Warsaw University of Technology

FACULTY OF
ELECTRONICS AND INFORMATION TECHNOLOGY

ECRYP - Cryptography and Information Security

ElGamal cipher (software implementation - 2 ciphering modes - ECB and CBC)

Marco Campione K-6254
Hasan Senyurt K-6644

Supervisor: Andrzej Wojeński
WARSAW May 23, 2022

Contents

1. Introduction

1.1. What is a cipher

Ciphers, also called encryption algorithms, are systems for encrypting and decrypting data. A cipher converts the original message, called plaintext, into ciphertext using a key to determine how it is done. Ciphers are generally categorized according to how they work and by how their key is used for encryption and decryption. Block ciphers accumulate symbols in a message of a fixed size (the block), and stream ciphers work on a continuous stream of symbols. When a cipher uses the same key for encryption and decryption, they are known as symmetric key algorithms or ciphers. Asymmetric key algorithms or ciphers use a different key for encryption/decryption.

1.2. Elgamal cipher

This project concerns the ElGamal cipher. ElGamal cryptosystem can be defined as the cryptography algorithm that uses the public and private key concepts to secure communication between two systems. It can be considered the asymmetric algorithm where the encryption and decryption happen by using public and private keys. In order to encrypt the message, the public key is used by the client, while the message could be decrypted using the private key on the server end. This is considered an efficient algorithm to perform encryption and decryption as the keys are extremely tough to predict.

1.3. Block ciphers

Block ciphers work on a fixed-length segment of plaintext data, typically a 64-bit or 128-bit block as input, and outputs a fixed length ciphertext. The message is broken into blocks, and each block is encrypted through a substitution process. Where there is insufficient data to fill a block, the blank space will be padded prior to encryption. The resulting ciphertext block is usually the same size as the input plaintext block.

1.4. ECB (Electronic Code Book)

The Electronic Code Book (ECB) mode uses simple substitution, making it one of the easiest and fastest algorithms to implement. The input plaintext is broken into several blocks and encrypted individually using the key. This allows each encrypted block to be decrypted individually. Encrypting the same block twice will result in the same ciphertext being returned twice.

1.5. CBC (Cipher Block Chaining)

In Cipher Block Chaining (CBC) mode, the first block of the plaintext is exclusive-OR'd (XOR'd), which is a binary function or operation that compares two bits and alters the

output with a third bit, with an initialization vector (IV) prior to the application of the encryption key. The IV is a block of random bits of plaintext. The resultant block is the first block of the ciphertext. Each subsequent block of plaintext is then XOR'd with the previous block of ciphertext prior to encryption, hence the term "chaining." Due to this XOR process, the same block of plaintext will no longer result in identical ciphertext being produced. Decryption in the CBC mode works in the reverse order. After decrypting the last block of ciphertext, the resultant data is XOR'd with the previous block of ciphertext to recover the original plaintext. The CBC mode is used in hash algorithms. Discarding all previous blocks, the last resulting block is retained as the output hash when used for this purpose.

2. Algorithm

In order to make it simpler and clearer we will use an example of Bob and Alice. We suppose that Alice wants to communicate with Bob. In ElGamal, only the receiver needs to create a key in advance and publish it. Let's look at Bob's procedure of key generation.

2.1. Key generation

To generate his private key and his public key Bob does the following:

- **Prime and group generation**: Bob needs to select a large prime p and the generator g of a multiplicative group Z_p^* of the integers modulo p.
- **Private key selection**: Bob selects an integer x from the group Z at random and with the constraint $1 \leq x \leq p-1$.
- **Public key assembly**: We calculate the public key part $y = g^x (\mathrm{mod}\ p)$. In ElGamal, the public key of Bob is the triplet (p, g, y) and his private key is x.
- **Public key publishing**: Bob must give this public key to Alice using a dedicated key server or other means.

To encrypt a plaintext message to Bob, Alice needs to get the public key. Our private key x is sent in y. The assumption that it is infeasible to compute using discrete logarithm means that this is safe. Let's look at Alice's plaintext message encryption.

2.2. Encryption

To encrypt a message Alice uses Bob's public key :

- **Obtain public key**: Alice acquires public key (p, g, y) from Bob.
- **Prepare M for encoding**: Prepare message M (to send) as set of integers $m_1, m_2, \ldots$ in the range of $\{1, \ldots, p-1\}$ These integers will be encoded one by one.
- **Select random exponent**: Alice selects a random exponent k that's takes place of second party's private exponent.
- **Compute public key**: To transmit k to Bob, Alice computes $a = g^k (\mathrm{modp})$ and combines it with the ciphertext to be sent to Bob.
- **Encrypt the plaintext**: Alice encrypts message M to ciphertext C. To do this, she iterates over $m_1, m_2, \ldots$ and for each m_i :

$$c_i = m_i * \left(g^x\right)^k \rightarrow c_i = m_i * y^k$$

The ciphertext C is the set of all c_i with $0 \leq i \leq |M|$.
The resulted message C is sent to Bob along with public key $a = g^k (\mathrm{mod}\ p)$. If an attacker listens to this transmission and acquires the public key part g^x of Bob, he would still not be able to derive g^{x*k} due to the discrete logarithm problem. ElGamal advises to use a

new random k for each of the single message blocks m_i, which would lead to much higher security.

2.3. Decryption

After receiving the encrypted message C and the randomized public key g^k, Bob must use the encryption algorithm to read plaintext message M. Let's look at Bob's ciphertext decryption algorithm:

- **Compute shared key**: ElGamal cryptosystem allows Alice to define a shared secret key without Bob's interaction. This is from Bob's private exponent x and Alice's random exponent k. The shared key is defined as follows: $\left(g^k\right)^{-x}$ = an inverse in G of g^{kx}.
- **Description**: For each ciphertext parts c_i in C, Bob computes the plaintext using $m_i = \left(g^k\right)^{-x} * c_i$(modp) He can read the message M sent by Alice by combining m_i.

2.4. Example

This example is made in order to better understand how ElGamal cryptosystem works.

Key generation :

1. A selects a prime p = 7187 and generator g = 754 of Z^*_{7187}
2. A chooses private key x = 147 and computes $g^x(mod\, p)$ = 754147 (mod 7187) = 6966
3. A sends public key (p = 7187, g = 754, g^x = 6966) to B.

Encryption :

1. To encrypt message m = 36, B selects random integer k = 55.
2. B computes $a = g^x (\mathrm{mod}\, p) \rightarrow a = 75455 (\mathrm{mod}\, 7187) \equiv 1571$
3. $c = m * (g^x)^k \rightarrow 36 * 696699 (\mathrm{mod}\, 7187) \equiv 6501$
4. B sends a = 1571 and c = 6501 to A.

Decryption :

1. A computes $m_{\text{plain}} = c/(a^x) (\mathrm{mod}\, p) \rightarrow m = c/(g^k)^x (\mathrm{mod}\, p) \rightarrow m = (g^k)^{-x} * c (\mathrm{mod}\, p) \rightarrow 36$. It satisfies the formula:

$$(g^k)^{-x} * c_i (\mathrm{mod}\, p) \equiv g^{-kx} * m * g^{xk} \equiv m (\mathrm{mod}\, p), \text{ since } g^{-kx} g^{xk} = 1.$$

2.5. ECB

To have a better idea on how this algorithm works, here a drawing of the *ECB* algorithm for for encryption and decryption.

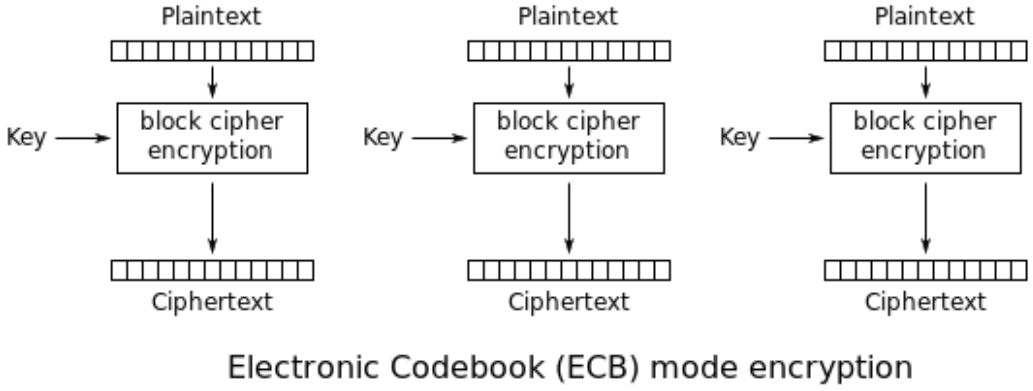

Figure 2.1. ECB encryption

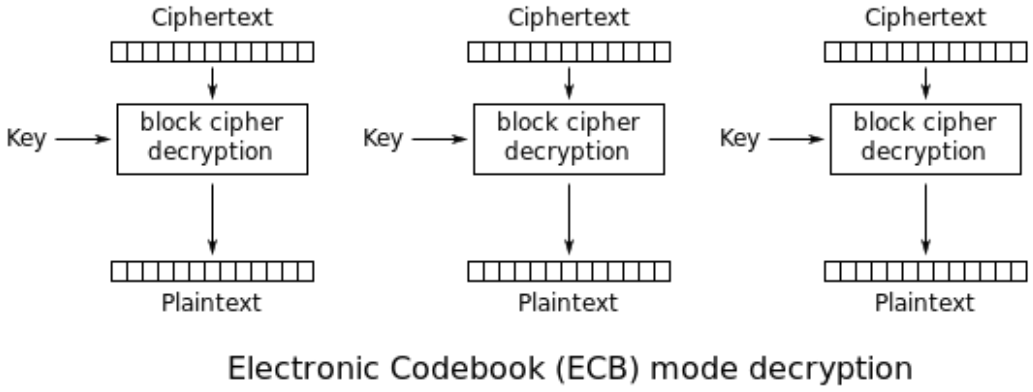

Figure 2.2. ECB decryption

2.6. CBC

To have a better idea on how this algorithm works, here a drawing of the *CBC* algorithm for for encryption and decryption.

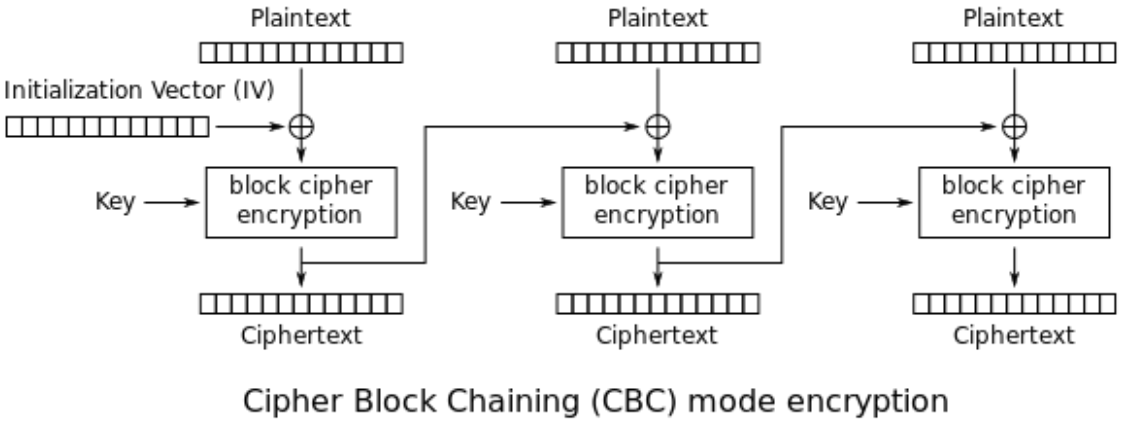

Figure 2.3. CBC encryption

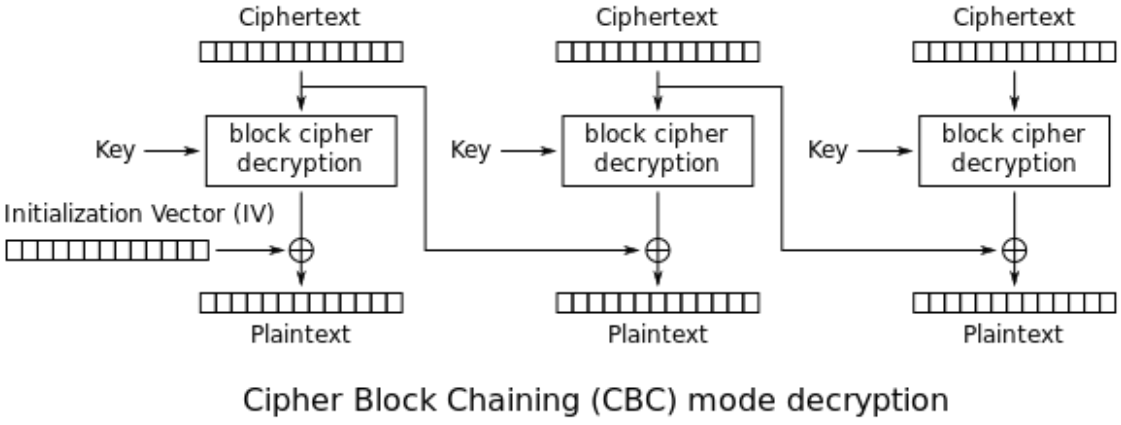

Figure 2.4. CBC denryption

3. Code

3.1. Code

Simple code explanation: Our program uses console mode as simple graphical user interface. There are two inputs which are plain text message (string) and number to choose ciphering mode (integer) from the user. There are two types of output which are string and integers.
Integer outputs are:

- field
- generator
- public key
- generator to the power of public key
- private key
- generator to the power of private key
- generator to the power of private key times public key
- encrypted message (ciphertext) in integer array

String output is:

- decrypted message

```python
#!/usr/bin/env python3
import random
from math import pow


#Encryption function which multiplies message (int) with g^(ab).
def encrypted_elgamal(message, g_power_ab):
    return (g_power_ab * message)

#Decryption function which divides encrypted message with g^(ab).
def decrypted_elgamal(encrypted_message, g_power_ab):
    return (encrypted_message // g_power_ab)


#Modulo function for the numbers which has power.
def power(base, power, mod):
    x = 1
    y = base

    while power > 0:
        if power % 2 != 0:
            x = (x * y) % mod
        y = (y * y) % mod
        power = int(power / 2)
    return x % mod


#Encryption function for ECB ciphering mode. It converts string char message to int and encrypts to ciphertext.
def ECB_encrypt(message, g_power_ab):

    cipher_text = []
    for i in range(0, len(message)):
        cipher_text.append(message[i])

    #array for ciphertext
    for i in range(0, len(cipher_text)):
        cipher_text[i] = encrypted_elgamal(ord(cipher_text[i]), g_power_ab) #plaintext to ciphertext
    return cipher_text

#Decryption function for ECB ciphering mode. It decrypts ciphertext to int and converts to char text. (plaintext)
def ECB_decrypt(encrypted_message, key):

    decrypted_message = []
    #array for plaintext
    for i in range(0, len(encrypted_message)):
        decrypted_message.append(chr(decrypted_elgamal(encrypted_message[i], key))) #ciphertext to plaintext

    return decrypted_message

#Encryption function for CBC ciphering mode. It makes XOR operation between first item of message and initialization vector.
#After, It encrypts the item and next item will be result of opeartion which is XOR between encrypted item and the next item of the list.
def CBC_encrypt(plain_text, key, iv):
```

```
53
54      c = []
55      for i in range(0, len(plain_text)):
56          c.append(ord(plain_text[i])) #character to integer
57
58      c[0] = c[0] ^ iv # XOR'ing with initialization vector
59
60      for i in range(0, (len(c) - 1)):
61          c[i] = encrypted_elgamal(c[i], key)
62          c[i+1] = c[i+1] ^ c[i]
63
64      c[len(c) - 1] = encrypted_elgamal(c[len(c) - 1], key)
65      return c
66
67  #Decryption function for CBC ciphering mode. It makes the opposite operation of
    CBC_encrpyt function to get plaintext message.
68  def CBC_decrypt(cipher_text, key, iv):
69
70      decrypted_message = []
71      plain_text = []
72
73      for i in range(0, len(cipher_text)):
74          decrypted_message.append(decrypted_elgamal(cipher_text[i], key))
75
76      plain_text.append(chr(iv ^ decrypted_message[0])) # XOR'ing with initialization
    vector
77
78      for i in range(1, len(cipher_text)):
79          plain_text.append(chr(cipher_text[i-1] ^ decrypted_message[i]))
80
81      return plain_text
82
83  #Function for calculating gcd.
84  def gcd(a, b):
85
86      if a < b:
87          return gcd(b, a)
88      elif a % b == 0:
89          return b
90      else:
91          return gcd(b, a % b)
92
93
94  #Function for checking whether number is prime or not.
95  def check_prime(number):
96      control = False
97      for i in range(2,number):
98          if number % i == 0:
99              control = False
100             break
101         else:
102             control = True
103
104     return control
105
106 #Function for creating generator.
107 def generator(field):
108     while(True):
```

```
109         results_mod = [] #array of possible generator ^ x (mod field) x={1,2,3...}
110         possible_generator = random.randint(2,field)
111 
112         for i in range(1,field):
113             results_mod.append(power(possible_generator,i,field))
114 
115         #checking if there is same numbers inside the array. If it not, then this is
    our generator.
116         if(len(results_mod) == len(set(results_mod))):
117             break
118     return possible_generator
119 
120 
121 #Function for generating key.
122 def gen_key(p):
123     key = random.randint(pow(10, 2), p)
124 
125     #checking gcd
126     while gcd(p, key) != 1:
127         key = random.randint(pow(10, 2), p)
128 
129     return key
130 
131 
132 def main():
133     print("Please enter a string to be encrypted: ")
134     message = input() #FIRST INPUT FOR THE PROGRAM: plaintext message.
135 
136     #choosing prime number between 10^2-10^5.
137     p = random.randint(pow(10, 2), pow(10, 5))
138     while(True):
139         if check_prime(p) == True:
140             break
141         else:
142             p = random.randint(pow(10, 2), pow(10, 5))
143 
144     #choosing generator in the field p.
145     g = generator(p)
146     print("\nIn the field: ", p, "\nwe found a the generator: ", g)
147 
148 
149     rec_key = gen_key(p) # Public key
150     g_power_a = power(g, rec_key, p)
151 
152     print("the public key: ", rec_key)
153     print("g to the power of a: ", g_power_a)
154 
155     send_key = gen_key(p)# Private key
156     g_power_b = power(g, send_key, p)
157 
158     print("the private key: ", send_key)
159     print("g to the power b: ", g_power_b)
160 
161 
162     g_power_ab = power(g_power_a, send_key, p)
163     print("g to the power of ab:", g_power_ab, "\n")
164 
165     while(True):
```

```
166         print("Please choose a ciphering mode:\n1 ECB\n2 CBC\n3 Exit program")
167         cipher_mode = input() #SECOND INPUT OF THE PROGRAM: choosing the ciphering
    mode.
168 
169         if cipher_mode == "1":
170 
171         # # ECB
172             #encryption of the message.
173             cipher_text = ECB_encrypt(message, g_power_ab)
174             print("We used ECB cipher mode to break up the message, into smaller
    parts, and encrypt every part individually")
175             print("This is our encrypted message: ", cipher_text, "\n")
176 
177             #decryption of the message.
178             plain_text = ECB_decrypt(cipher_text, g_power_ab)
179             dmsg = ''.join(plain_text)
180             print("This is our decrypted message: ", dmsg)
181             break
182 
183         elif cipher_mode == '2':
184 
185         # # CBC
186             #declaring initialization vector as random.
187             iv = random.randint(1, 2000000)
188             #encryption of the message.
189             cipher_text = CBC_encrypt(message, g_power_ab, iv)
190 
191             print("We used CBC cipher mode to break up the message, into smaller
    parts, and encrypt every part individually")
192             print("This is our encrypted message: ", cipher_text, "\n")
193 
194             #decryption of the message.
195             plain_text = CBC_decrypt(cipher_text, g_power_ab, iv)
196             dmsg = ''.join(plain_text)
197             print("This is our decrypted message: ",dmsg)
198             break
199 
200         elif cipher_mode == '3':
201             break
202         else:
203             print("Please enter 1 or 2 to choose a ciphering mode.\n")
204 
205 
206 if __name__ == "__main__":
207     main()
```

3.2. Testing

After the many simulations we conducted, we noticed that the ciphering algorithms we coded are working in all the situation tested, below two examples of the result obtained during the simulations both for ECB and CBC.

3.2.1. ECB

This is the result that we obtained with the ECB ciphering mode:

```
Please enter a string to be encrypted:
This is a message from Marco to Hasan

In the field:  43411
we found a the generator:  17537
the public key:  8651
g to the power of a:  11608
the private key:  2104
g to the power b:  27637
g to the power of ab: 30877

Please choose a ciphering mode:
1 ECB
2 CBC
3 Exit program
1
We used ECB cipher mode to break up the message, into smaller parts, and encrypt every part individually
This is our encrypted message:  [2593668, 3211208, 3242085, 3550855, 988064, 3242085, 3550855, 988064, 2995069, 988064, 3365593, 3118577, 3550855, 3550855, 2995069, 3180331, 3118577, 988064, 3149454,
3519978, 3427347, 3365593, 988064, 2377529, 2995069, 3519978, 3056823, 3427347, 988064, 3581732, 3427347, 988064, 2223144, 2995069, 3550855, 2995069, 3396470]

This is our decrypted message:  This is a message from Marco to Hasan
```

Figure 3.1. First test with a short text

```
Please enter a string to be encrypted:
This is a project for ECRYP (Cryptography and Information Security) course.

In the field:  49261
we found a the generator:  5395
the public key:  23607
g to the power of a:  13152
the private key:  32968
g to the power b:  43390
g to the power of ab: 46001

Please choose a ciphering mode:
1 ECB
2 CBC
3 Exit program
1
We used ECB cipher mode to break up the message, into smaller parts, and encrypt every part individually
This is our encrypted message:  [3864084, 4784104, 4830105, 5290115, 1472032, 4830105, 5290115, 1472032, 4462097, 1472032, 5152112, 5244114, 5106111, 4876106, 4646101, 4554099, 5336116, 1472032, 46921
02, 5106111, 5244114, 1472032, 3174069, 3082067, 3772082, 4094089, 3680080, 1472032, 1840040, 3082067, 5244114, 5566121, 5152112, 5336116, 5106111, 4738103, 5244114, 4462097, 5152112, 4784104, 5566121
, 1472032, 4462097, 5060110, 4600100, 1472032, 3358073, 5060110, 4692102, 5106111, 5244114, 5014109, 4462097, 5336116, 4830105, 5106111, 5060110, 1472032, 3818083, 4646101, 4554099, 5382117, 5244114,
4830105, 5336116, 5566121, 1886041, 1472032, 4554099, 5106111, 5382117, 5244114, 5290115, 4646101, 2116046, 1472032]

This is our decrypted message:  This is a project for ECRYP (Cryptography and Information Security) course.
```

Figure 3.2. Second test with a longer text

3.2.2. CBC

This is the result that we obtained with the *CBC* ciphering mode:

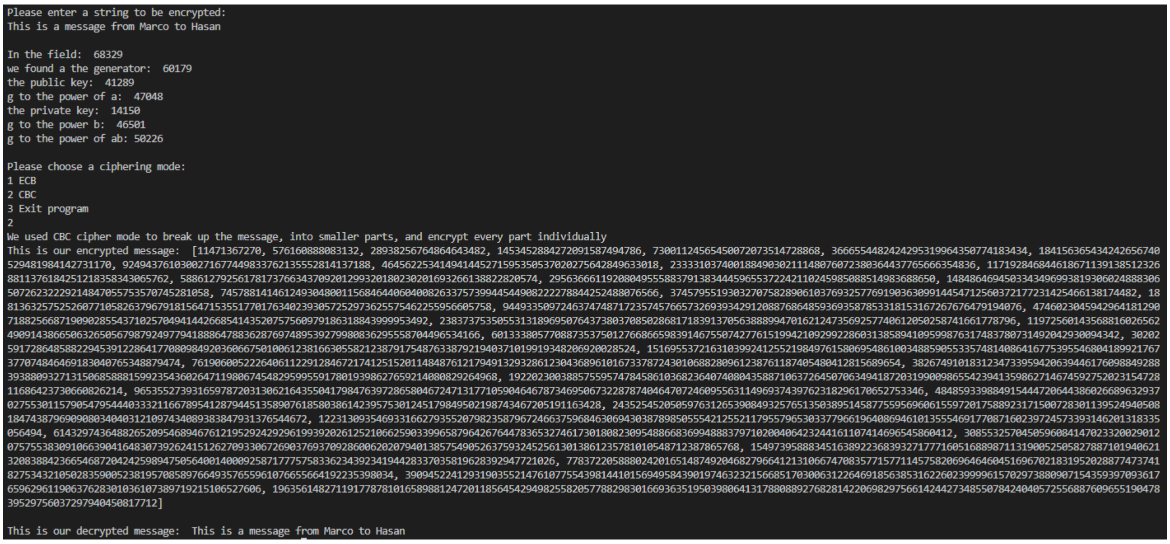

Figure 3.3. First test with a short text

Figure 3.4. Second test with a longer text