# Coursework for CSC3621 Cryptography

## Part 2

### Exercise 1

Question 1:

Code implementation of the Extended Euclidian Algorithm

The file Results/exercise1\_question1

Question 2:

The bases of Euclid’s algorithm is to fine the greatest common denominator of two positive non-zero integers. The basic algorithm uses the core understanding that you can reduce the problem of finding the gcd of two given numbers by taking the smallest number away from the largest number and finding the gcd of those two smaller numbers. This would be done recursively until the numbers are small enough to simply solve. By using modulus we can deduce that the gcd(x1, x2) = gcd(x2, x1 mod x2) where x1 is smaller than x2.

e.g. gcd(27, 21) = gcd(21, 21 mod 27) = gcd(21, 6)

gcd(21, 6) = gcd(6, 6 mod 21) = gcd(6, 3)

gcd(6, 3) = gcd(3, 3 mod 6) = gcd(3, 0) = 0

The Extended Euclidian Algorithm builds upon the basic algorithm allowing the calculation of not only the greatest common denominator but also that of Bezout's constants with little extra computation. These constants come from the equation gcd(a, b) = sa + tb where a and b are the two inputs. The resultant t found in this equation is also known as the modulo inverse of ‘a’, if ‘a’ is a given co-prime integer to ‘b’ is the modulus of the group.

Question 3:

3 Uses of Extended Euclidian Algorithm

1. Finding the greatest common denominator of two given numbers.
2. Finding the modular inverse of input a where input b is the modulus of the group. This is useful for key generation in the RSA Cryptosystem.
3. Testing if two numbers are co-prime to one another and therefore the resulting GCD of those two numbers would be 1.

### Exercise 2

Question 1:

Code implementation of linear equations

Question 2a:

Modular Inverse ( a-1 ) =

219802339092955255913243081374736248146637572723771805527242533284590535880989095947896263649292081572578174703531012466678224676938135185305924162097847568089867614885100976845782551435532564219936898309841920

Result (x = -b \* a-1) =

421183184045396008847949672535189703891596775623386918532288953516218079938085028752886155666620391419813631317167316631819337277694338966865265179264143550762845428781671313603290317056892475467340480506865970

Therefore a \* x + b = 0

Question 2b:

Returned NULL as it is unsolvable.

Question 3:

Special property of N is that it should be a prime number. Being a prime number means that all elements within its set are invertible (except for 0) and therefore we can calculate the inverse of any given element of that set allowing us to solve the linear equation using x = -b \* a-1.

The set of results cannot be solved for question 2b as ‘a’ is not co-prime to ‘N’. In other words the GCD of ‘a’ and ‘N’ is not equal to 1 (its 2) and therefore you cannot find the inverse of ‘a’. The linear equation is unsolvable.

For calculating the inverse of ‘a’ using Fermat’s theorem we would use the equation x-1 = xN-2 (mod N).

Comparing the time taken to compute Fermat’s theorem against the time taken to compute the inbuilt BigInteger modular inverse function. We can see that on average Fermat’s theorem takes around 10 times longer than that of modular inverse. This is likely due to raising ‘a’ to the power of the exponent N-2. This is a very expensive operation.

### Exercise 3

Question 1:

Code implementation of Diffie-Hellman

Question 2:

The file Results/exercise3\_question2.txt contains the transcript.

Question 3:

What we want from a secure channel is the secrecy and integrity of a message to hold from A to B (and B to A). Encrypting a message using the now known session key means that no one listening to the channel can read the messages being sent. To make sure that the message is integral we need to add an authentication key.

Encrypting all messages with the session key and using the session key as an authentication code is not very secure. Instead we want to produce multiple keys from the session key. One for encryption from A to B, one for encryption from B to A, one for authentication from A to B and one for authentication from B to A. These 4 keys can be calculated from the original session key. Using HMAC (Hash based message authentication code) with a random salt we can calculate a key extension and then using this key extension we can calculate the 4 keys needed for message transfer.

Question 4a:

Breaking DH

One way that the Attacker Eve could break the Diffie-Hellman key exchange is by the Man in the Middle attack. The bases of this is Eve intercepting all key setup messages sent from A to B and B to A. She then edits them so A believes it has set up communication with B and vice versa.

Say Alice wants to set up communication with Bob. Alice will send the first compute message with a publically known modulus and base as well as valueA which is created using her private key. Eve the attacker will intercept this message, change the value of valueA to use her public key and forward this on to Bob. When Bob replies with its valueB, Eve will calculate the secret key for communication between herself and Bob. She will also send a fake reply to Alice where valueB has been edited to use Eve’s private key allowing her to calculate the session key between herself and Alice.   
Eve will now have a communication channel set up between herself and Alice and herself and Bob. As long as Eve keeps listening and forwards on every message she will be able to read the full conversation. Eve would also be able to change any message that was sent to her and forge an appropriate reply without either of the two parties knowing.

Diffie-Hellman is vulnerable to this kind of attack as each party doesn’t authenticate who they are setting up a key exchange with. An example solution to this problem is using digital signatures to authenticate who each party is talking to.

Question 4b:

Code implementation of Man in the Middle Attack

The file Results/exercise3\_question4 includes the transcript for the man in the middle attack