Nathaniel Guevara (ntg17)
Akash Nayak (asn66)
10/22/19

Information and Network Security Computer Project 1

Overview

In this project, we were tasked with creating an algorithm in Matlab that can implement RSA encryption. In this implementation, we had to create an algorithm that would not only randomize the private key, e, but also the public keys, p and q.

As stated in the instructions file, we started to use the fundamental Matlab functions. We implemented modules that generated two primes (public key encryption exponent *e* and the corresponding decryption exponent *d*). We then asked the user to input plaintext m, and then convert m to a numerical representation. This was then used to find the ciphertext $c = m^e \pmod n$, and the decryption $m = c^d \pmod n$. For this, there will need to be some handling with the restrictions of Matlab and working out algorithms and factors along the way.

Approach and Implementation

The first part of implementing this encryption is creating randomized keys. An algorithm was needed to not only randomize the private key but also the public keys. In order to create a random prime number for p and q, we created an infinite loop, that would only end when a prime number has been found. We thought of using an infinite loop, because we didn't know how many times we would have to create random numbers, until one of them was prime, so an infinite loop would solve that. This way, the loop would always create random numbers until a prime number is found and the loop is broken out of. In the first step of this loop, you would create a random positive odd integer using the randi() function in matlab, where you set the max value and min that can be created. We wanted to only check odd numbers, since all even numbers are divisible by 2 making them composite (except for 2 itself), so we didn't want to slow down the run time by making even numbers. To make sure that the number is odd, we decreased the max value of the randi() function by a factor of 2, so that we can multiple the number created by this function by 2 and add 1. This guarantees that the number is odd because all even numbers are divisible by 2 so by multiplying the random number by 2, it guarantees that the result is even, so that when we subtract 1 from that result, it would be odd. This max value was initially set to be $2^{82589933}-1$, which is the largest known prime. However this is where the first error occurred. In matlab for the randi() function, the largest input you can use is 2^53. This is a huge limitation in matlab, as numbers less than 2^53 are significantly easier to break than ones that can be as big as $2^{82589933}-1$, due to modern day computing power being strong enough to brute force all the keys to decrypt the message in a reasonable time. To check to see if the number is prime, we used Fermat's Little Theorem. In order to do this, we had to implement the formula for Fermat's Little Theorem, which is $a^{n-1} \pmod n = 1$. In this formula, the number to test to see if it is prime or not is n, and a is a random number. To test this formula, we had to create a

random number, a, to test to see if this formula is true. Here we ran into another problem. If the value of a and/or n was to big, then it would overflow Matlab and would be treated as infinity. In order to solve this issue, we had to alter the formula in a way that it would still be equivalent to the formula, but you wouldn't have to worry about getting overflow errors from Matlab. To do this, we choose to create a loop, where you would multiply the value of a with another variable z. This z variable would start out as a^2 (mod n). This loop would continue until you have multiple a to z enough times so that z contains a^{n-1}. However this is not all. We also had to take z mod n on every iteration of the loop so that the value of z would not become too big that it would overflow Matlab. However, even with this method, Matlab would overflow, so we had to reduce the max possible value of p and q to $2^{12}+1$ in order to fix this. This further reduces the security of the encryption, by lowering the number of possible keys, making it easier for an attacker to test every possible key in a reasonable time. After we finally reached a^{n-1}, then we check to see if the number is composite or not. If z = 1, then the formula would be true and the number would be composite. Since this only checks to see if a number is composite rather than if it is prime, we had to run Fermat's Little Theorem multiple times. In order to do this, we created a loop to hold the Fermat's Little Theorem check, so that we can run it multiple times. We implemented this with a while loop with a counter, so that every time the loop ran successfully, then it would add to the count. Once a certain count has been reached, then the loop would end, and the number would be considered prime. If a number was found out to be composite, then there is no reason for the loop to continue, so a break was added. In order for the infinity loop to be broken, there needs to be a prime number, so we created a boolean called isPrime. Every iteration of the infinite loop, it would set isPrime as true. Then, if the number was found out to be a composite number, then the boolean would be set to false. After the loop to check Fermat's Little Theorem ran, if the boolean is still true, then the random number is prime and will be used as p or q, and you would break out of the infinite loop. Also before we break out of the infinite loop for the second key q, we had to check to make sure the number wasn't the same as p so we placed another condition to check that as well, before breaking out of the loop. Once p and q have been found, they are then printed out using fprintf.

Next we had to create a random key e. To do that, we first calculated n and ϕ(n) from the values of p and q. From there, we would calculate e. Since we also don't know how many times it would take to create a random number that satisfies the condition for the private key e, we created another infinite loop, so that random numbers would be generated until one of them satisfies the conditions for e. Inside the infinite loop, we would start by creating a random integer using randi(), with a max value of ϕ(n). Then we created a boolean expression (logical statement) where it checks if the gcd of this random number and ϕ(n) is 1. This is to satisfy the condition that the key, e, has to be relatively prime with ϕ(n). If it is true, then you break out of the loop. If not, then the loop would run again until it does find one. Once e is found, it is printed out using fprintf.

After that, we needed to create an algorithm to create the decryption key, d, from the encryption key, e. Since we once again didn't know how many random numbers it would take to find one that suits the condition of d, we created an infinite loop, where the only way to break is to get d. To get d, we set a counting variable x that starts at 0, then increases by 1 with every iteration of the loop. With that we would make a variable that equals 1 plus x times ϕ(n). Then we would test to see if the mod of this variable and e is 0. This was done because d has to have congruence relation with e so that d times e $\equiv$ 1 (mod ϕ(n)), and d has to be an integer. If this statement is true, then we would set d as the division between the number generated and e and break out of the loop. Once d is found, it is printed out with fprintf.

Then, we needed to create an algorithm to take a users input, and encrypt it. First we create a prompt for the user that asks them for a message to encrypt. Then we create a variable that takes in the user's input after displaying the prompt. After receiving the message, we had to convert the message from a string into an array of doubles. This is then printed out as a string to double check to see if it matches the user's input message matches. Next we have to create an empty array of the same length as the inputted ciphertext, for the encrypted text to go in. Finally, a loop was created to apply the formula for RSA encryption, $c = m^e \bmod n$, to each value in the plaintext array, and place those values in the corresponding index of the ciphertext array. However, we ran into an issue here, when solving $c = m^e \bmod n$. Sometimes, it would take too long to solve this algorithm, because e is very large. To help remedy this issue, we changed the algorithm slightly. We implemented the same strategy that we used to check for prime numbers with Fermat's Little Theorem, where we multiply m to itself, e times, while taking the (mod n) of it every iteration. This formula multiplies m to itself until you've reached m^e. This allows Matlab to achieve $m^e \pmod n$ without reaching an overflow error. We then create a loop that iterates through every index in the encrypted text to print it out for the user to see/send.

Lastly, we had to implement the decryption algorithm. To do this, we implemented the same strategy that we used for the encryption algorithm, but with the formula, $m = c^d \bmod n$. As before, when we implemented the formula directly, it wouldn't work because of overflow, so instead we implement the same strategy as before but with multiplying c by itself while taking the mod n of it until we reached c^d. Here we ran into another issue, when n was less than 2^8. Because a char is 1 byte long, the value of that char would change if it was modded with a number less than 2^8, so we had to set the lower limit of p and q to 2^4 so that n would always be big enough not to affect this. To end the program, the decrypted plaintext will be printed using the fprintf() function.

Testing

As a note for all of our tests, since we don't know what our p, q, e, and d values are before running each test, as they are randomly generated, they will not be included as part of our input. However their value will be shown as part of the output that we took directly from the command window. In order to test to see if the program works properly or not, we had to see if

the process could encrypt a message, and decrypt that same message, while keeping the message exactly the same.

For our first test, we wanted to test whether the function works as intended to. So we just wanted to input a letter to see if the algorithm works or not.

Test case 1: Input - a (which is the plaintext)
Expected Output - a
Actual Output -
p value is: 2791
q value is: 4091
n value is: 11417981
phiN value is: 11411100
e value is: 11326139
d value is: 2855159
What message would you like to encrypt?
a
Plaintext: a
Ciphertext: 940555
Decrypted text: a

For our next test, we wanted to test multiple letters at once to see if the algorithm can handle more than 1 letter, so we used a word.

Test case 2: Input- hello (which is the plaintext)
Expected Output - hello
Actual Output -
p value is: 2617
q value is: 509
n value is: 1332053
phiN value is: 1328928
e value is: 1287139
d value is: 330475
What message would you like to encrypt?
hello
Plaintext: hello
Ciphertext: 1383848927503920963920961111421
Decrypted text: hello

This next test was made to test if the algorithm can handle case-sensitive letters.

Test case 3: Input - AmIRite (which is the plaintext)
Expected Output - AmIRite

Actual Output -
p value is: 3299
q value is: 1871
n value is: 6172429
phiN value is: 6167260
e value is: 3107943
d value is: 3427087
What message would you like to encrypt?
AmIRite
Plaintext: AmIRite
Ciphertext: 213439950052226065631521015841325153645190218283 7
Decrypted text: AmIRite

For the next test, we wanted to see if numbers also work in our algorithm.

Test case 4: Input - 1234567890 (which is the plaintext)
Expected Output - 1234567890
Actual Output -
p value is: 433
q value is: 1699
n value is: 735667
phiN value is: 733536
e value is: 355297
d value is: 384865
What message would you like to encrypt?
1234567890
Plaintext: 1234567890
Ciphertext: 495772472048634787340777126536438443948167025442735626775 8
Decrypted text: 1234567890

For the next test, we tested all of the symbols to see if they work.

Test case 5: Input - `~!@##$%^&*()-_=+[{]}\|;:'",<.>/? (which is the plaintext)
Expected Output - `~!@##$%^&*()-_=+[{]}\|;:'",<.>/?
Actual Output -
p value is: 2111
q value is: 3083
n value is: 6508213
phiN value is: 6503020
e value is: 3261521

d value is: 4131701
What message would you like to encrypt?
`~!@##$%^&*()-_=+[{]}\|;:'",<.>/?
Plaintext: `~!@##$%^&*()-_=+[{]}\|;:'",<.>/?
Ciphertext:
3148164307756434514501791093374221374221495303162147652664499228866643226812286321767097481366735269796299125354778842052038078222822630478121657929631214733179628123645126732675702177543849441879736216776480874048986985816008
Decrypted text: `~!@##$%^&*()-_=+[{]}\|;:'",<.>/?

For our next test, we wanted to test spacing and tabs.

Test case 6: Input - Hello world I am here
Expected Output - Hello world I am here
Actual Output -
p value is: 2459
q value is: 823
n value is: 2023757
phiN value is: 2020476
e value is: 1949347
d value is: 1515847
What message would you like to encrypt?
Hello world I am here
Plaintext: Hello world I am here
Ciphertext:
153846314049192246092246093826632821101501352382663113436022460982436348217396398528211071152315175528211013727951404919113436014049 19
Decrypted text: Hello world I am here

As a note for this test, the runtime for this was more than a second.

For our next test, we wanted to test all of the other parts that we test all together.

Test case 7: Input - The Formula Is: f(x) = (d/dx) e^4x + 5
Expected Output - The Formula Is: f(x) = (d/dx) e^4x + 5
Actual Output -
p value is: 1151
q value is: 1283
n value is: 1476733
phiN value is: 1474300
e value is: 622131

d value is: 534871
What message would you like to encrypt?
The Formula Is: f(x) = (d/dx) e^4x + 5
Plaintext: The Formula Is: f(x) = (d/dx) e^4x + 5
Ciphertext:
10194133209891323506100473010880572376464773174356802392601029120224891004730135182713590396169814570310138697414948745559608110047302416901004730741494127494141940812749487455596081100473013235065916442867008745551004730218016100473048229
Decrypted text: The Formula Is: f(x) = (d/dx) e^4x + 5

For this test, we wanted to test to see what happens when you don't input anything.
Test case 8: Input -
Expected Output -
Actual Output -
p value is: 1693
q value is: 2389
n value is: 4044577
phiN value is: 4040496
e value is: 565399
d value is: 2409895
What message would you like to encrypt?

Plaintext:
Ciphertext:
Decrypted text:

Finally, we wanted to encrypt a really long message to test to see if Matlab could encrypt it in a reasonable time or not.
Test case 9: Input - As stated in the instructions file, we started to use the fundamental Matlab functions. We implemented modules that generated two primes (public key encryption exponent *e* and the corresponding decryption exponent *d*). We then asked the user to input plaintext m, and then convert m to a numerical representation. This was then used to find the ciphertext c = me (mod n), and the decryption m = c d (mod n). For this, there will need to be some handling with the restrictions of Matlab and working out algorithms and factors along the way.
Expected Output - As stated in the instructions file, we started to use the fundamental Matlab functions. We implemented modules that generated two primes (public key encryption exponent *e* and the corresponding decryption exponent *d*). We then asked the user to

input plaintext m, and then convert m to a numerical representation. This was then used to find the ciphertext c = me (mod n), and the decryption m = c d (mod n). For this, there will need to be some handling with the restrictions of Matlab and working out algorithms and factors along the way.

Actual Output -

p value is: 1153
q value is: 727
n value is: 838231
phiN value is: 836352
e value is: 230855
d value is: 183287
What message would you like to encrypt?

As stated in the instructions file, we started to use the fundamental Matlab functions. We implemented modules that generated two primes (public key encryption exponent e and the corresponding decryption exponent d). We then asked the user to input plaintext m, and then convert m to a numerical representation. This was then used to find the ciphertext c = me (mod n), and the decryption m = c d (mod n). For this, there will need to be some handling with the restrictions of Matlab and working out algorithms and factors along the way.

Plaintext: As stated in the instructions file, we started to use the fundamental Matlab functions. We implemented modules that generated two primes (public key encryption exponent e and the corresponding decryption exponent d). We then asked the user to input plaintext m, and then convert m to a numerical representation. This was then used to find the ciphertext c = me (mod n), and the decryption m = c d (mod n). For this, there will need to be some handling with the restrictions of Matlab and working out algorithms and factors along the way.

Ciphertext:

4940314622417763674622412838701417428387073192546603577636751838551555777636728387075957319257763675183855155574622412838705069162680025943852838705183852419845155574622413508397763674092627319257763675183855654075036811325573192556540731925515557283870731925466035776367565402419844660352680021132557319254622417763672838707595141742838707763677873873192551555773192556906916141742838707319254660357763672838707334324198477636775036850691651838556540731925462241776367756139750368268002818434113255518385594385776367536986731925441857776367731925515557594385506916441857750368283870518385241984515557776373192560238575036824198451555773192551555728387077636773192577636714174515557466035776367283870759573192577636759438524198450691650691673192546224175036827595731925776367594385241984506916506916731925462241750368274660357763672838707595731925776367594385241984506916506916731925462241750368

41984515557466035518385515557787387763674660357319255943855069164418577503682838705183852419845155577763677319256023857503682419845155577319255155572838707763674660352590723508397763674092627319257763672838707595731925515557776367141744622415369867319254660357763672838707595731925776367268002462241731925506916776367283870241984776367518385515557750368268002283870776367750368113255141745183855155572838707319256023852838707763675654025920877636714174515557466035776367283870759573192551555777636759438524198451555783709773192550691628387077636756540776367283870241984776367141747763675155572680025654073192550691651838559438514174113255776367506916731925750368506916731925462241731925515557283870141742838705183852419845155573508397763673440907595518385462241776367733431417446224177636728387075957319255155577763672680024622417319254660357763672838702419847763676627705183855155574660357763672838707595731925776367594385518385750368759573192550691628387073192560238528387077636759438577636757454177636756540731925776367756139565402419844660357763675155572590722592087763671417451555746603577636728387075957319257763674660357319255943855069164418577503682838705183852419845155577763675654077636757454177636759438577636746603577636775613956540241984466035776367515557259072350839776367451827241984506916776367283870759551838546224125920877636728387075957319255069167319257763677334351838511325511325577636751555773192573192546603577636728387024198477636781843473192577636746224124198456540731925776367759514174515557466035113255518385515557787387763677334351838528387075957763672838707595731925776367506916731925462241283870506916518385594385283870518385241984515557462241776367241984662770776367551597141742838701132551417481843477636714174515557466035776367733432419845069165369865183855155577873877636724198426800228387077636714174113255787382419845069165183852838707595565404622417763671417451555746603577636766277014174594385283870241984506916462241776367141741132552419845155577873877636728387075957319257763677334314174441857350839

Decrypted text: As stated in the instructions file, we started to use the fundamental Matlab functions. We implemented modules that generated two primes (public key encryption exponent e and the corresponding decryption exponent d). We then asked the user to input plaintext m, and then convert m to a numerical representation. This was then used to find the ciphertext c = me (mod n), and the decryption m = c d (mod n). For this, there will need to be some handling with the restrictions of Matlab and working out algorithms and factors along the way.

Also the run time on this test was about 5 seconds.

For all of these tests, the expected outcome matched the actual output, making all of our tests successful.

Conclusion

In this project, we successfully created a working RSA encryption in Matlab. We even let the user know what all of the keys used in making this encryption. However there were many limitations in using Matlab that greatly affect not only the speed of the encryption, but also the security of it as well. Because of Matlab's double precision, we could not make this encryption secure enough for people to use, due to the relatively low number of possible keys. Since there are a low number of key combinations, attackers could easily try every key and figure out the plaintext. We do not recommend people to actually implement RSA encryption with Matlab. With that said, the algorithm itself worked out very well. After making numerous amounts of changes to the code, in order to get it to function, it worked out the way that we wanted it to. It passed all of our test cases, making it able to work in any case, and in a reasonable time as well. Overall, we created a working RSA encryption in Matlab however we do not advise others to actually implement this encryption using Matlab, for security reasons.