Dillon Roller

CSC-412

Cryptography Portfolio

Intro

My name is Dillon Roller. I am a computer science major and math minor at the South Dakota School of Mines and Technology. This semester, I am taking a cryptography course to further apply and exercise my coding skills to real word applications. In this portfolio, I will be showcasing and explaining all parts of my code and modules, along with test files and how to run them.

Number theory is an abstract concept, and most people think that something so abstract can't possibly be useful, but that's why it's so important and interesting to me. It's really hard to get started but once you understand the necessary background information in abstract algebra, there are vast applications of it in cryptographic areas all around the world.

INFO:

Email: dillon.roller@mines.sdsmt.edu

Phone: 320-444-5910

Location: Rapid City, SD

Date: 12/16/2019

Files

Alongside this PDF is a folder with all the necessary files needed to run and use my cryptographic functions. The two most important files are:

- CryptoMath.h
    - This header file contains all function declarations for all functions used throughout my cryptographic course. This file must be inside the working folder when you compile
- CryptoMath.cpp
    - This file contains the implementation of all functions used throughout my cryptographic course. This file must also be inside the working folder when you compile.

All other .cpp files contain one function. This function is essentially a program that shows the user that the functions I wrote are indeed correctly working. These .cpp files are:

- AffineAttacks.cpp
- AffineCipher.cpp
- DES.cpp
- FrequencyAnalysis.cpp
- MultiAlphabetAttack.cpp
- MultiAlphabetCipher.cpp
- RSA.cpp

In addition to the functions needed to run each of these files, there are other number theory functions included in the CryptoMath.cpp that are specifically for this project, but they can be used for future work.

Running the code

If you are on windows (like I am), run your code in the following way:

1) Download Visual Studios
2) Create a project and add the CryptoMath .cpp and .h files to this project
3) Add any or all of the other .cpp files to this project
4) For the file you are going to execute, edit the file and change the name of the single function in that .cpp file to “main” (without quotes)
5) Build and run through Visual Studios and follow the prompts in the command line window. This is the program running and takes input from the keyboard.

If you are on Linux, run your code in the following way:

1) Ensure you have the g++ compiler installed on your machine
2) Add all files to a folder
3) For the file you are going to execute, edit the file and change the name of the single function in that .cpp file to “main” (without quotes)
4) Build by executing: g++ CryptoMath.cpp otherFile.cpp
5) This will create an executable named a.out in the working folder
6) Execute this program and enjoy the crypto

Code

The following functions are contained in the CryptoMath.h library:

- affineEncode - Given the alpha and beta keys along with the string of text, encrypts the text using affine cipher
- affineDecode - Given the alpha and beta keys along with the string of ciphertext, decrypts the text using inverse affine cipher
- affineAttackCiO - This attack uses only the ciphertext. Brute force is ran and all 312 possible decryptions are ran. There is no functionality for the program to recognize which of the possible outputs are legible in english language, so the user must look and deduce which row gives the correct result, and from that you can get the alpha and beta keys
- affineAttackKP - This attack uses some plaintext and its corresponding ciphertext. This is attack is done by creating a relationship between the input and the output. By only using two letters from the plaintext and its corresponding ciphertext, you can create two equations with two unknowns and solve the congruential equations simultaneously. This gives you alpha and beta keys.
- affineAttackChP - This attack assumes the user has in their possession the encryption machine, and they can run any plaintext they want through it and see its corresponding ciphertext. Since this is an affine mapping, the equation is of the form y = ax + b, where y is the ciphertext and x is the plaintext characters. The trick is to run *ab* through the machine and observe its output. Since *a* is the first letter of the alphabet, its index is 0. So *a* gets mapped to y = a * 0 + b which is equivalent to y = b. Now, *b* gets mapped to

y = a * 1 + b which is equivalent to y = a + b. Since we know b we can solve for a. Therefore, we can find the alpha and beta keys

- affineAttackChC - Same procedure as the chosen plaintext, since decryption is also an affine map.
- upper - Given a string, converts the string to uppercase
- gcdExtended - Given two integers, performs the extended Euclidean Algorithm on them, which, in addition to computing the gcd(a, b), also simultaneously computes, with little to no extra cost, the coefficients of Bezout's identity: ax + by = gcd(a, b). In normal euclidean division, only the remainders and kept, and the quotients are discarded. This function takes advantage of those quotients (along with the remainders) and computes x and y.
- gcd - Given a and b, recursively computes the greatest common divisor of a and b. This function uses the property that the gcd(a, b) is congruent to the gcd(b mod a, a)
- modInverse - Given a number *a* and a working modulus *m,* computes the inverse of *a* mod *m.* Since the gcdExtended solves equations of the form ax + by = gcd(a, b), let y = m. Then ax + bm = gcd(a, b) => ax = gcd(a, b) (mod m). We want the inverse of a, so we need ax = 1 and solve for x, so we need gcd(a, b) = 1. So we first check that gcd(a, b) is indeed 1 (which means they are relatively prime), then we compute gcdExtended(a, m). This return x and y, but we only care about the x for this particular function. This is our inverse mod m.
- isPrimitive - not functional as of yet

- phi - Given a number n, returns the number of numbers that are less than n and are relatively prime to n (gcd (a, n) == 1 for all *a* less than *n*)
- vigenereEncrypt - Given a plaintext string and key string, encrypts the plaintext using a multi-alphabet cipher. This is a block cipher that uses a vector as the key.
- vigenereDecrypt - Given ciphertext string and key, decrypts the ciphertext
- vigenereAttackKeyLength - Given a ciphertext string, attempts to deduce the length of the key used to encrypt it. This attack uses symmetry in the encrypted data.
- vigenereAttackKey - Given a ciphertext string and the length of the key, gives you the key used in the encryption. This works better for longer ciphertext strings. The frequency of each letter is computed and dotted with the frequencies of English letters, but we shift this vector everytime and see which dot product produces the greatest result. This tells us information about which letters were used.
- Freq - Given a text string, run a frequency analysis of each letter (English)
- shiftVectorRight - Given a vector of integers and number of shifts, shifts (more specifically, rotates) all elements that many times to the right. Any element that goes off the end goes back to the beginning.
- dotProd - Given two vectors of integers, computes the dot product
- modPow - Given a number a, exponent n, and modulus m, computes a^n mod m. This function avoids overflow by reducing the number everytime a multiply is done.
- RSAEncrypt - Given a number n, exponent e, and the message as an integer, encrypts the message by calling modPow(message, e, n).

- RSADecrypt - Given the secret prime p and q, and the working exponent e, decrypts the message. This is first done by computing the inverse of e mod phi(n), and calling modPow(message, d, n). This works since $(message^e)^d$ equals 1 since e * d == 1 mod phi(n).
- factor - Given n, the number to factor, and m, an integer indicating which factoring method should be used, attempts to factor n.
- fermatFactor - Attempts to factor n using Fermat's factorization method. This function uses that fact that an odd integer can be written as the difference of two squares. So it can be written in the form (a+b)(a-b). This function solves for a and b.
- rhoFactor - Uses Pollard's rho algorithm for factoring a number.
- pMinusOneFactor - Uses Pollard's p-1 algorithm (Not exactly sure on this)
- isPerfectSquare - Given a number, determines if it is a perfect square. This is done by first take the square root of the number, and then the floor of the square root of the number. If they equal, that means it is a perfect square.
- g - Used in conjunction with rhoFactor. It is essentially a polynomial mod n
- strToVec - Given a vector of integers, converts this to a string
- vecToStr - Given a string of integers, converts this to a vector of integers
- roundKey - Given the key string and the current round, computes the roundkey. This is computed by starting at the i^{th} bit in the key, where i is the current round. The next 8 bits are the roundKey.
- Expander - Given a string of 6 bits, expands it to a new number that has 8 bits. This is done by duplicating the middle 2 bits of the 6 bit number to create an 8 bit number.

- DEShelper - Given the message string to encrypt, the key string, and the number of rounds to perform, initializes variables. For each round, compute the round key and call DES with the left and right halves of the message, along with the round key. Returns the final encrypted message
- DES - Given the left and right 6 bits of the message, along with the current roundKey, calls our f function on the right 6 bits and the roundKey. The result of this is XORed with the left 6 bits and results in the new right 6 bits of the message. The new left 6 bits of the message is set to the old right 6 bits. The new left and right 6 bits are concatenated and returned.
- f - Defines the function used in DES. This is the one used in the simplified DES. The algorithm is as follows: This function takes in the right 6 bits at the beginning of the round and is expanded to 8 bits. This 8 bit result is then XORed with the round key and the 8 bit result is stored. We then split this 8 bit number into two parts, the left and right 4 bits. These 4 bits are ran through 2 S-boxes. An S-box is a table that maps a 4 bit input to a 3 bit output in the following way: The first bit indicates the row to use, and the remaining 3 bits correspond to the column. The output is the entry at s-box[row][col]. The 3 bit output from each s-box is concatenated into a 6 bit out and returned from this function.
- XOR - Given two binary strings, does bitwise addition mod 2
- isPrime - Schoolbook method for determining whether a number is prime or not. Complexity: O(n)

- Random_prime - Given b, generates a random prime between 2^{b+1} - 1 and 2^{b} - 1. This uses the rand() function and generates a random number in that range. This function then loops until the number it generates returns true from the isPrime function. At that point, we know it is prime and the number is returned.