Joshua Siegel

Phurushotham Shekar

Professor Trappe

ECE:424 Intro to Information & Network Security

October 22, 2019

RSA Encryption and Decryption

For this project, we had to implement our own version of the RSA encryption algorithm in MATLAB. To do this, we used multiple functions:

- **expnFast(b,p,m)**:
  - Input: base, power, and mod.
  - Output: b^p mod m
- **getPrime():**
  - Output: Random prime # between 370 and 620
- **encrypt(plaintext):**
  - Input: our message in plaintext
  - Output: ciphertext, d, n
- **decrypt(ciphertext, n, d):**
  - Input: ciphertext to decipher, n and d (same outputs as encrypt function)
  - Output: original plaintext that was input to encrypt

expnFast(b,p,m):

We first needed a way to do exponentiation accurately. Since when we take large exponents mod n, such as 10^{500} mod 16, if we evaluated the exponent before modding, then the number would overflow MATLAB's capacity. Therefore, to perform RSA accurately, we needed to write our own exponentiation function and mod function. Consider evaluating m^e mod n.

We initially completed this by multiplying m by a temporary variable, and modding it by n, and repeating this e times (shown in our **expn** function, which is not actually used in the algorithm, since it was replaced by **expnFast**). This worked, however it was slow. Therefore, to speed it up, we implemented modular exponentiation where we kept a temporary variable, and squared it until it gets as close to m^e as possible, and modding by n after each square. We determined the number of times to perform the squaring operation by calculating floor($\log_2(p)$) as this would give us the maximum number of times we could square without going over. Once we have squared enough, we just multiply by b and mod m for the remaining amount of operations. This method runs in O(log(p)) instead of O(p) like our initial method. Now with this custom modular exponentiation function, we are able to implement RSA encryption and decryption.

For example to calculate 10^{19}:

- floor($\log_2(19)$) = 4
- So in a loop we take the base 10 and square it 4 times which results in 10^{16}
- Then we manually multiply that result by 10 for (19-16) = 3 more times in order to arrive at 10^{19}.

getPrime():

We then wrote a function to create a random prime by picking a random number between 370 and 620, and then performing the Fermat's Primality Test 10 times. We do this by taking the numbers 2 to 12, raising it to the randomly generated number - 1 mod p, and checking if they are all 1. If the randomly generated number passes the Fermat Primality Test with the numbers from 2 to 12, then we can assume with high probability that the number is prime. We call getPrime() twice in order to generate **p** and **q** (our primes). If the second generated prime happens to be the same as the first generated prime, we generate a different prime so that both of the prime numbers are not the same.

encrypt(plaintext):

To encrypt, we first generate two different random primes by calling **getPrime()**, and making sure the two primes are different. If they are the same, we keep calling **getPrime()** until it gives us two different random primes. We then calculate **phi**, by multiplying (**p**-1)*(**q**-1). This then allows us to generate **e**. We generate **e** by picking a random integer between 2 and **phi** - 1, and then calculating the **gcd(e, phi)**. While the gcd(**e**, **phi**) is not 1, we calculate a different random **e**. We do this until we get our public encryption key **e**. We know that this is a valid key since it is relatively prime to **phi**, which we checked by setting the gcd to 1.

Now that we have our public encryption key **e**, we can calculate our private decryption key **d**, by using the extended euclidean algorithm. The extended euclidean algorithm works backwards from finding the greatest common divisor, and allows us to find the Bezout's coefficients for **e** and **phi**. The Bezout's coefficients are the two coefficients, let's call them **d** and a, such that **d***e** + a***phi** = gcd(**e**,**phi**), and in this case this gcd(**e**, **phi**) is 1. Luckily, MATLAB's gcd function will find the Bezout coefficients for us, and all we have to do is pass in **e** and **phi**. Once we get the two Bezout coefficients, we just need to determine which one is **d**, and which one we can throw out, since we don't need the other coefficient. We don't need the other coefficient because it

is multiplying **phi**, and we are doing our operations mod **phi**, and anything times **phi** mod **phi** is 0. We realized that MATLAB's gcd function always returns the Bezout's coefficients in the same order that we pass the variables into the gcd function, so we know that the private decryption key **d** is the first Bezout coefficient that is returned.

We ran into a problem that sometimes the private decryption key **d** was negative, and this caused problems in our decrypt method, because we were trying to raise the ciphertext to a negative exponent mod **n**, and it was breaking our exponentiation method. To resolve this, we check if **d** is negative, and if it is, we set it to be equal to **d** mod **phi**. Since we are doing operations mod **phi**, we can get the positive version of the key by taking the decryption key mod **phi**. Now we have **e**, **n**, **phi**, and **d**. We are ready to encrypt the plaintext message that was given to us.

We take the string passed into the encrypt function, and look at its characters one at a time. We take the ASCII value of the character, and get the encrypted character by calling **expnFast(character, e, n)**. This raises the ASCII value to the $\mathbf{e}^{th}$ power, mod **n**, and it does it in log(**e**) time. Now we do this for each character in the plaintext passed into the function, and we get the cipher values for each character. These cipher values are integers, and they represent the encrypted version of the character. We now return the array of

ciphertext, and so we can decrypt the ciphertext later, we also return the private decryption key **d**, and the public key **n**. We can then pass these values into the decrypt function to decrypt the ciphertext.

decrypt(ciphertext, n, d):

Since the ciphertext is stored as an array of numerical values of encrypted ASCII characters, we run decrypt by performing the decryption algorithm character by character and then concatenating them. The decryption algorithm uses the **fastExpn()** function to perform RSA decryption with $\mathbf{c}^{\mathbf{d}}$ mod **n**. The inputs are ciphertext (**c**), **n** a

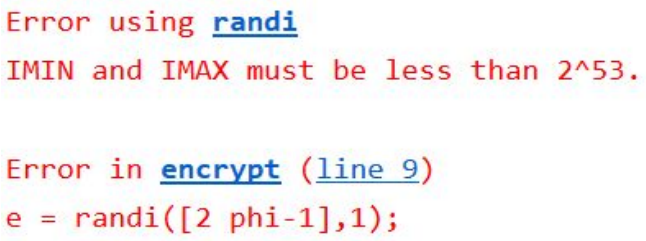
nd **d**, and we use all 3 inputs of **decrypt(c,n,d)** in order to run decryption and calculate the plaintext values. The decrypted plaintext value is then appended into an array called **plainText** and we return the array of characters from the decrypt function. Since a string is essentially an array of characters, the output, **plainText** is just the string of the original plaintext.

Testing:

When we first wrote our encryption and decryption functions, we tested it on very small input, and it worked fine. But then we decided to stress test it with very large input to see what would happen. We copied and pasted multiple paragraphs from the script of the bee movie and our decrypt function

broke, so we thought that the problem was with the large input. However, after some more testing we noticed it failed sometimes for small input too. After that, we realized this was because we were returning a negative value for the decryption exponent **d**, and that was breaking our fast exponentiation function. To fix this, we took **d** mod **phi** to prevent **d** from being negative, but preserved its ability to be a decryption key. After this fix, we were able to successfully encrypt and decrypt all inputs, including large texts.

To find the maximum number that could be squared, we started off with squaring numbers around 2^{31} since Matlab has a max value of 2^{64}. The numbers that we tested were incremented by 100,000, and we tested if the square of this number would still be represented as an integer. This took too long so we started incrementing by 10 million. Then once we found that MATLAB stored the result as a float, such as 9.223372e+18, we started narrowing down our boundaries until we found that 3,037,000,499 is the largest number that we could square, and the square is: 9223372030926248960. This is the largest squared value that MATLAB can store as an integer.



[1]The above image displays the error when trying to find a random number between 2 and **phi**-1. If **phi** is too big, we needed to select a **p** and a **q** that are below 2^{26} in order for phi to not be greater than 2^{53}

Using this knowledge, we thought that if we used an encryption key of **e** = 2, the largest primes we could use for **p** and **q** are: **p** = 3037000493; **q** = 3037000453; as these are largest primes less than 2^{26} (where multiplying them would result in a **phi** less than 2^{53} which is the limitation of randi), such that the square is still represented by an integer[1]. However since the encryption key has to be relatively prime to (**p**-1)*(**q**-1), we realized that 2 wouldn't work as it will never be relatively prime to (**p**-1)*(**q**-1). Since **p** and **q** are always odd, (**p**-1) and (**q**-1) are always even, and the multiplication of even numbers is always even, therefore they will always have 2 has a factor.

We also then realized that when we use very large primes, we sometimes get either a very large encryption key or decryption key, even on the order of 10^9. This caused our exponentiation function to take a very long time since it's raising something to the 10^9 power mod **n**. This is how we realized that we need to use smaller primes, so that our encryption or decryption doesn't take a very long time to exponentiate.

As a final test case, we passed this entire lab report as the input to encrypt (after removing all newlines and apostrophes), and then when we call decrypt, we got our report back!