) \cdot E(m_2, r_2) = E(m_1 + m_2, r_1 \cdot r_2) \tag{4}$$

2.4.2 Damgård-Jurik Cryptosystem

The Damgård-Jurik cryptosystem generalizes Paillier by supporting higher-order arithmetic operations and reducing ciphertext expansion [4].

Key Generation: Similar to Paillier, but with n replaced by n^s, where $s \geq 1$ is a parameter. Select $g \in Z^*_{n^{s+1}}$ such that $g = (1+n)^j \cdot x \mod n^{s+1}$, where j is coprime to n.

Encryption: For a message $m \in Z_{n^s}$, choose $r \in Z^*_n$ and compute:

$$E(m, r) = g^m \cdot r^{n^s} \mod n^{s+1} \tag{5}$$

Decryption: Given ciphertext c, use a recursive version of Paillier decryption to compute:

$$m = (j \cdot m \cdot d^{-1}) \mod n^s \tag{6}$$

where d is derived using the Chinese Remainder Theorem.

2.4.3 ElGamal Cryptosystem

The ElGamal cryptosystem, based on the Diffie-Hellman problem, supports both addition and multiplication under certain modifications (e.g., Exponential ElGamal) [5].

Key Generation:

1. Select a large prime p and a primitive root α modulo p.
2. Choose a random private key $x \in \{1, \ldots, p-2\}$.
3. Compute the public key $\beta = \alpha^x \mod p$.
4. The public key is (p, α, β), and the private key is x.

Encryption: To encrypt a message $m < p$, choose a random integer k and compute:

$$c_1 = \alpha^k \mod p \tag{7}$$

$$c_2 = m \cdot \beta^k \mod p \tag{8}$$

The ciphertext is (c_1, c_2).

Decryption: To decrypt (c_1, c_2), compute:

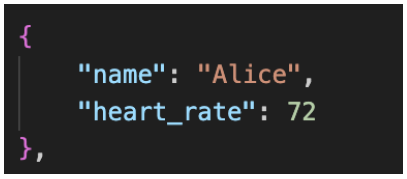
$$m = c_2 \cdot c_1^{-x} \mod p \tag{9}$$

—

Each algorithm exhibits unique trade-offs in terms of performance and cryptographic strength. This study evaluates their encryption and homomorphic operation performance under varying key lengths.

3 Experiments

The experiments will be divided into three parts, in the first part we will analyse the encryption performance of three partially homomorphic additive encryption schemes-Paillier, Damgård-Jurik, and ElGamal. The second part analyses the temporal performance of PaIllier for encryption, decryption and homomorphic addiction. In the third part, we'll analyse in detail the statistical analyses that PaIllier can perform. All experiments will be conducted using a synthetic dataset of 25 bpm samples in JSON format.In Figure 2 is an example of one of the dataset elements:



```
{
    "name": "Alice",
    "heart_rate": 72
},
```

Figure 2: Example of one of the dataset elements

3.1 Encryption performance of three partially homomorphic additive encryption

In this section, we evaluate the encryption performance of three additive partially homomorphic encryption schemes—*Paillier*, *Damgård-Jurik*, and *ElGamal*. The evaluation focuses on the time required to encrypt a set of plaintext data under varying key lengths: 512, 1024, and 2048 bits.

3.1.1 Performance Evaluation

Figure 3 illustrates the encryption times for each algorithm across different key lengths. As the key size increases, encryption times also increase due to the computational overhead of modular arithmetic operations.

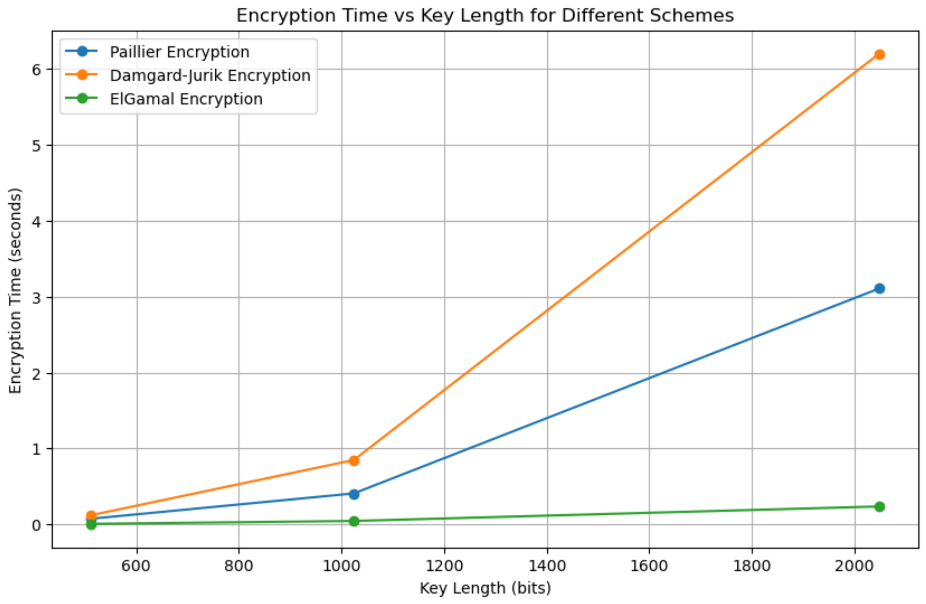


Figure 3: Encryption Time vs Key Length for Paillier, Damgård-Jurik, and ElGamal encryption schemes.

- **Paillier Encryption**: The encryption time increases linearly with key size, reflecting its reliance on modular exponentiation in $Z^*_{n^2}$. At 2048-bit keys, encryption takes approximately 3.1 seconds.
- **Damgård-Jurik Encryption**: This scheme exhibits the steepest growth in encryption time due to the higher complexity introduced by the n^{s+1}-modulus. At 2048-bit keys, encryption requires over 6 seconds.

- **ElGamal Encryption**: ElGamal demonstrates the smallest encryption times, maintaining near-constant performance relative to the other schemes. The increase in key size results in a relatively modest overhead compared to Paillier and Damgård-Jurik.

3.1.2 Discussion

The results highlight a clear trade-off between encryption efficiency and cryptographic flexibility:

1. **Paillier** offers balanced performance with strong additive homomorphism.
2. **Damgård-Jurik** provides enhanced arithmetic capabilities but at a significantly higher encryption cost.
3. **ElGamal** excels in speed but requires modifications to achieve additive operations (e.g., Exponential ElGamal).

The figure demonstrates that for larger key sizes, encryption with Damgård-Jurik becomes computationally expensive, whereas ElGamal remains efficient even with higher key lengths. This behavior can guide the selection of encryption schemes based on application-specific performance requirements.

3.2 Temporal performance of PaIllier for encryption, decryption and homomorphic addiction

The performance is analyzed in terms of encryption time, decryption time, and homomorphic addition time for varying key sizes: 512, 1024, and 2048 bits.

3.2.1 Experimental Setup

The `LightPHE` library was employed to implement the Paillier encryption scheme. The experimental procedure is as follows:

- **Encryption Time**: For each key size, all heart rate values were encrypted sequentially, and the total time for the encryption process was measured.
- **Homomorphic Addition Time**: Using the encrypted heart rate values, the homomorphic addition operation was performed to compute the total encrypted sum. The time taken for this addition was recorded.
- **Decryption Time**: The resulting sum from the homomorphic addition was decrypted, and the decryption time was measured.

Correctness was ensured by verifying that the decrypted sum matched the sum of the original heart rate values.

3.2.2 Performance Evaluation

The results of the experiment are presented in Figure 4, which shows the encryption, decryption, and homomorphic addition times as a function of the key length.

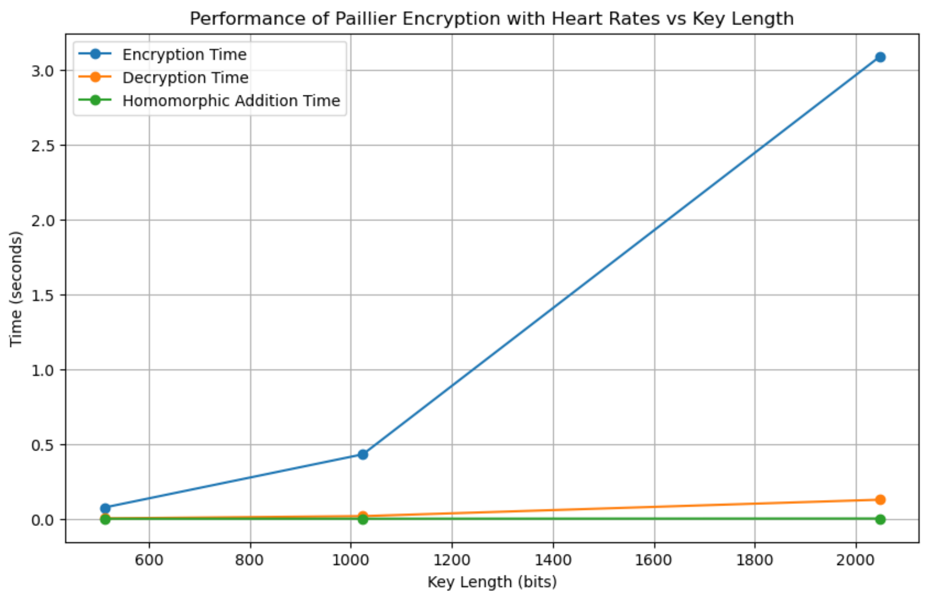


Figure 4: Performance of Paillier Encryption with Heart Rates vs Key Length.

- **Encryption Time**: The encryption time increases significantly with the key size, reflecting the computational cost of modular arithmetic operations in the Paillier encryption scheme. At **2048**-bit keys, the encryption time is approximately **3.1** seconds.
- **Homomorphic Addition Time**: The time required for the homomorphic addition operation remains relatively constant across all key sizes. This behavior is expected because the addition operation involves a constant number of modular multiplications, independent of the key size.
- **Decryption Time**: The decryption time increases slightly with larger key sizes. However, the growth is relatively moderate compared to the encryption time.

3.2.3 Discussion

The results highlight the impact of increasing key lengths on the computational efficiency of the Paillier encryption scheme:

1. **Encryption Overhead**: The encryption process is the most computationally expensive operation, particularly at larger key sizes, due to the reliance on modular exponentiation.
2. **Efficiency of Homomorphic Addition**: Homomorphic addition is computationally efficient and exhibits minimal overhead, making it suitable for applications requiring frequent additive operations on encrypted data.
3. **Decryption Overhead**: Although the decryption time increases with key size, it remains manageable compared to the encryption time.

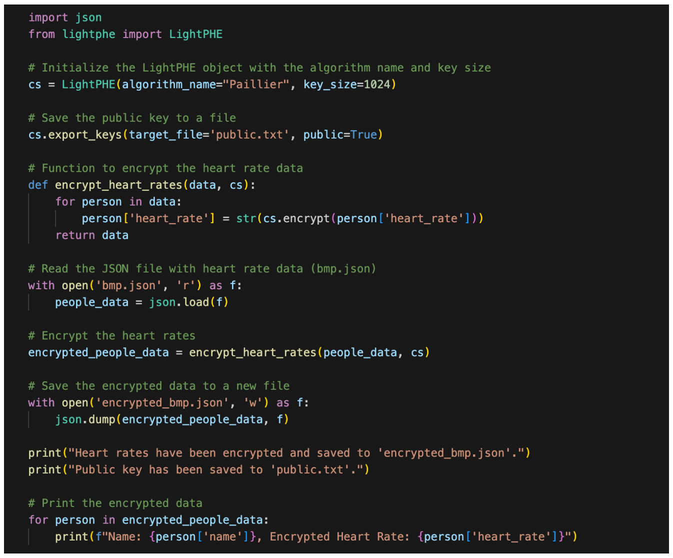
Overall, the Paillier encryption scheme demonstrates its suitability for secure additive operations, with key size serving as a trade-off between security and computational efficiency.

3.3 Statistical analyses using Paillier

This experiment aims to demonstrate the application of Paillier homomorphic encryption for securely storing and processing heart rate data. The process involved encrypting heart rates on the data holder, performing homomorphic operations in the data analyser, and only decrypting the results on the data holder side side. Below is a breakdown of the experiment and its outcomes.

3.3.1 Data Holder- Encryption and Data Processing

Initially in the Data Holder part, heart rate data for 25 individuals was read from a JSON file. Each individual's heart rate was then encrypted using the Paillier encryption scheme with a 1024-bit key. The resulting encrypted heart rates were saved in a new JSON file (`encrypted_bmp.json`), which was subsequently read by the Data Analyser for further processing. Additionally, the public key used for encryption was exported and stored separately in a file (`public.txt`). This approach ensures that the Data Analyser can perform operations on the encrypted data while maintaining the confidentiality of the original heart rate information, as illustrated in Figure 5.

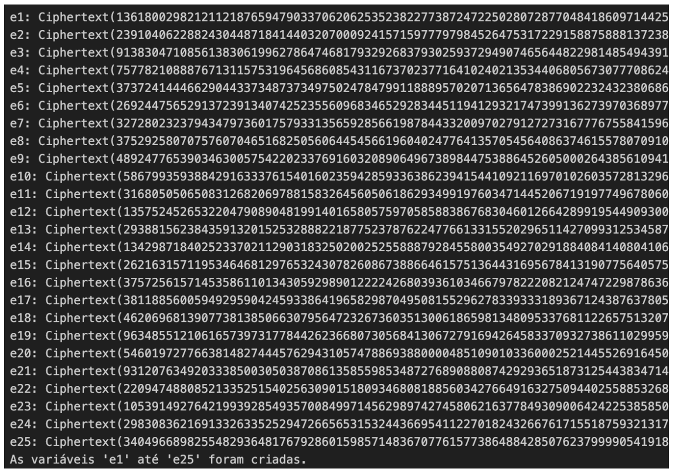


```python
import json
from lightphe import LightPHE

# Initialize the LightPHE object with the algorithm name and key size
cs = LightPHE(algorithm_name="Paillier", key_size=1024)

# Save the public key to a file
cs.export_keys(target_file='public.txt', public=True)

# Function to encrypt the heart rate data
def encrypt_heart_rates(data, cs):
    for person in data:
        person['heart_rate'] = str(cs.encrypt(person['heart_rate']))
    return data

# Read the JSON file with heart rate data (bmp.json)
with open('bmp.json', 'r') as f:
    people_data = json.load(f)

# Encrypt the heart rates
encrypted_people_data = encrypt_heart_rates(people_data, cs)

# Save the encrypted data to a new file
with open('encrypted_bmp.json', 'w') as f:
    json.dump(encrypted_people_data, f)

print("Heart rates have been encrypted and saved to 'encrypted_bmp.json'.")
print("Public key has been saved to 'public.txt'.")

# Print the encrypted data
for person in encrypted_people_data:
    print(f"Name: {person['name']}, Encrypted Heart Rate: {person['heart_rate']}")
```

Figure 5: Data Holder- Encryption and Data Processing

3.3.2 Data Analyser- Homomorphic Operations

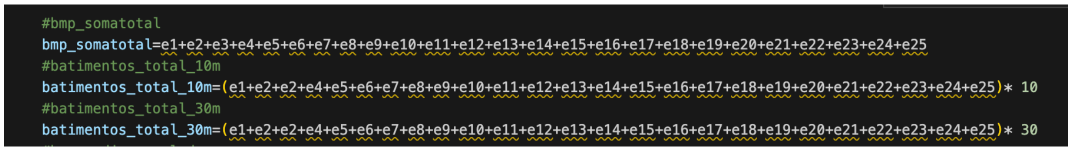
Once the encrypted heart rates were received by the Data Analyser, the encrypted values were extracted from the file `encrypted_bmp.json`. Variables were then created for each sample, named sequentially as `e1` through `e25`, as shown in Figure 6.



```
e1: Ciphertext(1361800298212112187659479033706206253523822773872472250280728770484186097144 25
e2: Ciphertext(239104062288243044871841440320700092415715977797984526475317229158875888137238
e3: Ciphertext(913830471085613830619962786474681793292683793025937294907465644822981485494391
e4: Ciphertext(757782108887671311575319645686085431167370237716410240213534406805673077708624
e5: Ciphertext(373724144466290443373487373497502478479911888957020713656478386902232432380686
e6: Ciphertext(269244756529137239134074252355609683465292834451194129321747399136273970368977
e7: Ciphertext(327280232379434797360175793313565928566198784433200970279127273167776755841596
e8: Ciphertext(375292580707576070465168250560644545661960402477641357054564086374615578070910
e9: Ciphertext(489247765390346300575422023376916032089064967389844753886452605000264385610941
e10: Ciphertext(586799359388429163337615401602359428593363862394154410921169701026035728 13296
e11: Ciphertext(316805050650831268206978815832645605061862934991976034714452067191977496780 60
e12: Ciphertext(135752452653220479089048199140165805759705858838676830460126642899195449093 00
e13: Ciphertext(293881562384359132015253288822187752378762247766133155202965114270993125345 87
e14: Ciphertext(134298718402523370211290318325020025255888792845580035492702918840841408041 06
e15: Ciphertext(262163157119534646812976532430782608673886646157513644316956784131907756405 75
e16: Ciphertext(375725615714535861101343059298901222242680393610346679782220821247472298786 36
e17: Ciphertext(381188560059492959042459338641965829870495081552962783393331893671243876378 05
e18: Ciphertext(462069681390773813850663079564723267360351300618659813480953376811226575132 07
e19: Ciphertext(963485512106165739731778442623668073056841306727916942645833709327386110299 59
e20: Ciphertext(546019727766381482744457629431057478869388000048510901033600025214455269164 50
e21: Ciphertext(931207634920333850030503870861358559853487276890880874292936518731254438347 14
e22: Ciphertext(220947488085213352515402563090151809346808188560342766491632750944025588532 68
e23: Ciphertext(105391492764219939285493570084997145629897427458062163778493090064242253858 50
e24: Ciphertext(298308362169133263352529472665653153244366954112270182432667617155187593213 17
e25: Ciphertext(340496689825548293648176792860159857148367077615773864884285076237999905419 18
As variáveis 'e1' até 'e25' foram criadas.
```

Figure 6: Data analyser reading encrypted files

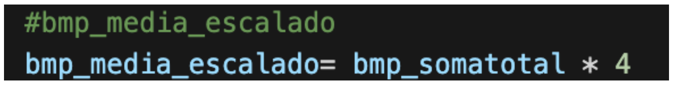
Them a set of homomorphic operations was performed. As depicted in Figure 7, the sum of heart rates, as well as the total heart rates over 10 and 30 minutes, were computed using homomorphic addition. These operations were performed directly on the ciphertexts, ensuring that the data remained secure during computation.



```
#bmp_somatotal
bmp_somatotal=e1+e2+e3+e4+e5+e6+e7+e8+e9+e10+e11+e12+e13+e14+e15+e16+e17+e18+e19+e20+e21+e22+e23+e24+e25
#batimentos_total_10m
batimentos_total_10m=(e1+e2+e2+e4+e5+e6+e7+e8+e9+e10+e11+e12+e13+e14+e15+e16+e17+e18+e19+e20+e21+e22+e23+e24+e25)* 10
#batimentos_total_30m
batimentos_total_30m=(e1+e2+e2+e4+e5+e6+e7+e8+e9+e10+e11+e12+e13+e14+e15+e16+e17+e18+e19+e20+e21+e22+e23+e24+e25)* 30
```

Figure 7: Paullier's Homomorphic Operations (sum)

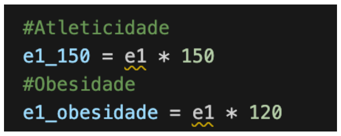
Due to Paullier's limitations, in order to calculate the average bmp value and the metrics to check whether it is an athlete or not, leading to a need to work with scaled values. In Figure 8, the average bpm was calculated by the total sum of heart rates multiplied by 4 (i.e. 0.04 scaled by 100).



```
#bmp_media_escalado
bmp_media_escalado= bmp_somatotal * 4
```

Figure 8: Paullier's Homomorphic operations (multiplication) calculating average bmp

To check if the individual are an athlete, is multiplied the individual heart rate value by 150 (i.e. 1.5 scaled by 100) and if this value was lower than 100bpm, the individual was considered an athlete. To check for obesity, we multiplied the individual heart rate value by 120 (i.e. 1.2 scaled by 100) and if this value is higher or equal than 100bpm, the individual is considered obese, and if it exceeds 120bpm, morbidly obese as shown in Figure 9.e



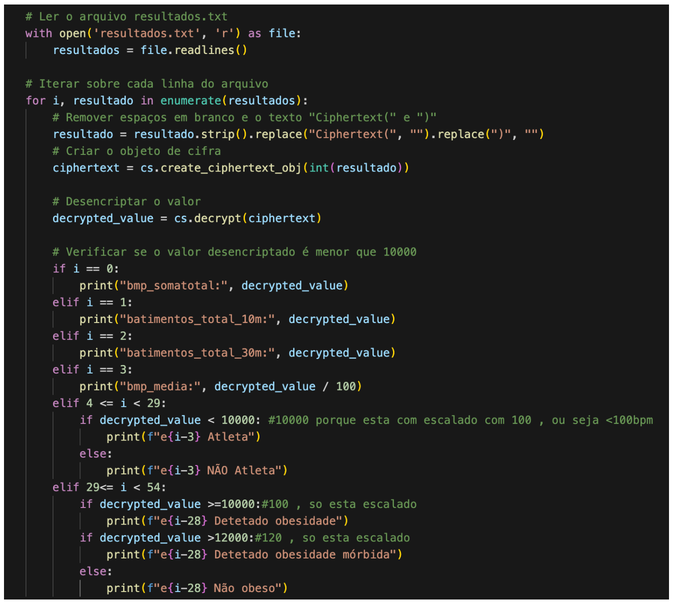
```
#Atleticidade
e1_150 = e1 * 150
#Obesidade
e1_obesidade = e1 * 120
```

Figure 9: Paullier's Homomorphic operations (multiplication) calculating fitness metrics for sample 1

To complete the Data Analyzer's work, all these calculated values were written to a txt file (`resultados.txt`) to be read and decrypted by the data holder who has access to the private key.

3.3.3 Data Holder- Decryption and analyse results

After statistical analysis by the data analyser, the data holder receives the encrypted data in the (`resultados.txt`) file and decrypts it using the following function:



```python
# Ler o arquivo resultados.txt
with open('resultados.txt', 'r') as file:
    resultados = file.readlines()

# Iterar sobre cada linha do arquivo
for i, resultado in enumerate(resultados):
    # Remover espaços em branco e o texto "Ciphertext(" e ")"
    resultado = resultado.strip().replace("Ciphertext(", "").replace(")", "")
    # Criar o objeto de cifra
    ciphertext = cs.create_ciphertext_obj(int(resultado))

    # Desencriptar o valor
    decrypted_value = cs.decrypt(ciphertext)

    # Verificar se o valor desencriptado é menor que 10000
    if i == 0:
        print("bmp_somatotal:", decrypted_value)
    elif i == 1:
        print("batimentos_total_10m:", decrypted_value)
    elif i == 2:
        print("batimentos_total_30m:", decrypted_value)
    elif i == 3:
        print("bmp_media:", decrypted_value / 100)
    elif 4 <= i < 29:
        if decrypted_value < 10000: #10000 porque esta com escalado com 100 , ou seja <100bpm
            print(f"e{i-3} Atleta")
        else:
            print(f"e{i-3} NÃO Atleta")
    elif 29<= i < 54:
        if decrypted_value >=10000:#100 , so esta escalado
            print(f"e{i-28} Detetado obesidade")
        if decrypted_value >12000:#120 , so esta escalado
            print(f"e{i-28} Detetado obesidade mórbida")
        else:
            print(f"e{i-28} Não obeso")
```

Figure 10: Data Holder -Decryption function

The following outcomes were observed:

- **Total Heart Rate (bmp_somatotal)**: The total sum of encrypted heart rates across all individuals was calculated as 1793.
- **Heart Rates for 10 and 30 Minutes (batimentos_total_10m and batimentos_total_30m)**: The total heart rate over a period of 10 minutes was 17830, while the total over 30 minutes was 53490. These values reflect the homomorphic operations performed on the ciphertexts.
- **Average Heart Rate (bmp_media)**: After rescaling (division by 100) ,the total heart rate, the average heart rate across all individuals was calculated as 71.72 BPM.

Based on the scaled heart rate metrics, the following classifications were made:

- **Athletic Fitness (Atleta)**: An individual was classified as an athlete if their heart rate, after multipling by 1.5 and scaled by a factor of 100, was below 10,000 BPM(i.e 100 BPM. Based on this criterion, 10 individuals were classified as athletes, while the others were classified as not athlete.
- **Obesity Detection (*Obesidade*)**: Individuals were classified as obese if their heart rate, after multipling by 1.2 and scaled by a factor of 100, were above 10,000 BPM(i.e 100 BPM), and as morbidly obese if the value exceeded 12,000 BPM(i.e 120 BPM). However, none of the individuals were classified as obese or morbidly obese based on these criteria.

3.3.4 Discussion

This experiment highlights both the potential and the challenges of using Paillier homomorphic encryption for securely processing sensitive health data. While the scheme enabled secure operations directly on encrypted heart rate data, we encountered significant limitations due to Paillier's inability to perform division natively. This restriction required us to adopt a workaround by scaling the values, multiplying them by appropriate factors to simulate division operations (e.g., scaling averages and fitness metrics).

By leveraging this scaling approach, we successfully computed essential metrics such as average heart rates and classifications for fitness and obesity while maintaining data privacy. Although this workaround introduced additional computational steps, it allowed us to overcome the encryption scheme's inherent limitations and achieve the desired outcomes.

This experiment demonstrates the feasibility of secure data processing in healthcare using Paillier encryption but also underscores the need for further optimizations, especially for handling more complex arithmetic operations at scale.

4 Conclusion

This project successfully demonstrated the application of homomorphic encryption, specifically the Paillier encryption scheme, for secure processing of sensitive heart rate data. We addressed the challenge of maintaining data privacy during analysis while enabling computations to be performed directly on encrypted values.

The experiments were conducted in three phases:

1. **Encryption Performance Analysis:** We compared the performance of three additive partially homomorphic encryption schemes—Paillier, Damgård-Jurik, and ElGamal—across varying key sizes. The results highlighted the trade-offs between computational cost and cryptographic flexibility, with Paillier providing a balanced performance.

2. **Temporal Performance of Paillier:** We evaluated the efficiency of Paillier encryption, decryption, and homomorphic addition. While encryption and decryption times increased with key size, homomorphic addition remained computationally efficient, making it suitable for frequent additive operations on encrypted data.

3. **Statistical Analysis Using Paillier:** We applied Paillier homomorphic encryption to securely compute heart rate metrics, such as total sum, averages, and fitness classifications. Despite the inherent limitations of the Paillier scheme (e.g., the inability to perform direct division), we overcame these challenges by implementing scaled values to approximate the required results. This approach enabled us to classify individuals securely as athletes or detect obesity without exposing sensitive data.

The results demonstrate that Paillier encryption is a viable solution for secure additive operations, offering a strong balance between privacy and computational efficiency. However, the experiments also revealed some limitations, such as the computational overhead for encryption and decryption and the need for workarounds when performing more complex operations like division.

In conclusion, this work highlights both the opportunities and challenges of deploying homomorphic encryption in practical healthcare applications. While Paillier proved effective for basic statistical operations, further optimizations or alternative schemes may be required to ensure scalability for larger datasets and more complex computations. Nevertheless, this study underscores the significant potential of homomorphic encryption in safeguarding sensitive data in privacy-critical domains, such as healthcare.

References

[1] Abbas Acar et al. “A survey on homomorphic encryption schemes: Theory and implementation”. In: *ACM Computing Surveys (CSUR)* 51.4 (2018), 79:1–79:35.

[2] Jung Hee Cheon et al. “Homomorphic encryption for arithmetic of approximate numbers”. In: *Advances in Cryptology – ASIACRYPT 2017*. 2017, pp. 409–437.

[3] Ivan Damgård and Mads Jurik. “A Generalisation, a Simplification and Some Applications of Paillier’s Probabilistic Public-Key System”. In: *Public Key Cryptography - PKC 2001*. Springer, 2001, pp. 119–136.

[4] Ivan B. Damgård and Mikkel Jurik. “Trapdoor Discrete Logarithms and Cryptosystems Based on Composite Degree Residues”. In: *Proceedings of CRYPTO 2001* (2001), pp. 261–279.

[5] Taher ElGamal. *A Public Key Cryptosystem and a Signature Scheme Based on Discrete Logarithms*. Vol. 31. 4. IEEE, 1985, pp. 469–472.

[6] Craig Gentry. “A Fully Homomorphic Encryption Scheme”. PhD thesis. Stanford University, 2009.

[7] Miran Kim, Jung Hee Cheon, and Jonghwa Park. “Privacy-preserving machine learning using homomorphic encryption”. In: *Springer, Secure Data Analytics*. 2019, pp. 68–82.

[8] Pascal Paillier. “Public-key cryptosystems based on composite degree residuosity classes”. In: *Advances in Cryptology – EUROCRYPT ’99*. 1999, pp. 223–238.

[9] Ronald L. Rivest, Leonard Adleman, and Michael L. Dertouzos. “On data banks and privacy homomorphisms”. In: *Foundations of Secure Computation* 4.11 (1978), pp. 169–180.

[10] Ronald L. Rivest, Adi Shamir, and Leonard Adelman. *A Method for Obtaining Digital Signatures and Public-key Cryptosystems*. Vol. 21. 2. ACM, 1977, pp. 120–126.

[11] Hao Wang, Yiping Lu, and Xiaoming Zhang. “Protecting patient data privacy with homomorphic encryption”. In: *Journal of Healthcare Informatics Research* 4.3 (2020), pp. 225–240.