20CYS205 – MODERN CRYPTOGRAPHY

IMPLEMENT AND CRYPTANALYSE PAILLIER ENCRYPTION SCHEME

Submitted by

REDDICHERLA THANUJ – CB.EN.U4CYS22056

S MOHANA VAMSI – CB.EN.U4CYS220057

SARIDE SOMESWARA SAI SRI CHAKRI – CB.EN.U4CYS220059

SHREE HARINI T – CB.EN.U4CYS220060

Under the guidance of

Aravind Vishnu S

Research Scholar

Amrita Vishwa Vidyapeetham Coimbatore

TIFAC-CORE IN CYBER SECURITY AMRITA SCHOOL OF ENGINEERING

AMRITA VISHWA VIDYAPEETHAM

COIMBATORE - 641 112

2023

Schema:

Key generation:

1. Pick two large prime numbers p and q randomly. Confirm that *gcd*(pq, (p-1)(q-1)) is 1. If not, pick another pair of prime numbers.
2. Compute **n = p*q**.
3. Define function **L(x) = (x−1) / n.**
4. Compute carmichael function **λ**(n) = **LCM(p-1, q-1).**
5. Pick a random integer **g** , such that g belongs to $Z^*_{n\times n}$ and order of g is non-zero multiple of n.
6. Calculate the modular multiplicative inverse μ **= $(L(g^\lambda \bmod n^2))^{-1} \bmod n$**. If no μ exists, then restart from step 1.
7. The public key is **(n, g),** which can be used for encryption.
8. The private key is **λ** which can be used for decryption.

Encryption:

1. Any block m of the message in the range $0 \le m < n$ can be encrypted
2. Pick any random number **r** in the range $0 < r < n$, gcd(r,n) = 1
3. Compute the ciphertext **$c = g^m * r^n \bmod n^2$** .

Decryption:

- c will be such that $0 < c < n^2$.
- Now the plaintext can be recalculated as **$m = L(c^\lambda \bmod n^2).\mu \bmod$** n.
- Also, **λ** and **μ** can be recalculated from the public key.

Correctness:

$c = g^m \times r^n \pmod{n^2}$
$c^\lambda = g^{m\lambda} \times r^{n\lambda} \pmod{n^2}$
$r^{n\lambda} \pmod{n^2} = (r^\lambda)^n \pmod{n^2}$
$=(r^{(p-1)(q-1)/gcd(p-1,q-1)})^n \pmod{n^2}$
$\Rightarrow r^{(p-1)} = 1 \pmod p$
r^((p−1)(q−1)/gcd(p−1,q−1)) = 1 (mod p)
$r^{(q-1)} = 1 \pmod q$
r^((p−1)(q−1)/gcd(p−1,q−1)) = 1 (mod q)
r(p−1)(q−1)/gcd(p−1,q−1) = 1 (mod pq)
$r^\lambda = 1 \pmod n$
$r^{n\lambda} = 1 \pmod{n^2}$

$\Rightarrow c^\lambda = g^{m\lambda} \times r^{n\lambda} \pmod{n^2}$
$= g^{m\lambda} \pmod{n^2}$
$= (1+n)^{m\lambda} \pmod{n^2}$
$= 1+n(m\lambda) \pmod{n^2}$

$L(c^\lambda \pmod{n^2}) = 1+n(m\lambda)-1/n \pmod n$
$= nm\lambda/n \pmod n$
$= m\lambda \pmod n$

$L(g^\lambda \pmod{n^2}) = 1+n\lambda-1/n$
$= \lambda$

$L(c^\lambda \pmod{n^2}) \,/\, L(g^\lambda \pmod{n^2}) = m\lambda/\lambda = m$

Homomorphic Properties of Paillier cryptosystem:

The Paillier cryptosystem is a partially homomorphic encryption scheme. It allows two types of computation:

- Addition of two ciphertexts.
- Multiplication of ciphertext by a plaintext number.

We majorly uses Addition of two cipher texts property in the applications.

Example:

Key generation:

p = 13
q = 17

n = p * q
n = 13 * 17
n = 221

λ = LCM(p-1, q-1)
λ = LCM(12 , 16)
λ = 48

g ∈ (1,n^2)
Let,

g = 4886

$\mu = (L(g^\lambda \bmod n^2))^{-1}$

μ = [L[4886^48(mod 221^2)]]^-1 (mod221)
μ = [L[30720]]^-1 (mod 221)   {Where, L(x)=(x-1)/n}
μ = [(30720-1)/221]^-1 (mod 221)
μ = (139)^-1 (mod 221)
μ = 159

Encryption:

Message1 = 123
r = 5

C1 = (g^m)*(r^n) (mod n^2)
C1 = (4886^123)*(5^221) (mod 221^2)
C1 = (42021)*(42996) (mod 221^2)
C1 = 8644

∴ Cipher text (C1) is 8644

Message2 = 11
r = 3
C2 = (g^m)*(r^n) (mod n^2)
C2 = (4886^11)*(3^221) (mod 221^2)
C2 = (15450)*(24696) (mod 221^2)
C2 = 7308

∴ Cipher text (C2) is 7308

According to Additive Homomorphism, when two cipher texts are multiplied, then the decryption of the result will be the sum of their plain texts.

C = C1*C2(mod n^2) = 8644*7308 (mod 221^2)
C = 63170352(mod 48841)
C = 18939

Now decrypting the resulted cipher text (C)

Decryption:

M = [L(C^ λ mod n^2)*μ] (mod n)
M = [L(18939^48 (mod 48841))* 159] (mod 221)
M = [L(13703)*159] (mod 221)
M = [62*159] (mod 221)
M = 9858 (mod 221)
M = 134 = 123+11

∴ Hence proved, as product of two cipher texts will result in sum of the plain texts

Applications:

Applications of paillier cryptosystem are:
1. Electronic voting:
Let us suppose an election counting scheme where the number of votes from each center must be encrypted to be sent to the election board. Palliers scheme comes to use in this by sending an aggregated cipher text, consisting of encrypted data from each center . This is further decrypted by the board

Python code:

```
import random

def is_prime(n, k=5):
    if n <= 1:
        return False
    if n <= 3:
        return True
    if n % 2 == 0:
        return False

    # Miller-Rabin primality test
```

```
    def miller_rabin(n, d, r):
        a = random.randint(2, n - 2)
        x = mod_exp(a, d, n)
        if x == 1 or x == n - 1:
            return True
        for i in range(r - 1):
            x = mod_exp(x, 2, n)
            if x == n - 1:
                return True
        return False

    d, r = n - 1, 0
    while d % 2 == 0:
        d //= 2
        r += 1

    for i in range(k):
        if not miller_rabin(n, d, r):
            return False

    return True

def generate_strong_prime(bits):
    while True:
        potential_prime = random.getrandbits(bits)
        if potential_prime % 2 == 0:
            potential_prime += 1
        if is_prime(potential_prime):
            return potential_prime

def gcd(a, b):
    while b:
        a, b = b, a % b
    return a

def lcm(a, b):
    return (a * b) // gcd(a, b)

def extended_gcd(a, b):
    x0, x1, y0, y1 = 1, 0, 0, 1
    while b != 0:
        q, r = divmod(a, b)
        a, b = b, r
        x0, x1 = x1, x0 - q * x1
        y0, y1 = y1, y0 - q * y1
    return a, x0, y0

def multiplicative_inverse(a, n):
```

```
    gcd, x, y = extended_gcd(a, n)
    if gcd != 1:
        raise ValueError(f"The multiplicative inverse does not exist for {a} (mod {n}).")
    else:
        return x % n

def mod_exp(a, b, n):
    result = 1
    a = a % n

    while b > 0:
        if b % 2 == 1:
            result = (result * a) % n

        a = (a * a) % n
        b //= 2

    return result

def text_to_long(text):
    text_bytes = text.encode('utf-8')
    text_long = int.from_bytes(text_bytes, byteorder='big')
    return(text_long)

def long_to_text(long_msg):
    text_bytes = long_msg.to_bytes((long_msg.bit_length() + 7) // 8, byteorder='big')
    text = text_bytes.decode('utf-8')
    return(text)

def generate_keys():
    bits = 1024
    p = generate_strong_prime(bits)
    q = generate_strong_prime(bits)

    n = p*q
    g = 1+n
    lamda = lcm((p-1),(q-1))

    n_squared = n**2
    temp = mod_exp(g, lamda, n_squared)
    temp1 = temp-1
    quotient, remainder = divmod(temp1, n)
    if (remainder!=0):
        raise ValueError(f"Something went wrong")

    mu = multiplicative_inverse(quotient, n)

    public_key = [n, g]
```

```
    private_key = [lamda, mu]

    return(public_key, private_key)

def encrypt_msg(msg, public_key):
    msg_long = text_to_long(msg)
    n = public_key[0]
    g = public_key[1]

    def select_r(n):
        while True:
            r = random.randint(0, n)
            if gcd(r, n) == 1:
                return r

    r = select_r(n)
    n_squared = n**2
    temp = mod_exp(g, msg_long, n_squared)
    temp1 = mod_exp(r, n, n_squared)
    cipher_text = (temp*temp1)%n_squared
    return(cipher_text)

def decrypt_msg(cipher_text, public_key, private_key):
    lamda = private_key[0]
    mu = private_key[1]
    n = public_key[0]
    n_squared = n**2
    temp = mod_exp(cipher_text, lamda, n_squared)
    temp = temp - 1
    quotient, remainder = divmod(temp, n)
    if (remainder!=0):
        raise ValueError(f"Something went wrong")
    plain_long = (quotient*mu)%n
    plain_text = long_to_text(plain_long)
    return(plain_text)

print("Generating Keys....................")
print()
public_key, private_key = generate_keys()
print("n value : ", public_key[0])
print()
print("g value : ", public_key[1])
print()
print("Encryption:")
msg = input("Enter your plain text >> ")
plain_long = text_to_long(msg)
print("Plain text in long : ", plain_long)
cipher_text = encrypt_msg(msg, public_key)
```

```
cipher_hex = hex(cipher_text)
print("Cipher_text : ", cipher_hex)
print()
print("Decryption:")
plain_text_back = decrypt_msg(cipher_text, public_key, private_key)
print("Plain_text : ", plain_text_back)
print("Print Any Key to Exit.........")
input()
```

- **The above code uses g value as n+1**

- def get_random_g(n, lamda):

```
def get_random_g(n, lamda):
    gcd1 = 0
    n_squared = n**2
    while gcd1 != 1:
        g = random.randint(0, n_squared)
        temp = mod_exp(g, lamda, n_squared)
        temp1 = temp-1
        quotient, remainder = divmod(temp1, n)
        gcd1 = gcd(quotient, n)
        if (remainder != 0):
            gcd1 = 0
    return (g)
```

If you add this function then the random g value will be calculated.

UI codes:

HTML Code:

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <link href="https://fonts.googleapis.com/css?family=Poppins" rel="stylesheet">
    <link href="../css/style1.css" rel="stylesheet">
    <title>Paillier Cryptosystem UI</title>
</head>
<body>

    <div class="navbar">
        <h1>Paillier Cryptosystem</h1>
        <button class="Go_back" onclick="window.location.href='../index.html'">Go Back</button>
    </div>
    <div class="rectangle-bar"></div>
    <div class="container">

        <!-- Corrected centering of text -->
        <h1 style="text-align: center;">Calculate here</h1>

        <!-- Key Generation Section -->
        <div class="section">
            <button id="genkeys">Generate Keys</button>
            <div id="keysOutput" class="output"></div>
```

```
    </div>

    <!-- Encryption Section -->
    <div class="section">
      <label for="plaintext">Enter Plaintext:</label>
      <input type="text" id="plaintext" placeholder="Type your message...">
      <button id="enc">Encrypt</button>
      <div id="encryptionOutput" class="output"></div>
    </div>

    <!-- Decryption Section -->
    <div class="section">
      <label for="ciphertext">Enter Ciphertext:</label>
      <input type="text" id="ciphertext" placeholder="Paste your ciphertext...">
      <button id="dec">Decrypt</button>
      <div id="decryptionOutput" class="output"></div>
    </div>
    <script src="../js/script.js"></script>
  </div>
</body>
</html>
```

JavaScript Code:

```
function isPrime(num, k = 5) {
  if (num <= 1n) return false;
  if (num <= 3n) return true;

  // Write (num - 1) as 2^s * d
  let s = 0n;
  let d = num - 1n;
  while (d % 2n === 0n) {
    s++;
    d /= 2n;
  }

  const witness = (a, n) => {
    if (a <= 1n || a >= n - 1n) return false;

    let x = mod_exp(a, d, n);
    if (x === 1n || x === n - 1n) return false;

    for (let i = 1n; i < s; i++) {
      x = mod_exp(x, 2n, n);
      if (x === n - 1n) return false;
    }

    return true;
  };

  for (let i = 0; i < k; i++) {
    const a = getRandomBigInt(2n, num - 1n);
```

```
      if (witness(a, num)) return false;
   }

   return true;
}

function getRandomBigInt(min, max) {
   const range = max - min + 1n;
   const random = BigInt(Math.floor(Math.random() * Number(range)));
   return min + random;
}

function generateRandomBigInt(minDigits, maxDigits) {
   if (minDigits < 1 || maxDigits < minDigits) {
     throw new Error('Invalid digit range');
   }
   const randomDigits = Array.from({ length: Math.floor(Math.random() * (maxDigits - minDigits
+ 1)) + minDigits }, () =>
     Math.floor(Math.random() * 10)
   ).join('');
   const randomBigInt = BigInt(randomDigits);
   return randomBigInt;
}

function generatePrimeWithDigits(min, max) {
   while (true){
      let primeCandidate = generateRandomBigInt(min, max);

      if (primeCandidate % 2n === 0) {
         primeCandidate++;
      }

      if (isPrime(primeCandidate)) {
         return primeCandidate;
      }
   }
}

function mod_exp(base, exponent, modulus) {
   if (modulus === 1n) return 0n;
   let result = 1n;
   base = base % modulus;
   while (exponent > 0n) {
      if (exponent % 2n === 1n) {
         result = (result * base) % modulus;
      }
      exponent = exponent >> 1n;
      base = (base * base) % modulus;
```

```
    }
    return result;
}
function modExp(a, b, n) {
    a = BigInt(a);
    b = BigInt(b);
    n = BigInt(n);

    let result = BigInt(1);
    a = a % n;

    while (b > 0n) {
        if (b % 2n === 1n) {
            result = (result * a) % n;
        }
        a = (a * a) % n;
        b = b / 2n;
    }

    return result;
}
function gcd(a, b) {
    return b === 0n ? a : gcd(b, a % b);
}

function lcm(a, b) {
    return (a * b) / gcd(a, b);
}

function extendedGCD(a, b) {
    if (b === 0n) {
        return [a, 1n, 0n];
    } else {
        const [d, x, y] = extendedGCD(b, a % b);
        return [d, y, x - y * (a / b)];
    }
}

function multiplicativeInverse(a, n) {
    const [g, x, y] = extendedGCD(a, n);

    if (g !== 1n) {
        throw new Error("Inverse does not exist");
    }

    return (x % n + n) % n;
}
```

```
function textToLong(text) {
   const encoder = new TextEncoder();
   const textBytes = encoder.encode(text);
   const textLong = BigInt('0x' + Array.from(textBytes).map(byte => byte.toString(16).padStart(2,
'0')).join(''));
   return textLong;
}

function longToText(longMsg) {
   const hexString = longMsg.toString(16).padStart((longMsg.toString(16).length + 1) & ~1, '0');
   const hexArray = hexString.match(/.{2}/g);

   if (!hexArray) {
      throw new Error('Invalid hex string');
   }

   const bytes = Uint8Array.from(hexArray.map(byte => parseInt(byte, 16)));
   const decoder = new TextDecoder('utf-8');
   return decoder.decode(bytes);
}

function divmod(a, b) {
   const quotient = a / b;
   const remainder = a % b;
   return [quotient, remainder];
}

function generateTwoPrimesWithDigits(min, max) {
   const prime1 = generatePrimeWithDigits(min, max);
   console.log("Prime 1 : ",prime1);
   const prime2 = generatePrimeWithDigits(min, max);
   console.log("Prime 2 : ",prime2);
   return{prime1, prime2}
}

function generate_random_g(n,lamda){
   let gcd1 = 0;
   let n_square = n ** 2n;
   const numberOfDigits = n_square.toString().length;
   let g = 1n;
   while (gcd!==1n){
      g = generateRandomBigInt(2, numberOfDigits-1);
      const temp = mod_exp(g, lamda, n_square);
      const temp1 = temp - 1n;
      const [quotient, remainder] = divmod(temp1, n);
      gcd1 = gcd(quotient, n);
      if (remainder!==0n){
        gcd1 = 0;
```

```
      }
      if (gcd1===1n){
        break;
      }
    }
    return(g);
  }

function generateKeys(){
    const { prime1, prime2 } = generateTwoPrimesWithDigits(90, 100);
    const prime_1 = BigInt(prime1);
    const prime_2 = BigInt(prime2);
    const n = prime_1*prime_2;
    const lamda = lcm(prime_1 - 1n, prime_2 - 1n);
    const g = generate_random_g(n,lamda);

    const n_squared = n ** 2n;
    const temp = mod_exp(g, lamda, n_squared);
    const temp1 = temp - 1n;

    const [quotient, remainder] = divmod(temp1, n);
    if (remainder !== 0n) {
        throw new Error("Something went wrong");
    }

    const mu = multiplicativeInverse(quotient, n);
    return[n,g,lamda,mu];
}

function encrypt(n,g,plaintext) {
    //const plaintext = document.getElementById('plaintext').value;
    const plain = textToLong(plaintext);
    const plain_long = BigInt(plain);
    console.log("Given Text in integer format : ",plain_long);

    let r = generatePrimeWithDigits(5, 10);
    let n_squared = n * n;
    let temp = modExp(g, plain_long, n_squared);
    let temp1 = modExp(r, n, n_squared);
    let ciphertext = (temp * temp1) % n_squared;
    let cipherhex = ciphertext.toString(16);
    return(cipherhex);
}

function decrypt(n,lamda,mu,cipherhex){
    const ciphertext = BigInt("0x" + cipherhex);
    let n_squared = n * n;
    let temp = modExp(ciphertext,lamda,n_squared);
```

```
    temp = temp - 1n;
    const [quotient, remainder] = divmod(temp, n);
    if (remainder !== 0n) {
        throw new Error("Something went wrong");
    }
    let plain_long = (quotient*mu)%n;
    let plaintext = longToText(plain_long);
    return(plaintext);
}

let n,g,lamda,mu;

document.getElementById('genkeys').addEventListener('click', async function () {
    try {
        [n, g, lamda, mu] = await generateKeys();

        const keysOutput = `
            <p>Public Key (n, g): (${n}, ${g})</p>
            <p>Private Key (lambda, mu): (${lamda}, ${mu})</p>
        `;

        document.getElementById('keysOutput').innerHTML = keysOutput;
    } catch (error) {
        console.error(error);
        document.getElementById('keysOutput').innerHTML = '<p>Error generating keys</p>';
    }
});

document.getElementById('enc').addEventListener('click', async function () {
    const plaintext = document.getElementById('plaintext').value;
    const ciphertext = await encrypt(n,g,plaintext);
    document.getElementById('encryptionOutput').innerHTML = `
        <p>Ciphertext: ${ciphertext}</p>
    `;
    document.getElementById('ciphertext').value= ciphertext
});

document.getElementById('dec').addEventListener('click', async function () {
    const ciphertext = document.getElementById('ciphertext').value;
    const plaintext = await decrypt(n,lamda,mu,ciphertext);
    document.getElementById('decryptionOutput').innerHTML = `
        <p>Plaintext: ${plaintext}</p>
    `;
});
```

Application Codes:

HTML Code:

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <link href="https://fonts.googleapis.com/css?family=Poppins" rel="stylesheet">
    <link href="../css/application.css" rel="stylesheet">
    <title>Application | Paillier Cryptosystem</title>

</head>
<body>
    <div class="navbar">
        <h1>Paillier Cryptosystem</h1>
        <button class="Go_back" onclick="window.location.href='../index.html'">Go Back</button>
    </div>
    <div class="rectangle-bar"></div>
    <div class="section1">
        <label for="reg1">Enter number of votes in region 1:</label>
        <input type="number" id="reg1" min="0"><br><br>
        <button id="enc1" style="font-size: 1vw;">Encrypt</button>
        <div id="encryptionOutput1"></div><br><br><br><br><br><br><br>
    </div>
    <div class="section2">
        <label for="reg2">Enter number of votes in region 2:</label>
        <input type="number" id="reg2" min="0"><br><br>
        <button id="enc2" style="font-size: 1vw;">Encrypt</button>
        <div id="encryptionOutput2"></div><br><br><br><br><br><br><br>
    </div>
    <div class="section3">
        <div id="mulcipher" style="width: 40vw; overflow-wrap:break-word;"></div><br>
        <button id="dec" style="font-size: 1vw; position: relative; left: 43%;">Decrypt</button><br>
        <div id="decryptionOutput"></div>
    </div>
    <script src="../js/script2.js"></script>
</body>
</html>
```

JavaScript Code:

```
function isPrime(num, k = 5) {
    if (num <= 1n) return false;
    if (num <= 3n) return true;

    // Write (num - 1) as 2^s * d
    let s = 0n;
    let d = num - 1n;
    while (d % 2n === 0n) {
        s++;
        d /= 2n;
    }

    const witness = (a, n) => {
        if (a <= 1n || a >= n - 1n) return false;
```

```
        let x = mod_exp(a, d, n);
        if (x === 1n || x === n - 1n) return false;

        for (let i = 1n; i < s; i++) {
            x = mod_exp(x, 2n, n);
            if (x === n - 1n) return false;
        }

        return true;
    };

    for (let i = 0; i < k; i++) {
        const a = getRandomBigInt(2n, num - 1n);
        if (witness(a, num)) return false;
    }

    return true;
}

function getRandomBigInt(min, max) {
    const range = max - min + 1n;
    const random = BigInt(Math.floor(Math.random() * Number(range)));
    return min + random;
}

function generateRandomBigInt(minDigits, maxDigits) {
    if (minDigits < 1 || maxDigits < minDigits) {
      throw new Error('Invalid digit range');
    }
    const randomDigits = Array.from({ length: Math.floor(Math.random() * (maxDigits - minDigits + 1)) +
minDigits }, () =>
      Math.floor(Math.random() * 10)
    ).join('');
    const randomBigInt = BigInt(randomDigits);
    return randomBigInt;
}

function generatePrimeWithDigits(min, max) {
    while (true){
        let primeCandidate = generateRandomBigInt(min, max);

        if (primeCandidate % 2n === 0) {
            primeCandidate++;
        }

        if (isPrime(primeCandidate)) {
            return primeCandidate;
        }
    }
}

function mod_exp(base, exponent, modulus) {
    if (modulus === 1n) return 0n;
    let result = 1n;
    base = base % modulus;
    while (exponent > 0n) {
        if (exponent % 2n === 1n) {
            result = (result * base) % modulus;
        }
```

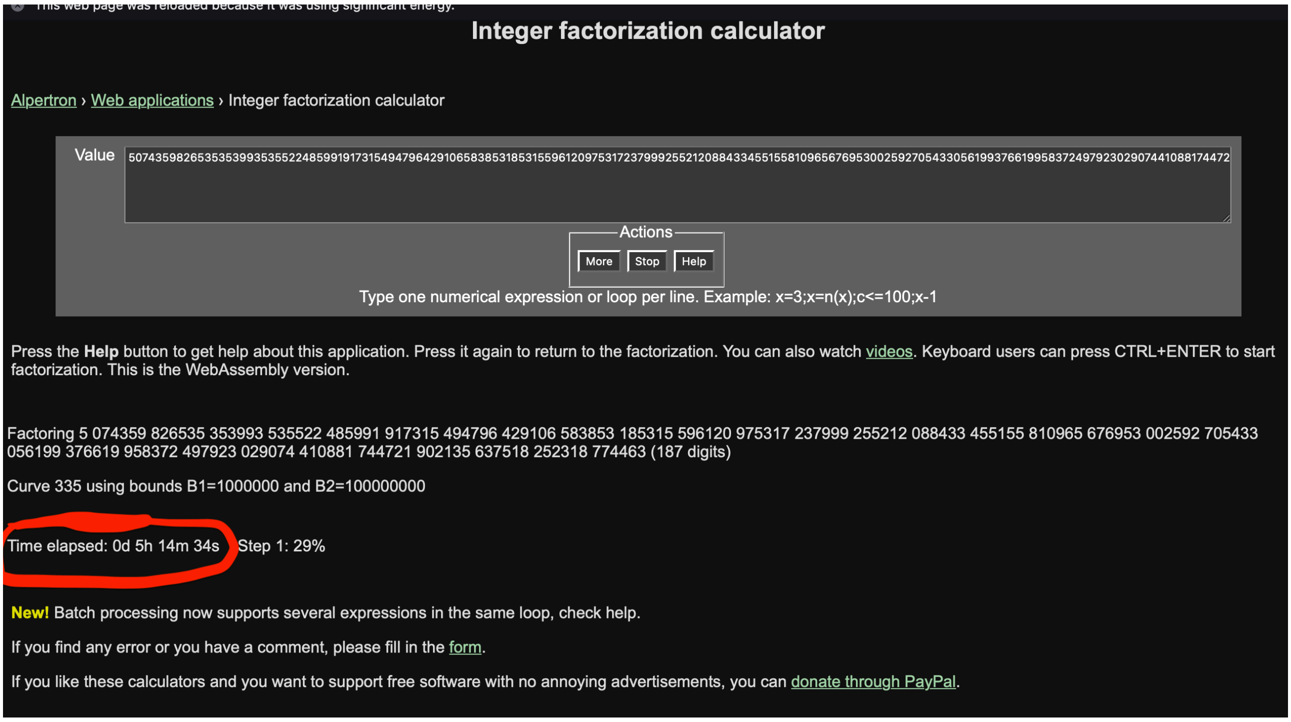
```
        exponent = exponent >> 1n;
        base = (base * base) % modulus;
    }
    return result;
}
function modExp(a, b, n) {
    a = BigInt(a);
    b = BigInt(b);
    n = BigInt(n);

    let result = BigInt(1);
    a = a % n;

    while (b > 0n) {
        if (b % 2n === 1n) {
            result = (result * a) % n;
        }
        a = (a * a) % n;
        b = b / 2n;
    }

    return result;
}
function gcd(a, b) {
    return b === 0n ? a : gcd(b, a % b);
}

function lcm(a, b) {
    return (a * b) / gcd(a, b);
}

function extendedGCD(a, b) {
    if (b === 0n) {
        return [a, 1n, 0n];
    } else {
        const [d, x, y] = extendedGCD(b, a % b);
        return [d, y, x - y * (a / b)];
    }
}

function multiplicativeInverse(a, n) {
    const [g, x, y] = extendedGCD(a, n);

    if (g !== 1n) {
        throw new Error("Inverse does not exist");
    }

    return (x % n + n) % n;
}

function textToLong(text) {
    const encoder = new TextEncoder();
    const textBytes = encoder.encode(text);
    const textLong = BigInt('0x' + Array.from(textBytes).map(byte => byte.toString(16).padStart(2,
'0')).join(''));
    return textLong;
}

function longToText(longMsg) {
    const hexString = longMsg.toString(16).padStart((longMsg.toString(16).length + 1) & ~1, '0');
```

```
    const hexArray = hexString.match(/.{2}/g);

    if (!hexArray) {
        throw new Error('Invalid hex string');
    }

    const bytes = Uint8Array.from(hexArray.map(byte => parseInt(byte, 16)));
    const decoder = new TextDecoder('utf-8');
    return decoder.decode(bytes);
}

function divmod(a, b) {
    const quotient = a / b;
    const remainder = a % b;
    return [quotient, remainder];
}

function generateTwoPrimesWithDigits(min, max) {
    const prime1 = generatePrimeWithDigits(min, max);
    console.log("Prime 1 : ",prime1);
    const prime2 = generatePrimeWithDigits(min, max);
    console.log("Prime 2 : ",prime2);
    return{prime1, prime2}
}

function generate_random_g(n,lamda){
    let gcd1 = 0;
    let n_square = n ** 2n;
    const numberOfDigits = n_square.toString().length;
    let g = 1n;
    while (gcd!==1n){
        g = generateRandomBigInt(2, numberOfDigits-1);
        const temp = mod_exp(g, lamda, n_square);
        const temp1 = temp - 1n;
        const [quotient, remainder] = divmod(temp1, n);
        gcd1 = gcd(quotient, n);
        if (remainder!==0n){
          gcd1 = 0;
        }
        if (gcd1===1n){
          break;
        }
    }
    return(g);
  }

function generateKeys(){
    const { prime1, prime2 } = generateTwoPrimesWithDigits(10, 15);
    const prime_1 = BigInt(prime1);
    const prime_2 = BigInt(prime2);
    const n = prime_1*prime_2;
    const lamda = lcm(prime_1 - 1n, prime_2 - 1n);
    const g = generate_random_g(n,lamda);

    const n_squared = n ** 2n;
    const temp = mod_exp(g, lamda, n_squared);
    const temp1 = temp - 1n;

    const [quotient, remainder] = divmod(temp1, n);
    if (remainder !== 0n) {
```

```
        throw new Error("Something went wrong");
    }

    const mu = multiplicativeInverse(quotient, n);
    return[n,g,lamda,mu];
}

function encrypt(n,g,plaintext) {
    let r = generatePrimeWithDigits(5, 10);
    let n_squared = n * n;
    let temp = modExp(g, plaintext, n_squared);
    let temp1 = modExp(r, n, n_squared);
    let ciphertext = (temp * temp1) % n_squared;
    return(ciphertext);
}

function decrypt(n,lamda,mu,ciphertext){
    let n_squared = n * n;
    let temp = modExp(ciphertext,lamda,n_squared);
    temp = temp - 1n;
    const [quotient, remainder] = divmod(temp, n);
    if (remainder !== 0n) {
        throw new Error("Something went wrong");
    }
    let plain_long = (quotient*mu)%n;
    return(plain_long);
}

let n,g,lamda,mu,ciphertext1,ciphertext2,mulciphertext;


try {
    [n, g, lamda, mu] = generateKeys();
    console.log("n: ",n);
    console.log("g: ",g);
    console.log("lamda: ",lamda);
    console.log("mu: ",mu);
} catch (error) {
    console.error(error);
}

document.getElementById('enc1').addEventListener('click', async function () {
    const plaintext = document.getElementById('reg1').value;
    ciphertext1 = await encrypt(n,g,plaintext);
    document.getElementById('encryptionOutput1').innerHTML = `
        <p>Ciphertext: ${ciphertext1}</p>
    `;
});

document.getElementById('enc2').addEventListener('click', async function () {
    const plaintext = document.getElementById('reg2').value;
    ciphertext2 = await encrypt(n,g,plaintext);
    document.getElementById('encryptionOutput2').innerHTML = `
        <p>Ciphertext: ${ciphertext2}</p>
    `;
    mulciphertext = ciphertext1*ciphertext2;
    document.getElementById('mulcipher').innerHTML=`
        <p>Resultant Ciphertext: ${mulciphertext}</p>
    `

});
```

```
document.getElementById('dec').addEventListener('click', async function () {
    const plaintext = await decrypt(n,lamda,mu,mulciphertext);
    document.getElementById('decryptionOutput').innerHTML = `
        <p>Total Votes: ${plaintext}</p>
    `;
});
```

Hard problem:



Here, online factoriser runner for 5hrs but its unable to factorise the key generated by the algorithm. So, hard problem of the paillier is factorization