

# Report on Various Encryption Systems

## Paillier Cryptosystem

The Paillier cryptosystem is a public key encryption system that uses the mathematical properties of large prime numbers and modular arithmetic. This report details the encryption and decryption processes and example implementation in Python.

### Key Generation

- Choose two large primes  $p$  and  $q$  such that  $\gcd((p-1)(q-1), pq) = 1$ .
- Compute  $n = pq$ .
- Calculate  $\phi(n) = (p-1)(q-1)$ .
- Find  $\mu$ , the modular inverse of  $\phi(n)$  modulo  $n$ . This  $\mu$  is kept private.
- The public key includes  $n$  and  $g = n + 1$ .

### Encryption

- The message  $m$  must be in the set  $\{0, 1, 2, \dots, n-1\}$ .
- Choose a random integer  $r$ .
- The ciphertext  $c$  is calculated as:

$$c = (g^m \cdot r^n) \mod n^2$$

### Decryption Process

- ★ Compute:

$$d = c^{\phi(n)} \mod n^2$$

- ★ This results in:

$$d = (g^m \cdot r^n)^{\phi(n)} \mod n^2 = (1+n)^{m \cdot \phi(n)} \mod n^2$$

- ★ Simplifying further, we get:

$$d = 1 + mn\phi(n) \mod n^2$$

- ★ Calculate:

$$\frac{d-1}{n} = m\phi(n) \mod n^2$$

- ★ Finally:

$$\mu \cdot \frac{d-1}{n} = m\phi(n) \cdot \text{inverse of } \phi(n) \mod n^2 = m$$

- ★ The original message  $m$  is successfully decrypted.

## Example Code Implementation

```
from Crypto.Util.number import getPrime, inverse, GCD
import random

def generate_keys(bits):
    while True:
        p = getPrime(bits)
        q = getPrime(bits)
        if GCD((p-1)*(q-1), p*q) == 1:
            break

    n = p * q
    phi_n = (p-1) * (q-1)
    mu = inverse(phi_n, n)
    g = n + 1
    public_key = (n, g)
    private_key = (phi_n, mu)

    return public_key, private_key

def encrypt(public_key, message):
    n, g = public_key
    r = random.randint(1, n-1)
    c = pow(g, message, n*n) * pow(r, n, n*n) % (n*n)
    return c

def decrypt(private_key, public_key, ciphertext):
    n, g = public_key
    phi_n, mu = private_key
    d = pow(ciphertext, phi_n, n*n)
    m_phi_n = (d - 1) // n
    m = m_phi_n * mu % n
    return m

# Example usage:
if __name__ == "__main__":
    public_key, private_key = generate_keys(1024)
    message = 42
    print("Original message:", message)

    ciphertext = encrypt(public_key, message)
    print("Encrypted message:", ciphertext)

    decrypted_message = decrypt(private_key, public_key, ciphertext)
    print("Decrypted message:", decrypted_message)
```

```
charitha_77@CHARITHA:/mnt/c/Users/91912/Desktop/The Art of Shadow Ops/Week2-Report$ python3 pilliar.py
Original message: 42
Encrypted message: 389886956736733347815024575680952308842523673749378614123460057086048851291455234610608537171684404378382354085012384775
3761084449133404441788706017192459273553881059114744016375192699350614550770610034054514520959453442259946584260026809374855570765041748842
5459685792484418536873462556641714159291528727341504576677183981429263172019859996299534584615362647655878905181808870593902228815684647794
6192206848586661694603148954478681622088147981513348972430410907024043117048255670448822992566804485314856571638234939490431789095089281350
3464333054534646810218097343236161305771393171842993026306625813255576965862229415201995650231470675965672510203360403000496423997448411356
2781954602820064830955278797798351797420296204955724069891315396898016419822437421446333446052088469102594256029008393617000586485539801527
1470828297734775319008244183840117702318680711841961025553573273588241115384159389883175019070985169931348476313784091891157549600090593624
225890929469249067399651256907768240812348619601755749636985584675157394424670218162330434964754847758880498212376263708454790212612977438

Decrypted message: 42
charitha_77@CHARITHA:/mnt/c/Users/91912/Desktop/The Art of Shadow Ops/Week2-Report$
```

Figure 1: Paillier Output

# RSA Cryptosystem

The RSA cryptosystem is one of the first public key cryptosystems and is widely used for secure data transmission. It is based on the mathematical properties of large prime numbers and modular arithmetic. This report details the encryption and decryption processes and shows an example implementation in Python.

## Encryption Process

### Key Generation

- Choose two large prime numbers  $p$  and  $q$ .
- Compute  $n = pq$ .
- Calculate  $\phi(n) = (p-1)(q-1)$ .
- Choose an integer  $z$  such that  $\gcd(z, \phi(n)) = 1$ .
- Compute  $d$ , the modular inverse of  $z$  modulo  $\phi(n)$ .
- The public key consists of  $(n, z)$  and the private key consists of  $(n, d)$ .

### Encryption

- The message  $m$  must be in the set  $\{0, 1, 2, \dots, n-1\}$ .
- The ciphertext  $c$  is calculated as:

$$c = m^z \mod n$$

## Decryption Process

- \* The original message  $m$  is obtained by:

$$m = c^d \mod n$$

- \* Further breakdown:

1. Calculate  $m_p = c^d \mod (p-1) \mod p$ .
2. Calculate  $m_q = c^d \mod (q-1) \mod q$ .
3. Use the Chinese Remainder Theorem (CRT) to find  $m$ :

$$m = \text{CRT}(m_p, m_q, p, q)$$

- \* Explanation:

- $d$  satisfies  $zd \equiv 1 \mod \phi(n)$ , which implies  $zd = 1 + k \cdot \phi(n)$ .
- $zd \equiv 1 \mod (p-1)$  and  $zd \equiv 1 \mod (q-1)$  due to the properties of  $d$  and  $z$ .
- Therefore,  $m \equiv c^d \mod n$  can be efficiently computed using CRT by first computing  $m_p$  and  $m_q$ .

## Example Code Implementation

```
from Crypto.Util.number import getPrime, inverse, GCD
import random

def generate_keys(bits):
    p = getPrime(bits)
    q = getPrime(bits)
    n = p * q
    phi_n = (p-1) * (q-1)
```

```

while True:
    z = random.randint(2, phi_n - 1)
    if GCD(z, phi_n) == 1:
        break

d = inverse(z, phi_n)
public_key = (n, z)
private_key = (n, d)

return public_key, private_key

def encrypt(public_key, message):
    n, z = public_key
    c = pow(message, z, n)
    return c

def decrypt(private_key, ciphertext):
    n, d = private_key
    m = pow(ciphertext, d, n)
    return m

# Example usage:
if __name__ == "__main__":
    public_key, private_key = generate_keys(1024)
    message = 42
    print("Original message:", message)

    ciphertext = encrypt(public_key, message)
    print("Encrypted message:", ciphertext)

    decrypted_message = decrypt(private_key, ciphertext)
    print("Decrypted message:", decrypted_message)

```

```

charitha_77@CHARITHA:/mnt/c/Users/91912/Desktop/The Art of Shadow Ops/Week2-Report$ python3 rsa.py
Original message: 42
Encrypted message: 178501932108220456669355790334996691883603399252694022352556518951369954339243478265227092341976606877447686160703047063
1787110921430823440923058382706108029170804730389359718288709763757534198316111797573867371037476770011332058360443153191479924919261520915
2392188162559950592798923102621205784643697151086162775405724957681880130098661296939931877501284144064222113431984570346836136796213577559
4130293296354269837709027391351858201830301876170988453743668334890875641922488899353252153609580351655048237886297045855346256480209479378
05719943575832993893998914539983546245130550510237754421434846104515651982692610
Decrypted message: 42
charitha_77@CHARITHA:/mnt/c/Users/91912/Desktop/The Art of Shadow Ops/Week2-Report$ python3 rsa.py

```

Figure 2: RSA Output

## Elgamal Cryptosystem

The ElGamal cryptosystem is a public key cryptosystem based on the Diffie-Hellman key exchange and is used for secure data transmission. This report details the encryption and decryption processes and provides an example implementation in Python.

### Key Generation

- Choose a large prime number  $p$ .
- Choose an integer  $\alpha$  such that  $\alpha$  is a primitive root modulo  $p$ .
- Choose a random integer  $a$  such that  $1 \leq a \leq p - 2$ .
- Compute  $\beta = \alpha^a \mod p$ .
- Public key is  $(p, \alpha, \beta)$  and private key is  $a$ .

## Encryption Process

- Choose a random integer  $k$  such that  $1 \leq k \leq p-2$  and  $\gcd(k, p-1) = 1$ .
- Compute  $C_1 = \alpha^k \mod p$ .
- Compute  $C_2 = m \cdot \beta^k \mod p$ , where  $m$  is the plaintext message.
- The ciphertext is  $(C_1, C_2)$ .

## Decryption Process

- ★ Compute  $D = C_1^a \mod p$ .
- ★ Compute  $D^{-1}$ , the modular inverse of  $D$  modulo  $p$ .
- ★ Compute the plaintext message  $m$  as  $m = C_2 \cdot D^{-1} \mod p$ .

To understand how decryption works in the ElGamal cryptosystem:

- ★ Begin with  $C_1^a \mod p$ :

$$D = (\alpha^k)^a \mod p$$

- ★ Compute  $D^{-1}$ , the modular inverse of  $D$  modulo  $p$ :

$$D^{-1} = (C_1^a)^{-1} \mod p$$

- ★ Finally, compute the plaintext message  $m$ :

$$m = C_2 \cdot D^{-1} \mod p$$

## Example Code Implementation

```
from Crypto.Util.number import getPrime, inverse, GCD
import random
```

```
def generate_keys(bits):
    p = getPrime(bits)
    alpha = random.randint(2, p-1)
    while pow(alpha, (p-1)//2, p) == 1:
        alpha = random.randint(2, p-1)
    a = random.randint(1, p-2)
    beta = pow(alpha, a, p)
    public_key = (p, alpha, beta)
    private_key = (a, p)
    return public_key, private_key

def encrypt(public_key, message):
    p, alpha, beta = public_key
    k = random.randint(1, p-2)
    while GCD(k, p-1) != 1:
        k = random.randint(1, p-2)
    C1 = pow(alpha, k, p)
    C2 = (message * pow(beta, k, p)) % p
    return (C1, C2)

def decrypt(private_key, ciphertext):
    a, p = private_key
    C1, C2 = ciphertext
    D = pow(C1, a, p)
    D_inv = inverse(D, p)
    m = (C2 * D_inv) % p
```

```


    return m

# Example usage:
if __name__ == "__main__":
    public_key, private_key = generate_keys(1024)
    message = 17
    print("Original message:", message)

    ciphertext = encrypt(public_key, message)
    print("Encrypted message:", ciphertext)

    decrypted_message = decrypt(private_key, ciphertext)
    print("Decrypted message:", decrypted_message)

```



```

charitha_77@CHARITHA:/mnt/c/Users/91912/Desktop/The Art of Shadow Ops/Week2-Report$ python3 Elgamal.py
Original message: 17
Encrypted message: (13713465220654483802302347988188899152736475973092641794757679902565165617210870937301248713038555290364361129113234975
8176825987540605657518277186146278964058752519780127901484279899544324889053320777323778612671363303637264892341066897412681195996140690769
679508399740261212748621058027643376649246301104731, 12382299801230015214673065436311194318284216145362694159755336411245123593404800504097
1964412435081283964157825227301109133594483136485292580886957286211300247569513467457305805291782542657004252764127267384690211004701453377
80818504638224543583689152305428774338848295743411600025645074783555975129604737046)
Decrypted message: 17
charitha_77@CHARITHA:/mnt/c/Users/91912/Desktop/The Art of Shadow Ops/Week2-Report$

```

Figure 3: Elgamal Output

## Conclusion

In this report, we have explored three prominent cryptographic systems: Pailliar's Cryptosystem, RSA (Rivest-Shamir-Adleman), and the ElGamal Cryptosystem. Each of these systems offers unique advantages and considerations in terms of key generation, encryption, and decryption processes.

## Comparison of Methods

- Pailliar's Cryptosystem:
  - **Key Generation:** Involves choosing large primes  $p$  and  $q$ , making it crucial for the system's security.
  - **Encryption and Decryption:** Relies on modular exponentiation, leading to a time complexity of  $O(\log n)$ .
- RSA Cryptosystem:
  - **Key Generation:** Requires the selection of large primes  $p$  and  $q$ , followed by computations for  $n$ ,  $\phi(n)$ , and finding suitable  $e$  and  $d$ .
  - **Encryption and Decryption:** Both operations are  $O(\log n)$ , leveraging modular exponentiation for efficiency.
- ElGamal Cryptosystem:
  - **Key Generation:** Involves selecting a large prime  $p$ , finding a primitive root  $\alpha$ , and computing  $\beta = \alpha^a \mod p$ .
  - **Encryption and Decryption:** Uses modular exponentiation with a time complexity of  $O(\log p)$ , providing a secure method with capabilities for digital signatures.

## Time Complexity Comparison

All three cryptographic systems rely heavily on modular arithmetic and prime number operations. Here is a summary of their time complexities:

- ★ Pailliar's Cryptosystem:  $O(\log n)$
- ★ RSA Cryptosystem:  $O(\log n)$
- ★ ElGamal Cryptosystem:  $O(\log p)$

## Other Considerations

- Security: The security of these systems relies on the difficulty of prime factorization (RSA), discrete logarithm problem (ElGamal), and other mathematical assumptions. - Implementation Complexity: Each system requires careful implementation of key generation, encryption, and decryption algorithms, often leveraging efficient modular arithmetic libraries.

```
charitha_77@CHARITHA:/mnt/c/Users/91912/Desktop/The Art of Shadow Ops/Week2-Report$ python3 paillier.py
Original message: 42
Encrypted message: 288311420249679121309034747785481635988913208585118650146909444608842355867041600640367102931590350727438776946829343226
Decrypted message: 42
Time taken: 0.0025 seconds
charitha_77@CHARITHA:/mnt/c/Users/91912/Desktop/The Art of Shadow Ops/Week2-Report$ python3 Elgamal.py
Original message: 42
Encrypted message: (191880063369141944264387374973, 164914858568612361166663097430)
Decrypted message: 42
Time taken: 0.0032 seconds
charitha_77@CHARITHA:/mnt/c/Users/91912/Desktop/The Art of Shadow Ops/Week2-Report$ python3 rsa.py
Original message: 42
Encrypted message: 665455763345360575830795728631089871756893659526322088860014
Decrypted message: 42
Time taken: 0.0033 seconds
charitha_77@CHARITHA:/mnt/c/Users/91912/Desktop/The Art of Shadow Ops/Week2-Report$
```

Figure 4: Output of all three with bits set to 100

From above we may assume that RSA and Elgamal takes slightly more time compared to Pailliar's encryption although time complexity were almost same .