ElGamal Encryption Assignment 1

Christian Vilen

2023-09-26

1 Introduction

This report will highlight and discuss the three different tasks of using ElGamal's encryption algorithm method and the technical implementations to utilise it in practice.

1.1 Code implementation

See the attached User.java file for the utilised methods and the AssignmentOneMain.java file for the tests.

2 Task 1

In task one, we need to send "2000 kr" to Bob from Alice using ElGamal's encryption algorithm given the following public material:

- The public shared base g = 666
- The public shared prime p = 6661
- Bob's public key $PK = g^x \mod p = 2227$

The public material has been used to create instances of the User.java class to represent Alice and Bob with the same shared base and prime (The public key has been randomised for all users). The User class has several fields to represent a real-world example of a person who can have multiple messages in their inbox and a generation of public and private keys. See image 1.

```java
public class User {
    private String name;
    private BigInteger g, p, privateKey, publicKey, friendsPublicKey;
    private BigInteger[] inbox;
    private int messageCount;

    public User(String name, BigInteger g, BigInteger p) {
        this.name = name;
        this.g = g;
        this.p = p;
        this.privateKey = new BigInteger(p.bitLength(), new Random());
        this.publicKey = g.modPow(privateKey, p);
        this.inbox = new BigInteger[10];
        this.messageCount = 0;
        this.friendsPublicKey = BigInteger.ZERO;
    }
```

Figure 1: User class fields and constructor

Therefore, we can create a new instance of the User.java class to represent Alice and Bob and share their public keys with each other to start the first step of ElGamal's algorithm. See image 2.

```java
BigInteger sharedG = BigInteger.valueOf(val:666);
BigInteger sharedP = BigInteger.valueOf(val:6661);
// ------------------------------------------------------------------------------------------
// Task 1:
System.out.println(x:"\n-------------------------\nTask 1:\n-------------------------\n");
// Alice and Bob computes their public keys
User Alice = new User(name:"Alice", sharedG, sharedP);
User Bob = new User(name:"Bob", sharedG, sharedP);

// Alice sends her public key to Bob
Bob.sendFriendpk(Alice.getPk());

// Bob sends his public key to Alice
Alice.sendFriendpk(Bob.getPk());
```

Figure 2: Initialise Alice and Bob

Once the public keys have been shared between Alice and Bob, Alice can create a message and send it to Bob using ElGamal's Encryption method. This is done in the User class by using the methods addMessage(), encryptMessage(), getMessage and decryptMessage() to create the message, encrypt it, and transfer it to Bob which then can decrypt it to be read. See image 3.

```java
public void addMessage(String message, boolean encrypt) {
    if (messageCount < 10) {
        BigInteger ciphertext2;
        // Encrypting the message if encrypt is true
        if (encrypt) {
            BigInteger[] ciphertext = encryptMessage(message);
            ciphertext2 = ciphertext[1];
            inbox[messageCount] = publicKey;
        } else {
            ciphertext2 = new BigInteger(message);
            inbox[messageCount] = friendsPublicKey;
        }
        // Adding the message to the inbox
        inbox[messageCount + 1] = ciphertext2;
        messageCount += 2;
    } else {
        System.out.println(x:"Inbox is full");
    }
}

    private BigInteger[] encryptMessage(String message){
    BigInteger messageInt = new BigInteger(message);
    BigInteger ciphertext2 = messageInt.multiply(friendsPublicKey.modPow(privateKey, p)).mod(p);
    return new BigInteger[]{publicKey, ciphertext2};
}

public String getMessage(boolean decrypt) {
    if (messageCount < 2) {
        return "No messages in inbox";
    } else {
        String decryptedString;
        if (decrypt) {
            decryptedString = decryptMessage(inbox);
        } else {
            decryptedString = inbox[1].toString();
        }

        for (int i = 0; i < messageCount - 2; i += 2) {
            inbox[i] = inbox[i + 2];
            inbox[i + 1] = inbox[i + 3];
        }
        messageCount -= 2;

        return decryptedString;
    }
}

private String decryptMessage(BigInteger[] ciphertext){
    BigInteger ciphertext1 = ciphertext[0];
    BigInteger ciphertext2 = ciphertext[1];
    BigInteger s = ciphertext1.modPow(privateKey, p);
    BigInteger decryptedMessage = ciphertext2.multiply(s.modInverse(p)).mod(p);
    return decryptedMessage.toString();
}
```

Figure 3: addMessage(), encryptMessage(), getMessage and decryptMessage()

The addMessage(), encryptMessage(), getMessage and decryptMessage() methods have then been used in the main in image 4 to produce the output as seen in image 5.

```
// Alice creates a message and stores it in her inbox[]
Alice.addMessage(message:"2000", encrypt:true);

// Alice sends the message to Bob
Bob.addMessage(Alice.getMessage(decrypt:false), encrypt:false);
Alice.pprint();
Bob.pprint();
System.out.println(x:"\n");

// Bob decrypts the message and can read it
System.out.println("The message to bob is: " + Bob.getMessage(decrypt:true) + ", and should be 2000");
```

Figure 4: Send message from Alice to Bob

Image 3 shows the pretty print from both Alice and Bob plus the correct and expected output.

```
--------------------------
Task 1:
--------------------------

------------------------
Pretty print of: Alice
------------------------
g:                 666
p:                 6661
privateKey:        1848
publicKey:         547
friendsPublicKey: 6097
------------------------
------------------------
Pretty print of: Bob
------------------------
g:                 666
p:                 6661
privateKey:        3320
publicKey:         6097
friendsPublicKey: 547
inbox[0]:          547
inbox[1]:          3588
------------------------


The message to bob is: 2000, and should be 2000
```

Figure 5: Output from task 1

3 Task 2

Task 2 requires that a new individual, Eve, should try to brute-force her way into acquiring Bob's private key to decrypt the message from Alice. This

can be done while accessing g, p and Bob's public key. Accordingly, a new method interceptEncryptedInbox() in the User.java class has been added to simulate the interception of the message. See image 6.

```java
public void interceptEncryptedInbox(User user) {
    this.messageCount = user.messageCount;
    for (int i = 0; i < user.messageCount; i++) {
        this.inbox[i] = user.inbox[i];
    }
}
```

Figure 6: interceptEncryptedInbox() method

The interceptEncryptedInbox() method has been used in the following main method in a combination to brute-force Eve's way into acquiring Bob's public key. This resulted in Eve being able to decrypt the message successfully. See image 7.

```java
// ------------------------------------------------------------------------------------------
// Task 2:
System.out.println(x:"\n-------------------------\nTask 2:\n-------------------------\n");

// Eve computes her public key
User Eve = new User(name:"Eve", sharedG, sharedP);

// Eve performs a brute-force attact to find bobs private key with his public key
BigInteger privateKeyGuess = BigInteger.ONE;
BigInteger result = sharedG.modPow(privateKeyGuess, sharedP);
while (!result.equals(Bob.getPublicKey())) {
    privateKeyGuess = privateKeyGuess.add(BigInteger.ONE);
    result = sharedG.modPow(privateKeyGuess, sharedP);
}
System.out.println("Eve has found Bob's private key: " + privateKeyGuess+"\n");
Eve.setPrivateKey(privateKeyGuess);

// Eve intercepts the message from Alice to Bob
Alice.addMessage(message:"2000", encrypt:true);
Bob.addMessage(Alice.getMessage(decrypt:false), encrypt:false);
Eve.interceptEncryptedInbox(Bob);
System.out.println(x:"Eve has intercepted the message from Alice to Bob");

Eve.pprint();
Bob.pprint();
System.out.println("Eve intercepted bob's message which is: " + Eve.getMessage(decrypt:true) + ", and should be 2000");
System.out.println("The message to bob is: " + Bob.getMessage(decrypt:true) + ", and should be 2000");
```

Figure 7: Main method of task 2

Image 8 shows the pretty print from both Bob and Eve plus the correct and expected output.

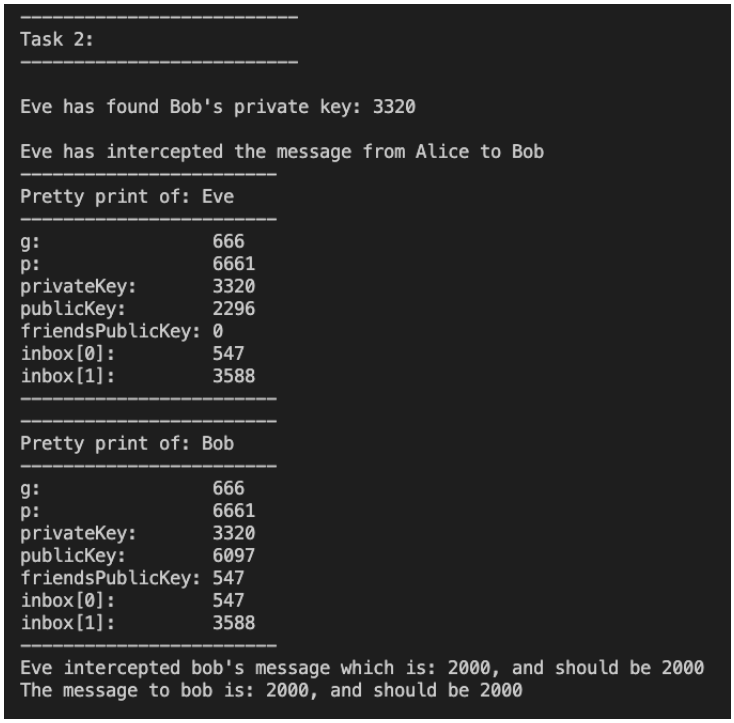

Figure 8: Task 2 output

4 Task 3

Task 3 requires a new individual, Weave, who should be able to change Bob's message from Alice. However, Weave does not have the capability to brute-force her way into Bob's private key or access to g, p or Bob's public key. Accordingly, Weave can only intercept the message from Alice and change it without encrypting and decrypting the original message. Therefore, the interceptEncryptedInbox() method has been used by Weave to intercept the message from Alice whereafter the changeMessage() method has been utilised to interfere the original message before it was sent to Bob. See changeMessage() method in image 9.

```
public void changeMessage(BigInteger multiplier) {
    for (int i = 1; i < messageCount; i += 2) {
        inbox[i] = inbox[i].multiply(multiplier);
    }
}
```

Figure 9: changeMessage() method

Weave can then change the original message by interfering with a multiplying method on it. This can be seen in the following main method in image 10.

```java
// ------------------------------------------------------------------------------------------
// Task 3:
System.out.println(x:"\n-------------------------\nTask 3:\n-------------------------\n");
// Weave computes her public key
User Weave = new User(name:"Weave", sharedG, sharedP);

// Weave intercepts the message from Alice to Bob and multiply the amount
BigInteger multiplyAmount = BigInteger.valueOf(val:2);

Alice.addMessage(message:"2000", encrypt:true);
Weave.interceptEncryptedInbox(Alice);
Weave.pprint();
Weave.changeMessage(multiplyAmount);
Weave.pprint();
String str = Weave.getMessage(decrypt:false);
System.out.println("weave message: " + str);
Bob.addMessage(str, encrypt:false);
Bob.pprint();
// Weave intercepts the message from Alice to Bob and multiplies the amount
System.out.println("Weave has intercepted the message from Alice to Bob and multiplied the amount by " + multiplyAmount);
System.out.println("The message to bob is: " + Bob.getMessage(decrypt:true) + ", and should be 2000 * " + multiplyAmount);
```

Figure 10: Main method for task 3

Image 11 shows the pretty print from both Bob and Weave plus the correct and expected output.

```
-------------------------
Task 3:
-------------------------


-------------------------
Pretty print of: Weave
-------------------------
g:                 666
p:                 6661
privateKey:        5428
publicKey:         6431
friendsPublicKey: 0
inbox[0]:          3124
inbox[1]:          3165
-------------------------
-------------------------
Pretty print of: Weave
-------------------------
g:                 666
p:                 6661
privateKey:        5428
publicKey:         6431
friendsPublicKey: 0
inbox[0]:          3124
inbox[1]:          6330
-------------------------
weave message: 6330
-------------------------
Pretty print of: Bob
-------------------------
g:                 666
p:                 6661
privateKey:        4572
publicKey:         3694
friendsPublicKey: 3124
inbox[0]:          3124
inbox[1]:          6330
-------------------------
Weave has intercepted the message from Alice to Bob and multiplied the amount by 2
The message to bob is: 4000, and should be 2000 * 2
```

Figure 11: Task 3 output