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 28, 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.

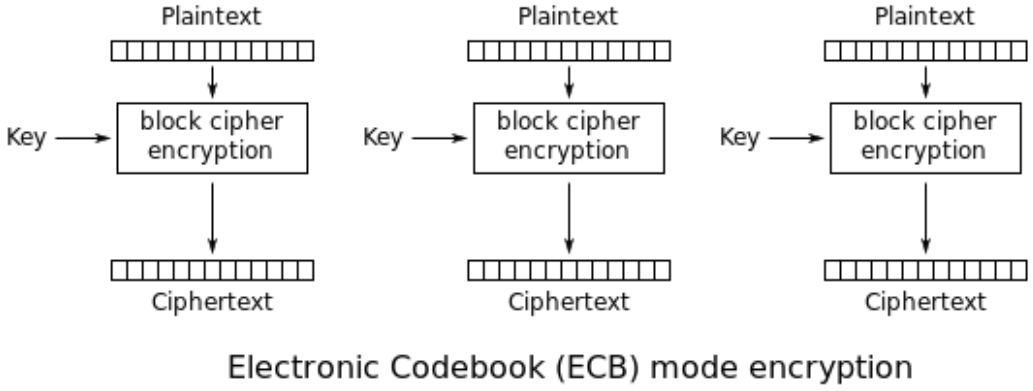

Figure 2.1. ECB encryption

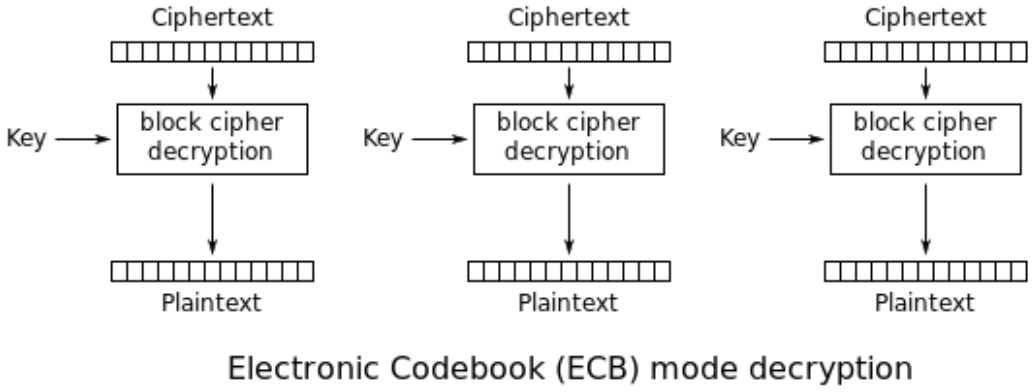

Figure 2.2. ECB decryption

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.

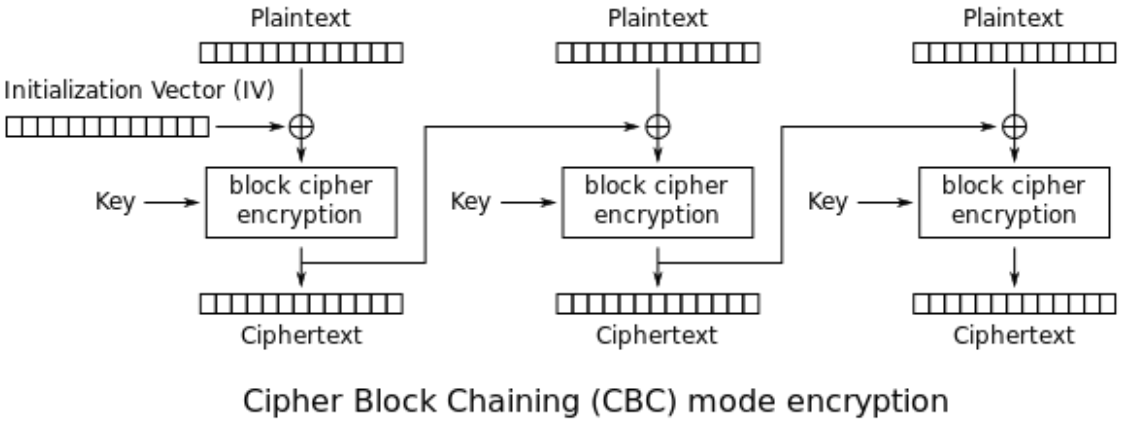

Figure 2.3. CBC encryption

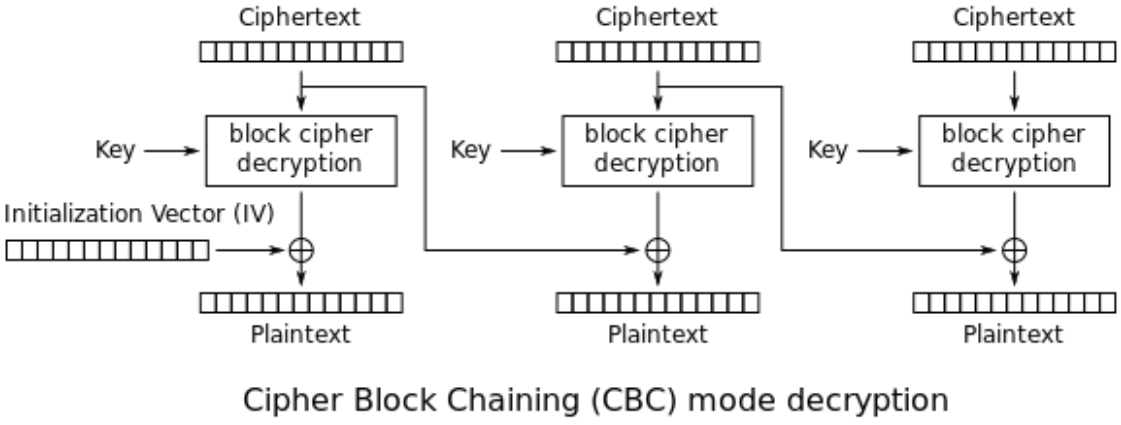

Figure 2.4. CBC denryption

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
import os


#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):
```

```
54
55      c = []
56      for i in range(0, len(plain_text)):
57          c.append(ord(plain_text[i])) #character to integer
58
59      c[0] = c[0] ^ iv # XOR'ing with initialization vector
60
61      for i in range(0, (len(c) - 1)):
62          c[i] = encrypted_elgamal(c[i], key)
63          c[i+1] = c[i+1] ^ c[i]
64
65      c[len(c) - 1] = encrypted_elgamal(c[len(c) - 1], key)
66      return c
67
68  #Decryption function for CBC ciphering mode. It makes the opposite operation of
    CBC_encrpyt function to get plaintext message.
69  def CBC_decrypt(cipher_text, key, iv):
70
71      decrypted_message = []
72      plain_text = []
73
74      for i in range(0, len(cipher_text)):
75          decrypted_message.append(decrypted_elgamal(cipher_text[i], key))
76
77      plain_text.append(chr(iv ^ decrypted_message[0])) # XOR'ing with initialization
    vector
78
79      for i in range(1, len(cipher_text)):
80          plain_text.append(chr(cipher_text[i-1] ^ decrypted_message[i]))
81
82      return plain_text
83
84  #Function for calculating gcd.
85  def gcd(a, b):
86
87      if a < b:
88          return gcd(b, a)
89      elif a % b == 0:
90          return b
91      else:
92          return gcd(b, a % b)
93
94
95  #Function for checking whether number is prime or not.
96  def check_prime(number):
97      control = False
98      for i in range(2,number):
99          if number % i == 0:
100             control = False
101             break
102         else:
103             control = True
104
105     return control
106
107 #Function for creating generator.
108 def generator(field):
109     while(True):
110         results_mod = [] #array of possible generator ^ x (mod field) x={1,2,3...}
111         possible_generator = random.randint(2,field)
```

```
112
113         for i in range(1,field):
114             results_mod.append(power(possible_generator,i,field))
115
116         #checking if there is same numbers inside the array. If it not, then this is
    our generator.
117         if(len(results_mod) == len(set(results_mod))):
118             break
119     return possible_generator
120
121
122 #Function for generating key.
123 def gen_key(p):
124     key = random.randint(pow(10, 2), p)
125
126     #checking gcd
127     while gcd(p, key) != 1:
128         key = random.randint(pow(10, 2), p)
129
130     return key
131
132
133 def main():
134
135     #output .txt file for the testing.
136     output_file = open("output_text.txt",'w')
137
138     #choosing prime number between 10^2-10^5.
139     p = random.randint(pow(10, 2), pow(10, 5))
140     while(True):
141         if check_prime(p) == True:
142             break
143         else:
144             p = random.randint(pow(10, 2), pow(10, 5))
145
146     #choosing generator in the field p.
147     g = generator(p)
148     print("\nIn the field: ", p, "\nwe found a the generator: ", g)
149     output_file.write("In the field: "+ str(p)+ "\nwe found a the generator: "+
    str(g))
150
151     rec_key = gen_key(p) # Public key
152     g_power_a = power(g, rec_key, p)
153
154     print("the public key: ", rec_key)
155     output_file.write("\nthe public key: "+ str(rec_key))
156
157     print("g to the power of a: ", g_power_a)
158     output_file.write("\ng to the power of a: "+ str(g_power_a))
159
160     send_key = gen_key(p)# Private key
161     g_power_b = power(g, send_key, p)
162
163     print("the private key: ", send_key)
164     output_file.write("\nthe private key: "+ str(send_key))
165
166     print("g to the power b: ", g_power_b)
167     output_file.write("\ng to the power b: "+ str(g_power_b))
168
169
```

```
170     g_power_ab = power(g_power_a, send_key, p)
171     print("g to the power of ab:", g_power_ab, "\n")
172     output_file.write("\ng to the power of ab:"+ str(g_power_ab)+ "\n")
173 
174     while(True):
175 
176         print("Please choose input type:\n1 Keyboard Input\n2 File Input
    (input.txt)\n3 Exit program")
177         input_type = input() #FIRST INPUT FOR THE PROGRAM: choosing input type.
178 
179         if input_type == "1":
180 
181             print("\nPlease enter a string to be encrypted: ")
182             message = input() #SECOND INPUT FOR THE PROGRAM: plaintext message.
183             output_file.write("\nPlaintext message is: "+message+"\n")
184 
185         elif input_type == "2":
186 
187             input_file = open("input_test.txt",'r') #open input test file to read.
188             message = input_file.read().splitlines()
189             count = 1
190 
191             #getting plaintext messages line by line in input test file.
192             for msg in message:
193                 print("\nMessage " ,str(count)," in input.txt file: ",msg)
194                 output_file.write("\nPlaintext message "+str(count) +" is: "+msg)
195                 count +=1
196 
197             input_file.close()
198 
199         elif input_type == "3":
200             break
201 
202         else:
203             print("\nPlease enter valid number.")
204             continue
205 
206 
207         print("\nPlease choose a ciphering mode:\n1 ECB\n2 CBC\n3 Exit program")
208         cipher_mode = input() #THIRD INPUT OF THE PROGRAM: choosing the ciphering
    mode.
209 
210         if cipher_mode == "1":
211 
212         # # ECB
213             #encryption of the message.
214             print("\nWe used ECB cipher mode to break up the message, into smaller
    parts, and encrypt every part individually")
215             output_file.write("\n\nWe used ECB cipher mode to break up the message,
    into smaller parts, and encrypt every part individually\n")
216 
217 
218             for msg in message:
219                 #checking if message is string or list. If it is string, then it
    means it is keyboard input, if it is list, then it is .txt file input.
220                 if type(message) == str:
221                     msg = message
222 
223                 cipher_text = ECB_encrypt(msg, g_power_ab)
224 
```

```
225                 print("\nThis is our encrypted message: ", cipher_text)
226                 output_file.write("\nThis is our encrypted message: "+
    str(cipher_text))
227 
228                 #decryption of the message.
229                 plain_text = ECB_decrypt(cipher_text, g_power_ab)
230                 dmsg = ''.join(plain_text)
231 
232                 print("This is our decrypted message: ", dmsg)
233                 output_file.write("\nThis is our decrypted message: "+ dmsg+"\n")
234 
235                 if msg == message:
236                     break
237 
238             break
239 
240         elif cipher_mode == '2':
241 
242         # # CBC
243             #declaring initialization vector as random.
244             iv = random.randint(1, 2000000)
245 
246             #encryption of the message.
247             print("\nWe used CBC cipher mode to break up the message, into smaller
    parts, and encrypt every part individually")
248             output_file.write("\n\nWe used CBC cipher mode to break up the message,
    into smaller parts, and encrypt every part individually\n")
249 
250             #checking if message is string or list. If it is string, then it means it
    is keyboard input, if it is list, then it is .txt file input.
251             for msg in message:
252                 if type(message) == str:
253                     msg= message
254 
255                 cipher_text = CBC_encrypt(msg, g_power_ab, iv)
256 
257                 print("\nThis is our encrypted message: ", cipher_text)
258                 output_file.write("\nThis is our encrypted message: "+
    str(cipher_text))
259 
260                 #decryption of the message.
261                 plain_text = CBC_decrypt(cipher_text, g_power_ab, iv)
262                 dmsg = ''.join(plain_text)
263 
264                 print("This is our decrypted message: ",dmsg)
265                 output_file.write("\nThis is our decrypted message: "+dmsg+"\n")
266 
267                 if msg == message:
268                     break
269             break
270 
271         elif cipher_mode == '3':
272             break
273         else:
274             print("Please enter 1 or 2 to choose a ciphering mode.\n")
275     output_file.close()
276 
277 if __name__ == "__main__":
278     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. To test it properly we implemented two testing methods:

- **Keyboard Input:** this option use keyboard input to test the two algorithms. The program after received the text from the user will encrypt and decrypt the message.
- **File Input (input.txt):** this option use a text file as input so we can test more texts easily without doing the keyboard input every time. If you'll select this method in the menu, the program will go through the file reading lines one by one then it will encrypt and decrypt every line. The program will also print all the terminal output in a txt file, so it'll be easier for the user to check if the program is working properly.

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 [Keyboard input]

```
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 [Keyboard input]

```
In the field:  90977
we found a the generator:  37751
the public key:  18763
g to the power of a:  34268
the private key:  74540
g to the power b:  30627
g to the power of ab: 27378

Please choose input type:
1 Keyboard Input
2 File Input (input.txt)
3 Exit program
2

Message  1  in input.txt file:  Marco and Hasan developed this project hoping to get nice grad

Message  2  in input.txt file:  Italy won the Eurovision last year.

Message  3  in input.txt file:  Turkish basketball team Anadolu Efes won Euroleague last two y

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 in

This is our encrypted message:  [2108106, 2655666, 3121092, 2710422, 3038958, 876096, 2655666,
 2765178, 2956824, 3038958, 3066336, 2765178, 2737800, 876096, 3175848, 2847312, 2874690, 3148
336, 2874690, 3011580, 2819934, 876096, 3175848, 3038958, 876096, 2819934, 2765178, 3175848, 8
This is our decrypted message:  Marco and Hasan developed this project hoping to get nice grad

This is our encrypted message:  [1998594, 3175848, 2655666, 2956824, 3312738, 876096, 3257982,
 2874690, 3148470, 2874690, 3038958, 3011580, 876096, 2956824, 2655666, 3148470, 3175848, 8760
This is our decrypted message:  Italy won the Eurovision last year.

We used ECB cipher mode to break up the message, into smaller parts, and encrypt every part in

This is our encrypted message:  [2108106, 2655666, 3121092, 2710422, 3038958, 876096, 2655666,
 2765178, 2956824, 3038958, 3066336, 2765178, 2737800, 876096, 3175848, 2847312, 2874690, 3148
336, 2874690, 3011580, 2819934, 876096, 3175848, 3038958, 876096, 2819934, 2765178, 3175848, 8
This is our decrypted message:  Marco and Hasan developed this project hoping to get nice grad

This is our encrypted message:  [1998594, 3175848, 2655666, 2956824, 3312738, 876096, 3257982,
 2874690, 3148470, 2874690, 3038958, 3011580, 876096, 2956824, 2655666, 3148470, 3175848, 8760
This is our decrypted message:  Italy won the Eurovision last year.

, 3121092, 3038958, 2956824, 2765178, 2655666, 2819934, 3203226, 2765178, 876096, 2956824, 2655666, 3148470, 3175848, 876096, 3175848, 3257982, 3038958, 876096, 3312738, 2765178, 2655666, 3121092, 314
8470, 1259388]
This is our decrypted message:  Turkish basketball team Anadolu Efes won Euroleague last two years.
```

Figure 3.3. Third test with input text [File input]

3.2.2. CBC

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

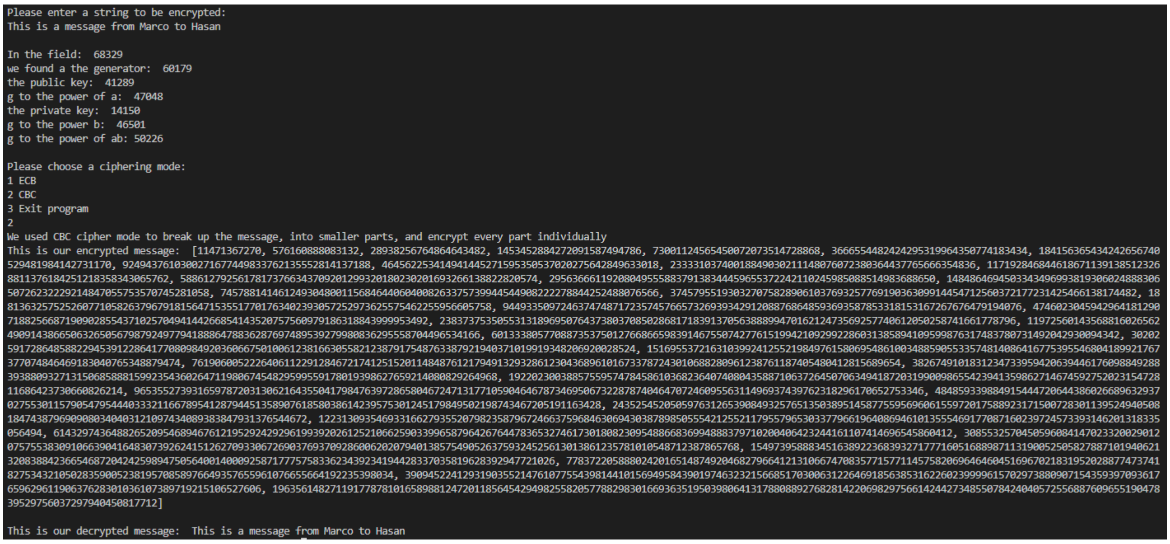

Figure 3.4. First test with a short text [Keyboard input]

Figure 3.5. Second test with a longer text [Keyboard input]

Figure 3.6. Third test with input text [File input]