

University of Southeastern Norway

RFMA310 - Discrete Mathematics

ElGamal encryption system

Candidate Number: 8002

December 7, 2020



Contents

1	Introduction	3
2	ElGamal encryption system	4
2.1	Diffie Hellman key exchange	4
2.2	Key generation	5
2.3	Encryption	6
2.4	Decryption	6
3	The mathematics behind ElGamal	8
3.1	Number theory	8
3.1.1	Prime numbers in cryptography	8
3.1.2	Modular exponentiation	8
3.1.3	Primitive roots	9
3.1.4	Fermat's little theorem	9
3.2	Cyclic group and multiplicative generator	10
3.3	The discrete logarithm problem	11
4	Implementing ElGamal encryption	12
5	Source code	15
6	Conclusion	18
7	References	19

List of Figures

1.1	Taher ElGamal	3
2.1	Illustrating the idea of Diffie Hellman	5
3.1	The different generators	10
4.1	Output(1)	13
4.2	Output(2)	14
5.1	Source code (1)	15
5.2	Source code (2)	16
5.3	Source code (3)	17
5.4	Source code (4)	17

Chapter 1

Introduction

With our increasingly technological world, privacy and safety are more important than ever, thus the role of *cryptography* keeps getting more important with each day that goes by. There are a lot of encryption algorithms designed to help people exchange sensitive information over an insecure channel, and one of these is called ElGamal. The ElGamal encryption system is based on the Diffie-Hellman key exchange and it was described by Taher Elgamal in 1985.



Figure 1.1: Taher ElGamal

In this report I will discuss how the ElGamal encryption system works, the math behind it and finally, my implementation of it.

Chapter 2

ElGamal encryption system

ElGamal encryption system is an asymmetric key encryption algorithm for public-key cryptography. With asymmetric encryption systems such as ElGamal, we use pairs of keys, where the public key can be openly distributed without compromising security, but the private key should be kept confidentially by the user.

The way we use the related pair of keys in ElGamal, in similarity to RSA, (another asymmetric encryption system) is in a way where the public key defines the encryption transformation, while the private key defines the associated decryption transformation. So, if we have a message, the public key will be used to encrypt the message from plaintext to ciphertext, while on the receiving end, the private key will be used to decrypt the message from ciphertext to plaintext. Although this type of asymmetric key system is slower than a symmetric key systems due to the key length, the security is increased with this type of key generating scheme.

2.1 Diffie Hellman key exchange

An important note with the ElGamal encryption system is that it's based on the Diffie-Hellman key exchange. The idea of the Diffie Hellman key exchange is to allow two parties who have not previously met to securely establish a key which they can use to secure their communications. A popular analogy for this is to think of Alice and Bob mixing paint.

Alice and Bob agree publicly on a starting color yellow, then Alice and Bob both randomly select a private color each, red and green that they mix in with their public yellow. Now, Alice keeps her private color, but sends the mixture to Bob, and he does the same. This is where the genius of the trick comes in, when they add their private color to the mixture they got from one another, they obtain the same secret color.

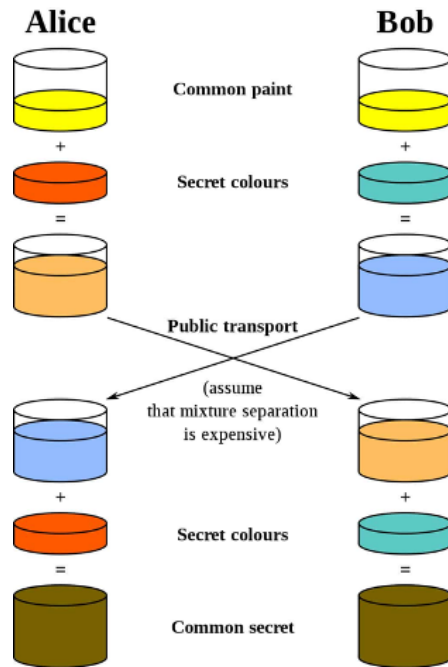


Figure 2.1: Illustrating the idea of Diffie Hellman

In this way, we can secure communication through large distances and as mentioned this scheme is what the ElGamal algorithm is based on, with this idea of a one way function which is hard to reverse. Now, moving on I will explain how the ElGamal algorithm works. We divide it in 3 parts: Key generation, encryption and decryption.

2.2 Key generation

Key generation is the first part of the algorithm, and if we look at how Alice generates a key pair, we have the following steps:

- Select a large prime number p
- Generate an efficient description of a cyclic group G of order q with generator g . Let e represent the unit element of G
- Choose an integer randomly from $\{1, \dots, q-1\}$ which will be the private key.

- Compute the public key, \mathbf{h} , from \mathbf{x} , \mathbf{p} and $\{g, h := g^x \bmod p\}$.
- The public key consists of the values $\{\mathbf{G}, \mathbf{q}, \mathbf{g}, \mathbf{h}\}$. Alice publishes this public key and retains \mathbf{x} as her private key, which must be kept secret.

An integral part of the algorithm is to select a large prime number \mathbf{p} . This is done in order to add complexity and increase security. Another important note here is that the private key, \mathbf{x} must be bigger than 1 and smaller than prime number, $\mathbf{p}-1$. After following these steps, Alice has managed to generate the key pairs for the encryption and decryption.

2.3 Encryption

Alice has generated the key pair and now Bob encrypts a message to Alice. The steps for the encryption are:

- Retrieve Alice's publicly available key $(\mathbf{G}, \mathbf{q}, \mathbf{g}, \mathbf{h})$ and map a message \mathbf{M} to an element \mathbf{m} of \mathbf{G} using a reversible mapping function.
- Choose a random integer from $\{1, \dots, \mathbf{q}-1\}$ in order to compute the shared secret \mathbf{s} .
- Compute $\mathbf{s} := \mathbf{h}^y$. This is called the shared secret, and it is equal to Alice's \mathbf{h} raised to the power of Bob's \mathbf{y} .
- Compute $\mathbf{c}_1 := \mathbf{g}^y$. Bob uses Alice's generator \mathbf{g} and his \mathbf{y} to compute \mathbf{c}_1 .
- Compute $\mathbf{c}_2 := \mathbf{m} * \mathbf{s}$. In order to ensure that Bob's message is only able to be opened by Alice, he encrypts his integer \mathbf{m} by computing \mathbf{c}_2 , which is equal to $\mathbf{m} * \mathbf{s}$.
- Bob sends the ordered pair of ciphertext $(\mathbf{c}_1, \mathbf{c}_2)$ to Alice.

2.4 Decryption

The keys have been generated, and the message has been encrypted. Now Alice must retrieve and decrypt Bob's message. She does this by:

- Retrieving the ordered pair $(\mathbf{c}_1, \mathbf{c}_2)$

- Compute the shared secret $\mathbf{s} := \mathbf{c}^x$, and since $\mathbf{c}_1 = \mathbf{g}^y$, $\mathbf{c}^x = \mathbf{g}^{xy}$, $= \mathbf{h}^y$ and thus it is the same shared secret that was used by Bob during encryption
- In order to decrypt the message, Alice must compute the inverse of \mathbf{s} in group \mathbf{G} .
- Once Alice has the element \mathbf{s} inverse, she decrypts Bob's message by computing $\mathbf{m} := \mathbf{c}_2 * \mathbf{s}^{-1}$, and this produces the original message \mathbf{m} .

After looking at how the algorithm works, it might seem difficult to understand how this is based on the Diffie-Hellman key exchange with all these mathematical equations, but if try to understand it, we see how it is similar to the analogy of Bob and Alice mixing colors.

Chapter 3

The mathematics behind ElGamal

Math is the basis for cryptography because in its essence, all that cryptography is, is mathematical functions. Like another public-key encryption method, RSA, the ElGamal encryption system is based on mathematical ideas. Since encryption and decryption are inverse procedures, there must be a mathematical relationship between the encryption and decryption keys. During this chapter I will introduce some of the mathematical concepts used for ElGamal.

3.1 Number theory

3.1.1 Prime numbers in cryptography

Prime numbers are numbers that have only 2 factors: 1 and themselves, such as 5, 7 and 11. Not only are prime numbers important in ElGamal, but within cryptography in general. This comes from the fact that prime number factorization of especially large numbers can take a long time to compute. Let's say we have a public key used to encrypt a message, that consists of a product of two big prime numbers, and a private key used to decrypt this message that consists of those prime numbers. As with asymmetric encryption, we can publish the public key because only we know the prime factors in order to decrypt the message. A third party can try to factor out the number, but if we chose a large number, computing this would be pretty time consuming.

3.1.2 Modular exponentiation

The modular exponentiation is an important operation for cryptographic transformations in public key cryptosystems like ElGamal. It is important to be able to find $\mathbf{b}^n \bmod \mathbf{m}$ efficiently without using an excessive amount of memory. It is impractical to first compute \mathbf{b}^n and then find its remainder

when divided by \mathbf{m} , because \mathbf{b}^n can be a huge number and we will need a huge amount of computer memory to store such numbers. Instead, we can avoid time and memory problems by using an algorithm that employs the binary expansion of the exponent n . The performance of public key cryptography is primarily determined by the implementation efficiency of the modular multiplication and exponentiation, and that's why this concept is so important.

3.1.3 Primitive roots

During the key generation for ElGamal we want to use something called a primitive root. "The theory of primitive roots state that: if \mathbf{p} is a prime number, then there is some integer \mathbf{a} in $\{1, 2, \dots, \mathbf{p} - 1\}$ such that all of the remainders $\mathbf{a}^t(\bmod \mathbf{p})$, for $\mathbf{t} = \{1, \dots, \mathbf{p} - 1\}$, are distinct. Such an integer \mathbf{a} is called a primitive root modulo \mathbf{p} ". For example, 2 is a primitive root modulo 5, since $2^1 \pmod 5$, $2^2 \pmod 5$, $2^3 \pmod 5$, and $2^4 \pmod 5$ are distinct, but 4 is not a primitive root modulo 5, since $4^2 \equiv 4^4 \equiv 1 \pmod 5$. It isn't required of us to select \mathbf{a} as a primitive root in the key generation phase but it helps increase the complexity of the process.

Furthermore, "from the definition of a primitive root where \mathbf{a} is a primitive root mod \mathbf{p} , we have that $\mathbf{p} - 1$ is the smallest positive exponent \mathbf{n} such that $\mathbf{a}^n \pmod{\mathbf{p}} \equiv 1$ ". This helps us in understanding the Euler-Fermat theorem when applied to primitive roots (this special case is also known as Fermat's little theorem).

3.1.4 Fermat's little theorem

Fermat's little theorem states that if \mathbf{p} is a prime number, then for any integer \mathbf{a} , the number $\mathbf{a}^p - \mathbf{a}$ is an integer multiple of \mathbf{p} . With $\mathbf{a}^{p-1} \equiv 1 \pmod{\mathbf{p}}$ we see that the nature of primitive roots that the following can only be true for the primitive root \mathbf{a} , the exponents \mathbf{x} and \mathbf{y} and the prime \mathbf{p} when $\mathbf{x} \equiv \mathbf{y} \pmod{\mathbf{p} - 1}$: $\mathbf{a}^x \equiv \mathbf{a}^y \pmod{\mathbf{p}}$.

In general, this theorem has proven to be quite useful. For example, we can know if a number is a prime or not, and whether it's composite without having to go through all of its factors. This theorem is indeed used in the logic of ElGamal, but also throughout other encryption systems.

3.2 Cyclic group and multiplicative generator

We saw in the key generation of a public key that Alice generates a multiplicative cyclic group \mathbf{G} , of order \mathbf{q} , with generator \mathbf{g} . Some groups have an interesting property: all the elements in the group can be obtained by repeatedly applying the group operation to a particular group element. If a group has such a property, it is called a cyclic group and the particular group element is called a generator.

$g \backslash a$	0	1	2	3	4	5	6	7	8	9	10
0	1	0	0	0	0	0	0	0	0	0	0
1	1	1	1	1	1	1	1	1	1	1	1
2	1	2	4	8	5	10	9	7	3	6	1
3	1	3	9	5	4	1	3	9	5	4	1
4	1	4	5	9	3	1	4	5	9	3	1
5	1	5	3	4	9	1	5	3	4	9	1
6	1	6	3	7	9	10	5	8	4	2	1
7	1	7	5	2	3	10	4	6	9	8	1
8	1	8	9	6	4	10	3	2	5	7	1
9	1	9	4	3	5	1	9	4	3	5	1
10	1	10	1	10	1	10	1	10	1	10	1

Figure 3.1: The different generators

If we consider an example with **mod p** where $p=11$. Next we have to find a generator \mathbf{g} . When starting, any number $\{0, \dots, n1\}$ (or \mathbf{Z}_p for short) is a candidate. The chart below is showing each g value as a row, each a value as a column, and the expression $\mathbf{g}^a \bmod(11)$ evaluated for each \mathbf{g} and \mathbf{a} .

For example, from this chart we see that generators 4,5,9 generate the same group as 3. In other words, there are a lot of properties in this chart but the relevant one is to consider the order (number of elements) in each possible group. This concept is important in ElGamal, because if we have both p and the order of the subgroup \mathbf{q} as large primes, we increase the security of our system.

3.3 The discrete logarithm problem

Finally, we come to one of the most important properties of ElGamal. “The discrete logarithm problem is the problem of determining to which power some value was raised in order to attain the result when working in a modular ring (usually, modulo some prime)”. As with the Diffie-Hellman key exchange, the ElGamal system is based on the discrete logarithm problem, meaning that the security of the ElGamal algorithm is based on the difficulty of computing discrete logarithms. Discrete logarithms are quickly computable in a few special cases. However, no efficient method is known for computing them in general, and that’s why they a lot of public-key cryptography base their security on this assumption.

Specifically, we define the discrete logarithm problem as: given a group \mathbf{G} , a generator \mathbf{g} of the group and an element \mathbf{h} of \mathbf{G} , to find the discrete logarithm to the base \mathbf{g} of \mathbf{h} in the group \mathbf{G} . A popular choice of groups for discrete logarithm based cryptosystems as we’ve seen is \mathbf{Z}_p^* , where \mathbf{p} is a prime number. An example where $2x \equiv 9 \pmod{13}$. Since $28 = 256$ which is $9 \pmod{13}$, $x = 8$ is a valid solution to the above problem.

As mentioned previously, for small values of \mathbf{p} , this problem can be solved through computing since there will only be $\mathbf{p}-1$ possible exponents as per the Euler-Fermat theorem. From Fermat’s little theorem we know that if \mathbf{p} is a prime number and \mathbf{a} is not a multiple of \mathbf{p} , then $\mathbf{a}^{p-1} \equiv 1 \pmod{\mathbf{p}}$, and the Fermat–Euler theorem says that if \mathbf{a} is relatively prime to \mathbf{m} , then $\mathbf{a}^{\phi(\mathbf{m})} \equiv 1 \pmod{\mathbf{m}}$, where $\phi(\mathbf{m})$ is called a totient function by Euler. We can use this to count the number of positive integers relatively prime and less than \mathbf{m} .

Now, on the other side, for large value of the prime number \mathbf{p} (usually at least 1024-bit), the discrete logarithm problem currently has no known solution because often the number of iterations required is actually exponential to the size of our input of the problem. It’s what makes the discrete logarithm problem hard to solve, hence why the ElGamal encryption system among others, use it as their foundation to base the security upon.

Chapter 4

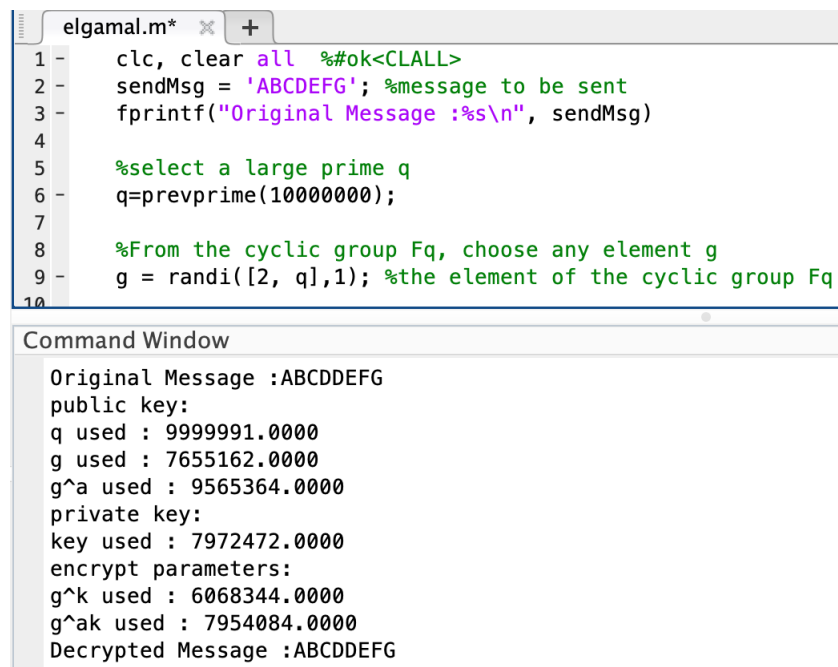
Implementing ElGamal encryption

I decided to implement ElGamal in MATLAB. It was between MATLAB and python, but I thought it would be nice to refresh my knowledge within MATLAB. The program runs as intended and it successfully encrypts and decrypts the message of choice.

In the program I choose any plaintext of my choice. Furthermore, I must choose the prime number q as whatever I want, and I must also choose the elements in the cyclic group.

Starting off with the implementation was a bit tricky because I didn't know where to start, but after focusing on understanding the math used in the algorithm things became much clearer. Another challenge was to make sure that the implementation was realistic in the sense that we could use a large prime number q , so I also had to make sure of that.

Below are the sampled outputs. I printed the public and private keys used during the encryption and the different parameters. For example, g is an element in the group, q is the prime number, g^a is the shared secret and so on. Firstly, the plaintext I chose was "ABCDEFGH", the prime number is "10000000" and the elements in the cyclic group are "2,1". In the 2nd sampled output, I changed the parameters, the plaintext became "VAMOS", the prime number is "11111111" and the elements in the group are "5,1". As we can see from these figures, the encryption and decryption was done successfully.



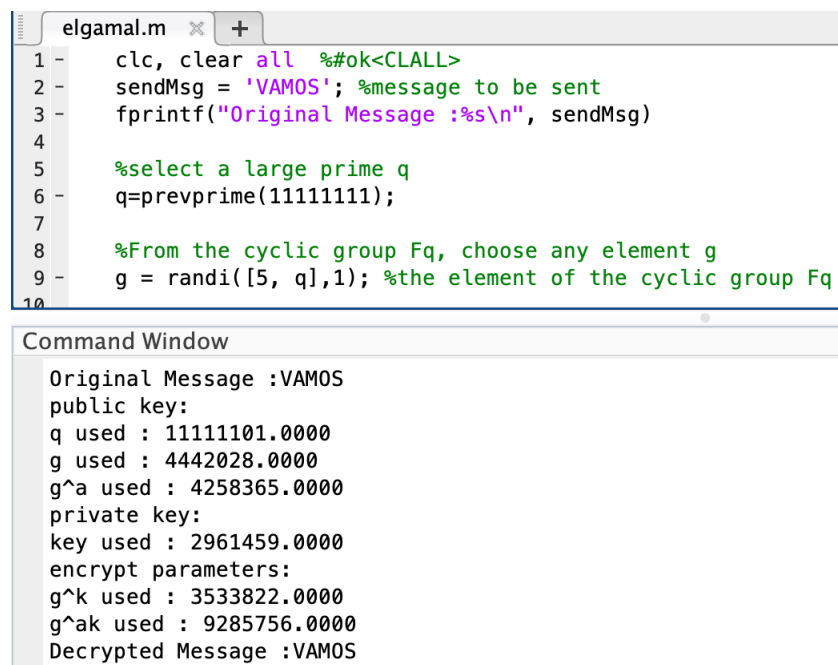
The image shows a MATLAB script editor window titled 'elgamal.m*' and a Command Window below it. The script implements an ElGamal encryption and decryption process. The Command Window displays the output of the script, including the original message, public and private keys, encryption parameters, and the decrypted message.

```
1 - clc, clear all %#ok<CLALL>
2 - sendMsg = 'ABCDEFGF'; %message to be sent
3 - fprintf("Original Message :%s\n", sendMsg)
4
5 - %select a large prime q
6 - q=prevprime(10000000);
7
8 - %From the cyclic group Fq, choose any element g
9 - g = randi([2, q],1); %the element of the cyclic group Fq
10
```

Command Window

```
Original Message :ABCDDEFG
public key:
q used : 9999991.0000
g used : 7655162.0000
g^a used : 9565364.0000
private key:
key used : 7972472.0000
encrypt parameters:
g^k used : 6068344.0000
g^ak used : 7954084.0000
Decrypted Message :ABCDDEFG
```

Figure 4.1: Output(1)



The image shows a MATLAB script editor window titled 'elgamal.m' with a line number margin on the left. The script contains 10 lines of code for ElGamal encryption. Below the editor is a 'Command Window' showing the output of the script. The output includes the original message 'VAMOS', the public key components (q, g, g^a), the private key (key), the encryption parameters (g^k, g^ak), and the decrypted message 'VAMOS'.

```
1 -   clc, clear all %#ok<CLALL>
2 -   sendMsg = 'VAMOS'; %message to be sent
3 -   fprintf("Original Message :%s\n", sendMsg)
4
5 -   %select a large prime q
6 -   q=prevprime(11111111);
7
8 -   %From the cyclic group Fq, choose any element g
9 -   g = randi([5, q],1); %the element of the cyclic group Fq
10
```

Command Window

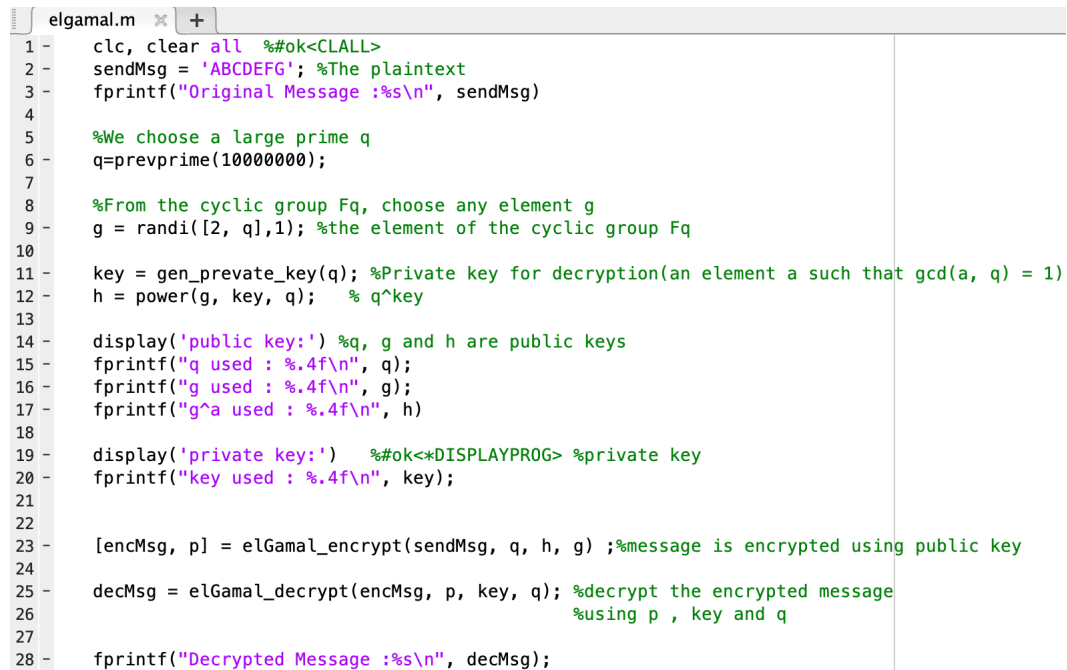
```
Original Message :VAMOS
public key:
q used : 11111101.0000
g used : 4442028.0000
g^a used : 4258365.0000
private key:
key used : 2961459.0000
encrypt parameters:
g^k used : 3533822.0000
g^ak used : 9285756.0000
Decrypted Message :VAMOS
```

Figure 4.2: Output(2)

Chapter 5

Source code

Below is the source code for the project with comments that explain the functionality:



```
1 - clc, clear all %#ok<CLALL>
2 - sendMsg = 'ABCDEFGH'; %The plaintext
3 - fprintf("Original Message :%s\n", sendMsg)
4 -
5 - %We choose a large prime q
6 - q=prevprime(10000000);
7 -
8 - %From the cyclic group Fq, choose any element g
9 - g = randi([2, q],1); %the element of the cyclic group Fq
10 -
11 - key = gen_prvate_key(q); %Private key for decryption(an element a such that gcd(a, q) = 1)
12 - h = power(g, key, q); % q^key
13 -
14 - display('public key:') %q, g and h are public keys
15 - fprintf("q used : %.4f\n", q);
16 - fprintf("g used : %.4f\n", g);
17 - fprintf("g^a used : %.4f\n", h)
18 -
19 - display('private key:') %#ok<*DISPLAYPROG> %private key
20 - fprintf("key used : %.4f\n", key);
21 -
22 -
23 - [encMsg, p] = elGamal_encrypt(sendMsg, q, h, g) ;%message is encrypted using public key
24 -
25 - decMsg = elGamal_decrypt(encMsg, p, key, q); %decrypt the encrypted message
26 - %using p , key and q
27 -
28 - fprintf("Decrypted Message :%s\n", decMsg);
```

Figure 5.1: Source code (1)


```

30 % ElGamal encryption
31 function dr_msg = elGamal_decrypt(encMsg, p, key, q)
32     %p: it is recieved from encryption
33     %key: decrypting the private key
34     %q: public key
35     dr_msg = '';
36     s = power(p, key, q) ;%decryption calculates secret key
37     for ii=1 : length(encMsg)
38         dr_msg =strcat(dr_msg , (char(fix(encMsg(ii)/s)))) ;%decrypt the message by
39                                     %computing M = encMsg x s^-1
40     end
41 end
42
43 % ElGamal encryption
44 function [en_msg2, p] = elGamal_encrypt(msg, q, h, g)
45     % a message is encrypted using public key q, h and g
46     en_msg = msg;
47
48     k = gen_prevate_key(q); % Private key for the sender
49
50     s = power(h, k, q); %the shared secret
51     p = power(g, k, q); %the sender calculates g^k
52
53     display('encrypt parameters:')
54     fprintf("g^k used : %.4f\n", p);
55     fprintf("g^ak used : %.4f\n", s);
56     for ii =1:length(en_msg)
57         en_msg2(ii) = s * double(en_msg(ii)); %#ok<AGROW> %encrypt m by computing X=msg x s
58     end
59 end

```

Figure 5.2: Source code (2)

```

61 % Generating private key
62 function key= gen_prevate_key(q)
63     %private key is generated such that gcd(key, q) = 1
64     key = randi([100000, q],1);
65
66     while (gcd(q, key) ~= 1)
67         key = randi([100000, q],1) ;
68     end
69 end
70
71 % power calculation
72 function rr = power(a, b, c)
73     %a: element of the cyclic group Fq , or the shared secret
74     %b: exponent
75     %c: prime number
76     x = 1;
77     y = a;
78
79     while b > 0
80         if mod(b, 2) == 0
81             x = mod((x * y) , c);
82         end
83         y = mod((y * y) , c );
84         b = fix(b / 2) ;
85     end
86
87     rr= mod(x, c);
88 end

```

Figure 5.3: Source code (3)

```

90 function rr= gcd(x, y)
91     %the condition function used when private key is generated
92     if x < y
93         rr= gcd(y,x);
94     elseif mod(x , y) == 0
95         rr= y ;
96     else
97         rr= gcd(y, mod(x , y));
98     end
99 end

```

Figure 5.4: Source code (4)

Chapter 6

Conclusion

The ElGamal encryption system is a fascinating cryptographic scheme. It is a secure system which is why it's still being used today.

What I've found the most exciting about this system is the "magic" of cryptography and math. Firstly, the Diffie Hellman key exchange, if we go back to the analogy of mixing colors and the fact that Alice and Bob, without knowing each other's "secret" color can still get the same color, it is sort of magical. This in turn, means that we can secure communication on a line between Alice and Bob who live on the opposite side of the globe.

Secondly, the discrete logarithm problem that this algorithm is built upon. With time, maybe quantum computers will become powerful enough to solve properties of this problem, and it will be exciting to see this development, but for now it still remains a great foundation to build a cryptographic system on.

With that being said, i have learned a lot about ElGamal (which i had never heard of before this assignment), the math behind this system and indirectly the importance of math within cryptography with the use of prime numbers, cyclic groups, the different theorems and so on.

Chapter 7

References

Link to code: <https://drive.google.com/file/d/1ze0AtmUM6XEqp705PY91IQTh2uIHIDP1/view?usp=sharing>

My references:

[1] Rosen, Kenneth (2011). Discrete mathematics and its applications, McGraw hill.

[2] “Asymmetric cryptography”. (March 2020). <https://searchsecurity.techtarget.com/definition/asymmetric-cryptography>

[3] “Public-key cryptograph”. (July 2018). https://en.wikipedia.org/wiki/Public-key_cryptography

[4] “ElGamal encryption”. (February 2017). https://en.wikipedia.org/wiki/ElGamal_encryption

[5] “Discrete Logarithms, The ElGamal Cryptosystem and Diffie-Hellman Key Exchange”. (July 2020). <https://www.commonlounge.com/discussion/2be4d294aa9e44d4b67f6644cd9b5ced/main>

[6] “Discrete logarithm”. (October 2017). https://en.wikipedia.org/wiki/Discrete_logarithm

[7] “ElGamal”. (2017). <https://math.asu.edu/sites/default/files/elgamal.pdf>

[8] “Prime numbers in cryptography”. (January 2012). <https://stackoverflow.com/questions/439870/why-are-primes-important-in-cryptography>

[9] “Discrete logarithms”. (September 2016). <https://www.doc.ic.ac.uk/~mrh/330tutor/ch06.html>

- [10] “Multiplicative cyclic group and key generation”. (2015). <https://tinyurl.com/y6h5rw9o>
- [11] “ElGamal”. (March 2011). <https://orion.math.iastate.edu/cbergman/crypto/psfiles/4up/elgamal.pdf>
- [12] “ElGamal public-key encryption ”. (2014). http://ipco-co.com/PET_Journal/Papers%20CEIT'14/025.pdf
- [13] “Euler’s theorem”. (October 2017). https://en.wikipedia.org/wiki/Euler%27s_theorem