IS53012B/S Computer Security

Mohammed Tahmid (33595286)
mtahm001@gold.ac.uk

Dardan Quqalla (33498388)
dquqa001@gold.ac.uk

Butrint Termkolli (33551538)
bterm001@gold.ac.uk

March 8, 2020

Abstract

This document outlines the RSA algorithm and the Needham-Schroeder Protocol as part of the Goldsmiths, UoL BSc Computer Science module **IS53012B/S: Computer Security** taught by **Dr Ida Pu**. In the first part we go through the RSA algorithm and how it works and then provide an example using Alice and Bob notation. We then discuss how Charlie (the hacker) can brute force RSA if they keys aren't large enough. We then go onto explain our own basic Java implementation of the RSA protocol and finally we discuss trusted servers and our implementation of the NSPK protocol.

Total Number of Hours Spent	92 hours
Hours Spent for Algorithm Design	16 hours
Hours Spent for Programming	28 hours
Hours Spent for Writing Report	30 hours
Hours Spent for Testing	18 hours
Note for the examiner (if any)	Please see run instructions below.
To run RSA Algorithm demo from the /code folder:	**java RSAAlgorithm**
To run NSPK Protocol demo from the /code folder:	**java NSPK**

Files included in */code* folder:

RSAAlgorithm (.java, .class) *- Main RSA Algorithm demonstration program!*

NSPK (.java, .class) *- Main Needham-Schroeder Protocol demonstration program!*

Person (.java, .class) *- Helper class for NSPK*

Server (.java, .class) *- Helper class for NSPK*

RSA_Helper (.java, .class) *- Stripped down RSA program for use with NSPK.java (Helper class for NSPK)*

1 The RSA Algorithm - An Introduction

The RSA algorithm is named after its inventors Rivest, Shamir and Adleman. The algorithm is a block cipher where the plain text and cipher text are roughly of equal length and between 0 and $n-1$ for some n. [1] The algorithm is asymmetric meaning the keys for encryption and decryption are different from each other. We call this public-key cryptography.

The security of RSA is based on the difficulty of prime factorisation. With large enough numbers, this essentially works as a one-way function as it is easy to multiply 2 prime numbers, p and q and get the result n, but it is difficult to take the result n and calculate the inverse to figure out the two prime factors (p and q).

With large enough prime numbers, RSA is computationally secure as it would take an infeasible amount of time to break RSA. The recommended length of an RSA key nowadays is 2048 bits and above. [2]

2 How RSA Works

RSA is made up of a pair of public and private keys. Here are the main steps of the algorithm:

$$p,\ q,\ n,\ \phi(n),\ e\ and\ d$$

STEP 1. Generate 2 large prime numbers, p and q and multiply them to get n

$$n = p * q$$

STEP 2. Calculate $\phi(n)$

$$\phi(n) = (p-1) * (q-1)$$

STEP 3: Select an e such that e is greater than 1 but less than $\phi(n)$ and is also relatively prime to $\phi(n)$

$$e = 1 < e < \phi(n), where\ e\ is\ relatively\ prime\ to\ \phi(n)$$

STEP 4: Compute d such that d is modular inverse of $(e, \phi(n))$.
In other words, $e * d\ mod\ \phi(n) = 1$

$$d = e^{-1}\ mod\ \phi(n)$$

STEP 5: We discard p and q. We now have the public and private key. We publish the public key and we keep our private key!

$$public\ key = (e, n)$$
$$private\ key = (d, n)$$

STEP 6: To encrypt message m, we break the message into blocks, then

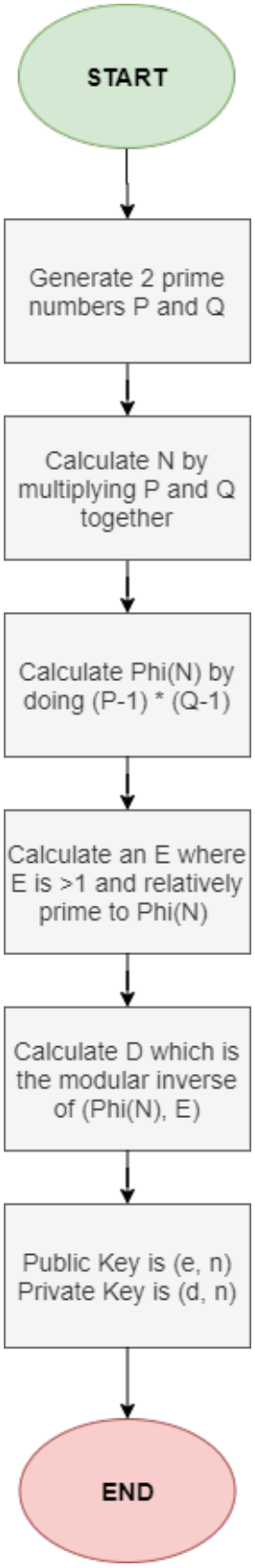
$$ciphertext = m^e\ mod\ n$$
$$where\ \mathbf{m}\ is\ a\ message\ block$$

STEP 7: To decrypt the cipher text:

$$message = c^d\ mod\ n$$
$$where\ \mathbf{c}\ is\ the\ cipher\ text$$

NOTE: RSA uses something that we call a **one-way trapdoor function**. This means it's easy to apply the function one-way but hard to invert it unless you know the secret/trapdoor. In the case of RSA, the calculating product of 2 large prime numbers (p, q) is easy to compute, but factoring the product (the inverse) is difficult. So the encryption cannot be broken unless the hacker knows p and q. [3]

3 Algorithm: Flowchart To Calculate Private and Public Keys



4 An Example: Alice and Bob

Scenario: Alice wants to send a message to Bob. The message contains the date of a secret coursework deadline: ***1303***

4.1 Encryption

Note: For the example we will be using small prime numbers so it is easy to compute but RSA in real life uses huge prime numbers

Bob generates his public/private key pair
First he generates 2 random prime numbers (p, q) and calculates n

$$p = 71, q = 41$$

$$n = 2911\ (because\ 71 * 41)$$

Next he calculates $\phi(n)$

$$\phi(n) = 2800\ (because\ (71 - 1) * (41 - 1))$$

He then selects a random e between 1 and $\phi(n)$ that's relatively prime to $\phi(n)$
The e we have chosen at random is:

$$e = 2473$$

Bob now has his public key that he can share to the world:

$$publickey = (2473, 2911)$$

Alice now uses Bob's public key to encrypt the date of the secret deadline: ***1303***

$$1303^{2473}\ mod\ 2911 = 360$$

THE CIPHER TEXT IS 360

4.2 Decryption

Bob receives the cipher text 360 and wants to decrypt it to get the original message Alice sent him.

Bob calculates d which is the inverse of e

$$d = 137$$

Bob now has his private key which only he knows:

$$privatekey = (137, 2911)$$

To decrypt the message he uses his private key:

$$(360^{137})\ mod\ 2911 = 1303$$

THE ORIGINAL MESSAGE IS 1303

Bob now knows the date of the secret coursework deadline

4.3 Special Case - Charlie The Hacker

4.3.1 Brute force - When the keys are not large enough

If the values for p and q are too small, Charlie can use brute force techniques and find d and ultimately decipher the message. Below outline some steps Charlie can take:

1) We must assume that Charlie knows e and n as they are public.
2) Charlie intercepts the cipher text
3) Charlie needs p and q to get the decryption key d
4) Charlie uses prime factorisation to find p and q from n
5) Charlie can now easily calculate $\phi(n)$ because he knows p and q
6) Charlie now has all the ingredients to find d, he has $p, q, phi(n), n$ and e so he can calculate the inverse
7) Charlie decrypts the message

5 Our RSA Implementation Demo & Explanation

Note: Our RSA implementation uses **long** values for simplicity. In real life, because numbers would be so large, you would use **BigInteger**.

Below is a class diagram outline our functions.

RSAAlgorithm	
- p: Long	+ RSAAlgorithm()
- q: Long	+ getD(): Long
- n: Long	+ getN(): Long
- phin: Long	+ getE(): Long
- e: Long	+ calculateN(p: Long, q: Long): Long
- d: Long	+ calculatePhi(p: Long, q: Long): Long
- eCounter: Long	+ gcd(a: Long, b: Long): Long
	+ getRandomE(phin: Long): Long
	+ getInverse(a: Long, b: Long): Long
	+ findPrime(maximum: Long): Long
	+ modularExponentiation(x: Long, n: Long, m: Long): Long
	+ encryption(e: Long, n: Long, m: String): BigInteger
	+ decryption(d: Long, n: Long, cipherText: BigInteger): String

Below is the output of our program when you run it (RSAAlgorithm.java).

```
Command Prompt

C:\Users\Tahmid\Desktop\CompSec Coursework>java RSAAlgorithm
-----------------------------------------------------------------------------------------

The first prime number is: 17
The second prime number is: 541
N is: 9197 (because 17 * 541)
Phi N is: 8640 (because Euler's Totient, 17-1 * 541-1)
There are 2303 possible e values that are below 8640 and coprime to 8640
The E chosen at random is: 1441
The inverse (D) is calculated: 7201

-----------------------------------------------------------------------------------------

Alice wants to send a message to Bob. The message contains the date of a secret meeting: 1303

1) Bob generates his public/private key pair. He shares his public key with the world and keeps his private key safe.
2) Alice uses Bobs public key to encrypt the message
3) Bob decrypts the message using his private key (7201)

-----------------------------------------------------------------------------------------

Encryption (public) key is: 1441 (e), 9197 (n)
--> Cipher text generated with encryption key: 9140

Decryption (private) key is: 7201 (d), 9197 (n)
--> Result generated from using decryption key on cipher text: 1303

-----------------------------------------------------------------------------------------

Charlie the hacker can brute force RSA when prime numbers and keys are small.
--> Charlie finds the prime factors of n by using factorisation algorithm.

He finds that 17*541 is 9197 (n)
He can now find phi(n): 16*540 = 8640
Charlie now has p, q, n, phi(n), and e so get can get d.
Charlie brute forces and finds the decryption key: 7201
```

Code Screenshots (5 Best Screenshots)

```java
/*
    This function encrypts the message, in other words, it turns plain text into cipher text
    Input: It takes 3 parameters, e and n values (which make up the public key of said person) and m,
    Output: It returns the cipher text
*/
static BigInteger encryption(long e, long n, String m) {
    BigInteger message = new BigInteger(m);

    BigInteger result = (message.pow((int)e).mod(BigInteger.valueOf(n))); //Run the actual encryption

    // Prints out the encryption key for demo purposes
    System.out.println("Encryption (public) key is: " + e + " (e), " + n + " (n)");
    // Prints out the cipher text
    System.out.println("--> Cipher text generated with encryption key: " + result + "\n");

    return result; //Returns the result - the cipher text
}
```

The above screenshot shows our implementation of the RSA encryption algorithm. The function takes a message m and encrypts it with the RSA encryption algorithm - $m^e \, mod \, n$. It returns the cipher text generated from the encryption.

```java
/*
    This function decrypts the message, in other words, it turns cipher text back into plain text
    It takes 3 parameters, d and n values (which make up the private key of said person) and c, the cipher text
    Output: It returns the plain text message
*/
static String decryption(long d, long n, BigInteger cipherText) {
    BigInteger oriMessage = (cipherText.pow((int)d).mod(BigInteger.valueOf(n))); //Run the actual decryption fun

    // Prints out the decryption key for demo purposes
    System.out.println("Decryption (private) key is: " + d + " (d), " + n + " (n)");
    // Prints out the message decrypted
    System.out.println("--> Result generated from using decryption key on cipher text: " + oriMessage + "\n");
    System.out.println("Bob now knows the secret message");
    System.out.println("\n-----------------------------------------------------------------------------------------

    return oriMessage.toString(); //Returns the result - the decrypted message
}
```

The above screenshot shows our implementation of the RSA decryption algorithm. The function takes a cipher text c and decrypts it with the RSA decryption algorithm - $c^d \, mod \, n$. It returns the message/result generated from the decryption.

```java
static long getRandomE(long phin) {
    ArrayList<Integer> eValueList = new ArrayList<Integer>(); //Creates a new arraylist
    for(int i = 2; i < phin; i++) { //Loops from 2 to the value of phi(n)
        if(gcd(phin, i) == 1) { //Checks if the greatest common dividor is 1
            eValueList.add(i); //If it is, adds it to the arraylist
            eCounter++; //Increments counter by 1, again, only for demo purposes
        }
    }
    Random rand = new Random(); //Creates new random object
    Integer randomEValue = eValueList.get(rand.nextInt(eValueList.size())); //Goes throug
    return randomEValue; //Returns the random e value
}
```

The above code shows our function to generate an e value. It takes $\phi(n)$ as input and checks if the $greatest \ common \ divisor$ of each number from 2 to $\phi(n)$ and $\phi(n)$ is equal to 1. **If it is, then it is a suitable candidate for e value.** For demo purposes we keep a counter of all suitable e values and store all suitable candidates in an arraylist and randomly pick out a suitable value from that list.

```java
/*
    This function is used to help find a suitable prime in the findPrime() function
    This is the modular eexponentiation algorithm Dr Ida Pu provided under week 5 on learn.
    This function was taken and adapted from: http://homepages.gold.ac.uk/rachel/CryptoTool
*/
public static long modularExponentiation(long x, long n, long m) {
    long y = 1, u = x % m;
    do {
        if(n % 2 == 1) {
            y = (y * u) % m;
        }
        n = (int)Math.floor(n/2);
        u = (u * u) % m;

    } while(n != 0);
    return y;
}
```

To compute $x^n \bmod m$:

Initialise $y = 1, u = x \bmod m$
Repeat
    if $n \bmod 2 = 1$ then $y = y \times u \bmod m$
    $n = n \; div \; 2$
    if $n \neq 1$ then $u = u \times u \bmod m$
Until $n = 0$
Output y

The above screenshot shows the fast modular exponentiation. It essentially computes $x^n \; mod \; m$ and helps us generate suitable prime numbers in the findPrime() function.

```java
/*
    This is the Extended Euclidean Algorithm
    This function calcuates the d value for RSA which forms the private key
    It takes 2 numbers (in the case of RSA, e and phi(n)) as parameters and returns the inverse
    The algorithm was provided by Dr Ida Pu under week 5 on learn.gold
*/
static long getInverse(long a, long b) {
    long store = a, temp, q, r = 1, s = 0;
    int sign = 1;
    while(b != 0)
    {
        q = a/b;
        temp = r;
        r = temp*q+s;
        s = temp;
        temp = b;
        b = a-q*temp;
        a = temp;
        sign = -sign;
    }
    if(sign == -1) {
        s = b-s;
    }

    return (r-s) % store;
}
```

```
static long FindInverse(long a, long b)
{
    long store = a;
    long temp;
    long q;
    int sign = 1;
    long r = 1;
    long s = 0;
    while(b! = 0)
    {
        q = a/b;
        temp = r;
        r = temp * q + r;
        s = temp;
        temp = b;
        b = a - q * temp;
        a = temp;
        sign = -sign;
    }
    long answer = (r - (sign * s))%store;
    return(answer);
}
```

The above shows the Extended Euclids Algorithm for generating the d value (in other words the inverse of e). The function takes 2 numbers as input ($\phi(n)$ & e) and calcuates the modular inverse. The algorithm was provided by Dr Ida Pu under week 5 on learn.gold.

5.1 Brute force special case

To maintain simplicity of our implementation, the prime numbers and keys we generate are too small for the algorithm to be secure. Charlie the hacker can brute force RSA when numbers are too small. An example of Charlie brute forcing is shown below. **Please refer back to section 4.3.1 for full explanation.**

We've included an example of Charlie brute forcing RSA in our implementation code:

```
public static long charlieBruteForce(long n, long e) {
    long originalN = n; //Creates a copy of n for demo purposes
    ArrayList<Long> isPrime = new ArrayList<Long>(); //Store the factors in an arraylist

    //This for loop checks which numbers are prime factors of n
    for (long i = 2; i <= n; i++) { //Goes through all numbers from 2 till n
        if (n % i == 0) { //Checks if modulous equals to 0
            isPrime.add(i); //If it is a prime factor of n, then it adds it to the arraylist
            n = n/i;
            i--;
        }
    }

    long charlieGetsPhiN = calculatePhi(isPrime.get(0), isPrime.get(1)); //Charlie now calcuates phiN because he know which 2 nu
    long charlieHasE = e; //Charlie already knows e because e is public
    long charlieGetsD = getInverse(charlieGetsPhiN, e); //Charlie breaks RSA and gets the decryption key!

    //Print statements to simulate scenario
    System.out.println("Charlie the hacker can brute force RSA when prime numbers and keys are small.");
    System.out.println("--> Charlie finds the prime factors of n by using factorisation algorithm.\n");
    System.out.println("He finds that " + isPrime.get(0) + "*" + isPrime.get(1) + " is " + originalN + " (n)");
    System.out.println("He can now find phi(n): " + (isPrime.get(0)-1) + "*" + (isPrime.get(1)-1) + " = " + charlieGetsPhiN);
    System.out.println("Charlie now has p, q, n, phi(n), and e so get can get d.");

    return charlieGetsD; //Return the decryption key
}
```

5.2 Program notes and assumptions

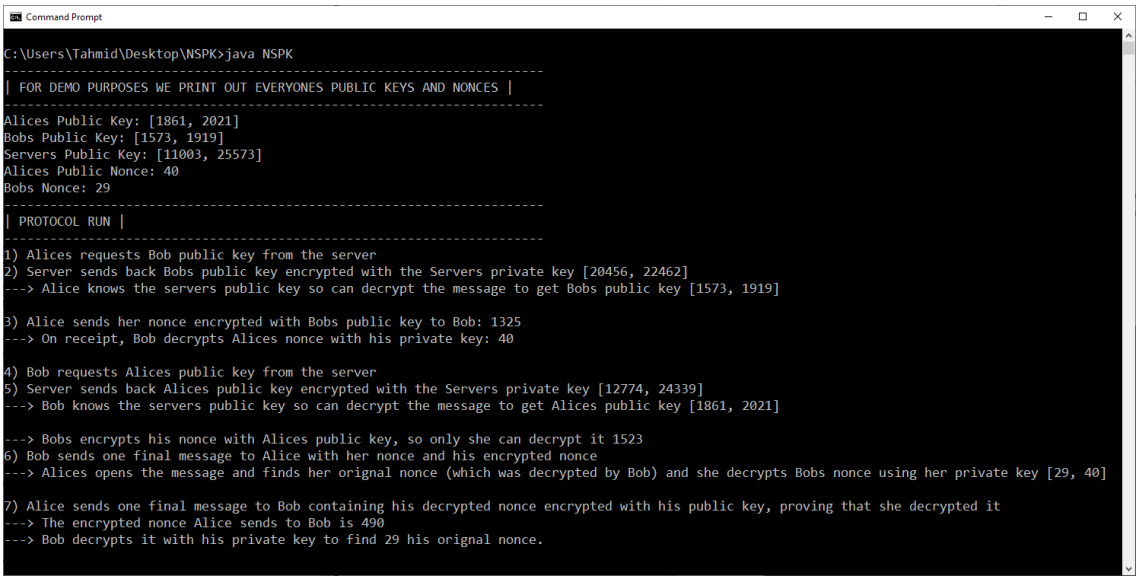
- Our RSA implementation uses long values for simplicity. In real life, because numbers would be so large, you would use BigInteger.
- Both prime numbers p and q would be chosen privately and discarded after e and d are calculated.
- In real life d value is private so only the sender has access to it. n and e are public.
- In the real world, for any random number generation, e.g generating a random prime number, there are specific cryptographically secure ways of generating truly random numbers. Random functions built into java are not cryptographically secure.

6 Needham-Schroeder Public Key Protocol

The Needham-Schroeder protocol is used to achieve mutual authentication between 2 parties. Our implementation of the protocol is based on the RSA algorithm we created in Part 1. It is simplified for demonstration purposes.

Assumption: The encryption is perfectly secure (absolutely secure).

Below is the output of our program when you run it.



Our program fulfills the 7 steps outlined in the coursework document, here's how it works:

1) Alice requests Bobs public key K_B from Server S
2) Server S sends back a signed version of Bob's key to Alice. In our case, the server signs the key by encrypting Bob's public key K_B using the Servers private key k_S.

This means when Alice gets the message from the server she can decrypt it using the Servers public key to retrieve the public key of Bob K_B

3) Alice then generates a random nonce n_A and encrypts it with Bobs public key K_B and sends it to Bob
4) Bob can decrypt the nonce n_A with his private key
5) Bob requests Alices public key K_A from Server S
6) Server S sends back a signed version of Alice's key to Bob. In our case, the server signs the key by encrypting Alices's public key K_A using the Servers private key k_S.

This means when Bob gets the message from the server he can decrypt it using the Servers public key to retrieve the public key of Alice K_A

7) Bob then generates a random nonce n_B and encrypts it with Alices public key K_A and sends it to Alice along with her decrypted nonce
8) Alice decrypts the message with her public key to find Bobs nonce n_B and her nonce K_A that she sent to Bob
9) Alice sends back Bobs nonce to him proving she decrypted it

6.1 Our Program Class Diagram

6.2 Code Snippets (4 Best Screenshots)

```java
/*
    This function simulates sending an encrypted nonce.
    It takes 3 arguments, the receiver (person), the key for the nonce to be encrypted with (this
    going to be the recivers public key) and the nonce
    It returns the encrypted nonce
*/
public BigInteger sendEncryptedNonce(Person receiver, ArrayList<BigInteger> key, Long nonce) {
    //This is essentially the RSA encryption algorithm happening (m^e mod n)
    BigInteger result = ((BigInteger.valueOf(nonce)).pow(key.get(0).intValue()).mod(key.get(1)));
    return result; //Return the encrypted nonce
}

/*
    This function decrypts the nonce using the persons private key
    It takes 1 argument - the nonce to decrypt
    It returns the decrypted nonce
*/
public BigInteger decryptNonce(BigInteger nonce) {
    //This is essentially the RSA decryption algorithm happening (c^d mod n)
    BigInteger result = (nonce.pow((int)this.getD()).mod(BigInteger.valueOf(this.getN()))); //Take
    return result; //Return the decrypted nonce
}
```

The above code shows the functions in the $Person$ class used to send an encrypted nonce and to decrypt a nonce that was sent to them. $sendEncryptedNonce()$ takes a nonce and encrypts it with the the key passed in the parameter. **In our case of NSPK, it should be encrypted with intended recipients public key.** Likewise, the $decryptNonce()$ function takes a nonce as an argument as uses the persons private key to decrypt the message back to the original nonce. The encryption algorithm used is that of RSA's – $m^e \ mod \ n$. The decryption algorithm used is that of RSA's – $c^d \ mod \ n$.

```java
public ArrayList<BigInteger> decryptKeyFromServer(ArrayList<BigInteger> encryptedKey, Server s) {
    ArrayList<BigInteger> decryptedKey = new ArrayList<BigInteger>(); //Arraylist to store the dec

    //Get the public key of the server
    BigInteger serverEValue = BigInteger.valueOf(s.serverGetPublicKey().get(0));
    BigInteger serverNValue = BigInteger.valueOf(s.serverGetPublicKey().get(1));

    //This is essentially the RSA decryption algorithm happening (c^d mod n)
    //Decrypt key using the servers public key
    BigInteger result = (encryptedKey.get(0).pow(serverEValue.intValue()).mod(serverNValue));
    BigInteger result2 = (encryptedKey.get(1).pow(serverEValue.intValue()).mod(serverNValue));

    //Add decrypted key to arraylist
    decryptedKey.add(result);
    decryptedKey.add(result2);

    return decryptedKey; //Return the decrypted public key
}
```

The above code shows the $decryptFromServer()$ function - this function is used by a person after the server sends back the encrypted key of the person they want to communicate with. The person takes the encrypted key the server sent them and decrypts it using the servers public key to get public key of the person they want to communicate with.

```java
private String name; //Variable to store person name
long nonce; //Variable to store person nonce
RSA_Helper rsa; //Each person will need RSA public private keys

/*
    Constructor to instantiate a person
    It takes 1 parameter - name of the person
*/
public Person(String name) {
    this.name = name; //Assigns person name to variable
    this.rsa = new RSA_Helper(); //Creates new RSA instance for person, this gives us a publ

    /*
        Generates a random nonce for the person between 1 and 50
    */
    Random rand = new Random(); //New random object
    nonce = rand.nextInt((50 - 1) + 1) + 1; //Pick random number from 1 to 50 and let it be
}
```

The above code shows the constructor of the $Person$ class. For each person created, they get assigned a name, an instance of the RSA class (this helps us generate their public and private keys) and they get assigned a random nonce between 1 and 50.

```java
public ArrayList<BigInteger> requestKey(Person p){
    ArrayList<BigInteger> encryptedKey = new ArrayList<BigInteger>(); //Create new arraylist to store the encrypted key

    long personPKE = getPublicKey(p).get(0); //Get the public key pair of the person requested
    long personPKN = getPublicKey(p).get(1); //Get the public key pair of the person requested

    long serverPrivateKey1 = rsa.getD(); //Get the servers private key
    long serverPrivateKey2 = this.getN(); //Get the servers private key

    //This is essentially the RSA encryption algorithm happening (m^e mod n)
    BigInteger encryptedEVal = ((BigInteger.valueOf(personPKE)).pow((int)serverPrivateKey1).mod(BigInteger.valueOf(serverPrivateKey2)));
    BigInteger encryptedNVal = ((BigInteger.valueOf(personPKN)).pow((int)serverPrivateKey1).mod(BigInteger.valueOf(serverPrivateKey2)));

    encryptedKey.add(encryptedEVal); //Add the encrypted key pair to the arraylist
    encryptedKey.add(encryptedNVal); //Add the encrypted key pair to the arraylist

    return encryptedKey; //Return the arraylist
}
```

The above shows the $requestKey()$ function from the $Server$ class. This function essentially gets the server to sign the public key of a person with its own private key, proving to the person who decrypts it (with the servers public key) that it was indeed sent by this server and not another one.

7 Conclusion & Discussion

7.1 Needham-Schroeder Protocol Man-in-the-middle Attack

The Public Key version of the Needham-Schroeder Protocol is vulnerable to a man-in-the-middle attack. In other words, Charlie our hacker will execute 2 instances of the protocol, one with Alice and one with Bob. He will pretend to be Alice when interacting with Bob.

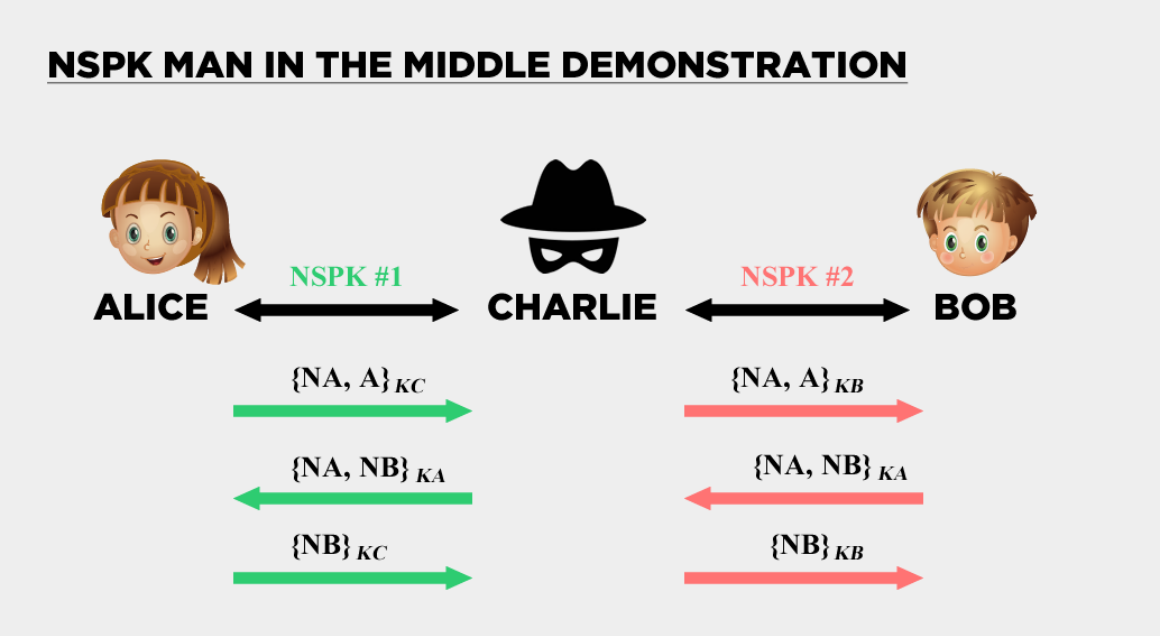
This means after Alice initiates a session with Charlie, he can pass on those messages to Bob. Bob would believe that he is in communication with Alice. [4] This whole problem stems from the fact that Alice does not know the identity of who she is communicating with. This can be shown in the diagram we created below.

As mentioned, the problem comes from not knowing the identity of the person you're communicating with. There was a fix released in 1995 for the protocol that involves Bob sending his identify back to Alice along with Alice's nonce and his nonce. [4] The solution looks something like this:

$B \rightarrow A : \{N_A, N_B, B\}_{K_A}$

If Alice realises the message is not from whom she is intending to speak to (Bob) then she can simply abort the protocol execution and ignore the message.

Another solution to the vulnerability is the Kerberos protocol which is based on NSPK but does not have this flaw. [5]



An original diagram created by us

7.2 Coursework Experience & Final Remarks

We would like to thank **Dr Ida Pu** for teaching computer security in a fun and intuitive way this year. She has always taught us to think outside the box and think bigger and we've gained valuable insight into computer security for the real world. One of the key takeaways that we've learnt and will never forget is when evaluating security systems is to ask 2 main questions:

1) Does it work?
2) Does it *really* work?

Our overall experience with the coursework was good. We found the coursework to be challenging - but in a good way. Online resources were a big help. The RSA algorithm was fairly straight forward however we did struggle to grasp our heads around the NSPK protocol for a little bit. The mathematics in the module can be fairly challenging at times but Dr Ida made it easy to understand with her slides and explanations.

One of the problems we ran into when coding the Needham Schroeder protocol was when we were generating the public and private keys of Alice, Bob and the Server using our RSA algorithm. Dr Ida taught us that in RSA the message should not be larger than the public key of the entity that's encrypting the message. In our case, the Server was encrypting the public key of Bob and Alice and sending to each other. In some instances, the keys of Alice and Bob were larger than the Servers Public Key (n) and therefore the cipher would not work properly. We invented a quick work around to this, a simple if statement to check if Alices and Bobs keys are larger than the servers, if they are then keep on generating new keys until the keys of both parties are smaller than the server.

In the future, if we were to improve our programs, we would implement RSA with $BigInteger$ instead of $long$. If we had more time, we would have created a graphical user interface (GUI) representing how the communication works and tried to improve the time and space complexity of some of our functions/algorithms.

8 References

[1] https://cgi.csc.liv.ac.uk/ alexei/COMP522_16/COMP522-RSA-16.pdf
[2] https://crypto.stackexchange.com/questions/47335/what-is-the-key-size-currently-used-by-rsa-and-diffie-hellman-for-secure-communi
[3] http://mathworld.wolfram.com/TrapdoorOne-WayFunction.html
[4] https://en.wikipedia.org/wiki/Needham
[5] https://en.wikipedia.org/wiki/Kerberos_(protocol)