Final Project - The Paillier Cryptosystem

Ramya Mohan

1 Introduction

The Paillier cryptosystem, introduced by Pascal Paillier in 1999, is a public-key encryption scheme renowned for its additive homomorphic property, which allows computations to be performed directly on encrypted data without decryption. This feature distinguishes it from traditional schemes and makes it highly applicable in areas such as secure electronic voting, privacy-preserving data analysis, and multiparty computation. The cryptosystem is based on the decisional composite residuosity assumption and leverages modular arithmetic over the group $\mathbb{Z}_{n^2}^*$, extending earlier work in probabilistic encryption. Its introduction marked a significant step forward in the design of cryptographic systems that not only secure data but also enable encrypted operations. Studying the Paillier cryptosystem is important both theoretically, as it enhances our understanding of homomorphic encryption and number-theoretic assumptions, and practically, as it supports the growing need for privacy-preserving computation in fields like cloud computing, machine learning, and blockchain technologies.

2 Mathematical Foundation

The Paillier cryptosystem relies on the hardness of the *composite residuosity problem*, formalized as the *Decisional Composite Residuosity Assumption (DCRA)*. In number theory, an integer $c \in \mathbb{Z}_{n^2}^*$ is called a *composite residue modulo* n^2 if there exists some $y \in \mathbb{Z}_{n^2}^*$ such that $c = y^n \mod n^2$, where $n = pq$ is a product of large primes. The DCRA asserts that, given n and a random c, it is computationally infeasible to decide whether c is a composite residue modulo n^2. This intractability underpins the semantic security of the scheme and is assumed to be as hard as factoring n. As a result, the Paillier cryptosystem remains secure under the assumption that composite residuosity is difficult to distinguish without knowledge of the factorization of n.

3 Working of the Paillier Cryptosystem

Key Generation (Public and Private Keys) To generate the keys, first select two large prime numbers p and q randomly and independently, such that $\gcd(pq, (p-1)(q-1)) = 1$. Then compute $n = p \cdot q$ and also calculate n^2. Next, choose a generator $g \in \mathbb{Z}_{n^2}^*$ such that $\gcd(L(g^\lambda \bmod n^2), n) = 1$, where the L-function

is defined as $L(x) = \frac{x-1}{n}$. Compute $\lambda = \text{lcm}(p-1, q-1)$, and then determine μ as the modular inverse of $L(g^\lambda \bmod n^2)$ modulo n, i.e., $\mu = (L(g^\lambda \bmod n^2))^{-1} \bmod n$. The public key consists of the pair (n, g), while the private key consists of the pair (λ, μ).

Encryption (Using the Public Key) To encrypt a plaintext message m, choose a random value $r \in \mathbb{Z}_n^*$ such that $0 < r < n$. Then compute the ciphertext as $c = g^m \cdot r^n \bmod n^2$.

Decryption (Using the Private Key) To decrypt a ciphertext c, compute $u = c^\lambda \bmod n^2$, then evaluate $L(u) = \frac{u-1}{n}$. Finally, recover the original plaintext by calculating $m = L(u) \cdot \mu \bmod n$.

4 Mathematical Proof of Paillier Decryption

We use the simplification $g = 1 + n$, which guarantees $\gcd(L(g^\lambda \bmod n^2), n) = 1$ and enables a simplified proof using the binomial theorem. Let $c = g^m r^n \bmod n^2$. Then decryption requires computing $c^\lambda \bmod n^2$:

$$c^\lambda = (g^m r^n)^\lambda \equiv g^{m\lambda} \cdot r^{n\lambda} \bmod n^2$$

Since $\lambda = \varphi(n)$ and $\varphi(n^2) = n\varphi(n)$, Euler's theorem implies $r^{n\lambda} \equiv 1 \bmod n^2$, so:

$$c^\lambda \equiv g^{m\lambda} \equiv (1+n)^{m\lambda} \bmod n^2$$

Using the binomial theorem, $(1+n)^{m\lambda} = 1 + nm\lambda + n^2 k$ for some k, so:

$$c^\lambda \equiv 1 + nm\lambda \bmod n^2$$

Now compute $m = L(c^\lambda \bmod n^2) \cdot \mu \bmod n$, where $L(u) = \frac{u-1}{n}$, so:

$$m = \frac{1 + nm\lambda - 1}{n} \cdot \mu \bmod n = m\lambda \cdot \mu \bmod n$$

Since $\mu = \lambda^{-1} \bmod n$, we conclude:

$$m = m \bmod n$$

Thus, the decryption process is correct, and the original message m is successfully recovered.

5 Example: Encrypting and Decrypting "MATH"

We choose primes $p = 17, q = 19$, so $n = pq = 323$, $n^2 = 104329$, $\lambda = \text{lcm}(p-1, q-1) = \text{lcm}(16, 18) = 144$, and select generator $g = n + 1 = 324$. We compute $\mu = (L(g^\lambda \bmod n^2))^{-1} \bmod n$.

Since $g^\lambda \bmod n^2 = 1 + n\lambda = 1 + 323 \cdot 144 = 46513$, $L(46513) = \frac{46513-1}{323} = 144$, $\mu = 144^{-1} \bmod 323 = 83$.

//We encode "MATH" as $[13, 1, 20, 8]$ using the mapping $A = 1, B = 2, \ldots, Z = 26$. Choose random values $r = [67, 59, 103, 41] \in \mathbb{Z}_n^*$. We compute ciphertexts using $c = g^m \cdot r^n \bmod n^2$:

$$c_1 = 324^{13} \cdot 67^{323} \bmod 104329 = 5626$$

$$c_2 = 324^{1} \cdot 59^{323} \bmod 104329 = 27389$$

$$c_3 = 324^{20} \cdot 103^{323} \bmod 104329 = 22356$$

$$c_4 = 324^{8} \cdot 41^{323} \bmod 104329 = 65487$$

To decrypt, compute $u = c^\lambda \bmod n^2$, then $m = L(u) \cdot \mu \bmod n$, where $L(u) = \frac{u-1}{n}$. For each ciphertext:

$$u_1 = 5626^{144} \bmod 104329 = 83012, \quad m_1 = \frac{83012 - 1}{323} \cdot 83 \bmod 323 = 13$$

$$u_2 = 27389^{144} \bmod 104329 = 46513, \quad m_2 = \frac{46513 - 1}{323} \cdot 83 \bmod 323 = 1$$

$$u_3 = 22356^{144} \bmod 104329 = 95609, \quad m_3 = \frac{95609 - 1}{323} \cdot 83 \bmod 323 = 20$$

$$u_4 = 65487^{144} \bmod 104329 = 59110, \quad m_4 = \frac{59110 - 1}{323} \cdot 83 \bmod 323 = 8$$

Decrypted message: "MATH"

6 Security Analysis and Example Attack

The Paillier cryptosystem provides strong security guarantees through multiple features. Its semantic security ensures that encrypting the same plaintext multiple times yields different ciphertexts, thanks to the random value $r \in \mathbb{Z}_n^*$, making pattern detection by attackers infeasible. Additionally, Paillier's additive homomorphic property allows meaningful computations on encrypted data—such as summing votes or aggregating statistics—without decrypting, preserving privacy in sensitive applications. Finally, its security is based on the decisional composite residuosity assumption (DCRA), a problem considered computationally hard, much like the factoring assumption in RSA.

Despite its strengths, the Paillier cryptosystem also has vulnerabilities. The encryption process relies on the random selection of r, and if r is reused or poorly generated, it can leak information about the plaintext.

Moreover, since the private key is derived from the factorization of n, if an attacker can factor n, they can compute λ and μ, breaking the system. Paillier is also not semantically secure against chosen ciphertext attacks (CCA) without additional cryptographic padding or safeguards.

To demonstrate a vulnerability, consider the intercepted ciphertexts $c_1 = 5626$, $c_2 = 27389$, $c_3 = 22356$, and $c_4 = 65487$, sent using parameters $n = 323$, $g = 324$. An attacker, knowing these public values, can brute-force all valid plaintext values $m \in [1, 26]$ (assuming letters A–Z) and valid random values $r \in \mathbb{Z}_n^*$. For each guess, the attacker computes $c = g^m \cdot r^n \bmod n^2$ and compares it with the observed ciphertexts. Through this exhaustive search, the attacker eventually recovers:

- $c_1 = 5626 \Rightarrow m = 13$ (M)
- $c_2 = 27389 \Rightarrow m = 1$ (A)
- $c_3 = 22356 \Rightarrow m = 20$ (T)
- $c_4 = 65487 \Rightarrow m = 8$ (H)

This reveals the message "MATH". Although effective here, such brute-force attacks are only feasible when n is small. In real-world scenarios, where n is hundreds of digits long, these attacks are computationally infeasible, reinforcing the importance of using strong, large parameters.

7 Improving the Paillier Cryptosystem with Hybrid Encryption

One practical limitation of the Paillier cryptosystem is its inefficiency when encrypting large messages, as it operates over large integers modulo n^2. Additionally, while Paillier offers semantic security under the Decisional Composite Residuosity Assumption (DCRA), it is not secure against chosen ciphertext attacks (CCA). To address both issues, a hybrid encryption scheme can be employed. This combines the strength of asymmetric encryption (Paillier) for securing keys with the speed and CCA-security of symmetric encryption (such as AES) for encrypting the actual message.

How Hybrid Encryption Works

The core idea is to use Paillier to encrypt a randomly generated symmetric key, and then use that key with a fast symmetric cipher (e.g., AES in GCM mode) to encrypt the message. This allows the system to benefit from both the homomorphic properties of Paillier and the efficiency and authenticated encryption of AES.

Steps

1. **Key Generation:** The sender generates a random AES key K (e.g., 128-bit).

2. **Message Encryption:**

   - The plaintext message M is encrypted using AES with key K, producing ciphertext C_{sym}.
   - The AES key K is then encrypted using the recipient's Paillier public key, resulting in ciphertext C_{asym}.

3. **Transmission:** The sender transmits both C_{asym} and C_{sym} to the receiver.

4. **Decryption:**

   - The receiver uses their Paillier private key to decrypt C_{asym}, recovering the AES key K.
   - The recovered key K is used to decrypt C_{sym}, revealing the original message M.

The complete Python implementation of this hybrid encryption system and its output are provided in the Appendix for reference.

Security and Efficiency Gains

Hybrid encryption addresses the vulnerabilities of the Paillier cryptosystem by combining the strengths of both asymmetric and symmetric encryption. While Paillier's additive homomorphic property ensures privacy for computations, it is inefficient for large messages and susceptible to chosen ciphertext attacks (CCA). By using Paillier only to securely exchange a symmetric key, and then employing AES to encrypt the actual message, the hybrid approach significantly improves efficiency and prevents CCA vulnerabilities. AES, being fast and providing authenticated encryption, mitigates risks like message tampering and ensures robust security, making the system both secure and practical for larger, real-world applications.

8 Applications of Improved Paillier Cryptosystem

The hybrid encryption system, combining Paillier and AES, is suitable for settings requiring secure key exchange and efficient, large-scale data encryption.

1. Cloud Computing and Secure Data Sharing

In cloud environments, users store sensitive data that must be securely accessed and shared. Hybrid encryption ensures AES efficiently encrypts data, while Paillier secures the AES key exchange, preventing unauthorized access and ensuring cloud providers cannot view the data.

2. Privacy-Preserving Medical Data Analysis

In healthcare, hybrid encryption enables secure sharing of medical data for research. AES encrypts patient records, while Paillier securely exchanges AES keys, ensuring privacy and compliance with data-sharing regulations like HIPAA.

3. Secure Voting Systems

Hybrid encryption enhances electronic voting by using Paillier's homomorphic property to aggregate encrypted votes, while AES encrypts vote data. This ensures both security and efficiency, keeping votes private and protected from tampering.

4. Financial Transactions and Digital Payments

In finance, hybrid encryption secures transactions by using AES for efficient encryption of transaction details and Paillier for secure key exchange, ensuring privacy and security without exposing data during transmission.

5. Blockchain and Privacy-Preserving Smart Contracts

In blockchain, hybrid encryption secures smart contract execution. AES encrypts sensitive data, while Paillier secures the key exchange, allowing only authorized participants to decrypt and access contract details, enhancing privacy in decentralized applications.

By combining AES and Paillier, this hybrid system ensures security, privacy, and scalability in cloud computing, healthcare, digital payments, and blockchain applications.

A Appendix

```
from Crypto.Cipher import AES
from Crypto.Random import get_random_bytes
from phe import paillier
import base64


# Pad message for AES
def pad(msg):
    padding_length = 16 - len(msg) % 16
    return msg + chr(padding_length) * padding_length
```

```
def unpad(msg):
    padding_length = ord(msg[-1])
    return msg[:-padding_length]

# Key Generation (Paillier)
public_key, private_key = paillier.generate_paillier_keypair()

# Generate AES key
aes_key = get_random_bytes(16)

# Encrypt message with AES
message = "MATH"
cipher_aes = AES.new(aes_key, AES.MODE_ECB)
ciphertext_aes = cipher_aes.encrypt(pad(message).encode())

# Encrypt AES key with Paillier
aes_key_int = int.from_bytes(aes_key, byteorder='big')
ciphertext_paillier = public_key.encrypt(aes_key_int)

# Decrypt AES key
decrypted_aes_key_int = private_key.decrypt(ciphertext_paillier)
decrypted_aes_key = decrypted_aes_key_int.to_bytes(16, byteorder='big')

# Decrypt message
cipher_aes_dec = AES.new(decrypted_aes_key, AES.MODE_ECB)
decrypted_message = cipher_aes_dec.decrypt(ciphertext_aes).decode()
decrypted_message = unpad(decrypted_message)

# Output
print("Original␣Message:", message)
print("Decrypted␣Message:", decrypted_message)
```

Listing 1: Hybrid Encryption using Paillier and AES

[7]:

```
!pip install pycryptodome phe
from Crypto.Cipher import AES
from Crypto.Random import get_random_bytes
from phe import paillier
import base64

# Helper function to pad message to AES block size (16 bytes)
def pad(msg):
    padding_length = 16 - len(msg) % 16
    return msg + chr(padding_length) * padding_length

def unpad(msg):
    padding_length = ord(msg[-1])
    return msg[:-padding_length]

# Step 1: Paillier Key Generation
public_key, private_key = paillier.generate_paillier_keypair()

# Step 2: Symmetric Key Generation (AES key)
aes_key = get_random_bytes(16)  # 128-bit AES key

# Step 3: Encrypt the message "MATH" using AES
message = "MATH"
cipher_aes = AES.new(aes_key, AES.MODE_ECB)  # Use ECB for simplicity (not secure in real life)
ciphertext_aes = cipher_aes.encrypt(pad(message).encode())

# Step 4: Encrypt the AES key using Paillier
# Convert AES key bytes to an integer
aes_key_int = int.from_bytes(aes_key, byteorder='big')
ciphertext_paillier = public_key.encrypt(aes_key_int)

# Transmit ciphertext_paillier and ciphertext_aes

# Step 5: Decryption
# Decrypt AES key using Paillier
decrypted_aes_key_int = private_key.decrypt(ciphertext_paillier)
decrypted_aes_key = decrypted_aes_key_int.to_bytes(16, byteorder='big')

# Decrypt message using AES
cipher_aes_dec = AES.new(decrypted_aes_key, AES.MODE_ECB)
decrypted_message = cipher_aes_dec.decrypt(ciphertext_aes).decode()
decrypted_message = unpad(decrypted_message)

# Output
print("Original Message:", message)
print("Decrypted Message:", decrypted_message)
```

```
Requirement already satisfied: pycryptodome in c:\users\ramsy\anaconda3\lib\site-packages (3.22.0)
Requirement already satisfied: phe in c:\users\ramsy\anaconda3\lib\site-packages (1.5.0)
Original Message: MATH
Decrypted Message: MATH
```

Figure 1: Implementation of Hybrid Encryption in Paillier Cryptosystem