SamCWill's Cryptography Suite

Preface

Hello! I'm SamCWill, and as a requirement for a cryptography course, I had to implement a variety of cryptography functions. I hope that you can make use of them! Many of the basic ideas that these functions are based on are available in "Introduction to Cryptography with Coding Theory" by Wade Trappe and Lawrence Washington, and many of the functions with more general usefulness, such as gcd(), modInverse(), etc, can be found thoroughly described, often with accompanying pseudocode, on Wikipedia.

This suite implements a couple of classical cryptosystems, namely the Affine and Vigenere ciphers, alongside attacks for those ciphers, a simple letter frequency counter, and a very basic implementation of a "one-time pad". It also includes a basic cryptomath library, which includes gcd, extendedgcd, modInverse, modPower, isPrime, randomPrime, and factor functions, as well as simple DES and RSA implementations.

The only dependency of this suite is the MPIR: Multiple Precision Integers and Rationals library, which I've used liberally to perform operation on numbers that cannot fit in 64 bits. Much of the apparent complexity of the code stems from jumping through hoops in order to use MPIR, but after reading through a bit of its documentation, one quickly realizes that it's nothing to fear, and its functions are named quite intuitively!

Much of the aforementioned cryptomath library, and the RSA implementation use MPIR over generic data types, and can therefore function with much, much larger numbers than the rest of the suite can. Bringing the rest of the suite up to this standard is certainly on my to do list.

The remainder of this paper will go into details on specific functions within the suite. In the case that this document fails to answer any of your questions, feel free to contact me at samuelcwill@gmail.com.

Classical Cryptosystems

The ciphers implemented within this section are terrible! If you value the security of your information, do not use these! They're only here for educational purposes, and in order to illustrate their shortcomings, I've provided simple attacks for each of them. The only possible exception to this is the one-time pad, which, as long as the keys are distributed secretly, is technically as difficult to crack as the random number generator powering it.

Affine Cipher

The affine cipher is incredibly rudimentary, and simply performs the following operations to encrypt and decrypt:

$$E(x) = (\alpha x + \beta)(mod\ m)$$
$$D(x) = \alpha^{-1}(x - \beta)(mod\ m)$$

m is the size of the alphabet, and will be 26 through this entire suite: the number of lowercase letters in the english alphabet. a is any number that is coprime to m, and b is any number. a^-1 is the modular multiplicative inverse of, which is found using a function that will be discussed later.

In order to use this cipher, simple call affine::encode(a, b, plaintext), and affine::decode(a, b, ciphertext).

There is also an option to attack this cipher with a known plaintext attack, in which you input a known plaintext and the ciphertext it generates, and it will return to you the key: a and b. This is called with affine::knownPlaintextAttack(&a, &b, ciphertext, plaintext).

Vigenere Cipher
Another simple and easy to crack cipher which simply cycles through each letter in a key, adding it to the corresponding letter in the message, until the entire message has been encrypted. The code looks something like this:

```
for( int i = 0; i < plaintext.size(); i++)
        ciphertext[i] = (((ciphertext[i] - ‘a’) + (key[i % key.size()] - ‘a’)) % m) + ‘a’;
```

The plus and minus ‘a’s are simply to convert ascii values into numbers in the range [0,25] and then back again. Decoding is nearly the same:

```
for( int i = 0; i < plaintext.size(); i++)
        plaintext[i] = (((ciphertext[i] - ‘a’) - (key[i % key.size()] - ‘a’) + m) % m) + ‘a’;
```

Now we’re just subtracting the key value away instead of adding it in. You may also notice that I’m adding an extra m in. This is simply to keep the value positive. In the case that this addition is unnecessary, the modulus operation will just undo it anyways.

The attack for this cipher is a good deal more complicated. It starts by dividing the message into columns by possible key lengths. The it averages the coincidence indices of each of those columns. These indices gauge how similar the letter distribution of each column is to that of the english language. Very similar distributions get higher indices. Different column divisions are tried out, and their indices recorded, for every viable key length, 1 through half the length of the ciphertext. Any key longer than this would be very rarely cracked by this approach.

The highest coincidence indices point to the correct key length, and once this is known, it is easy to use the frequency count of each column individually to ascertain the key letter used to modify that column.

One-Time Pad
This algorithm is quite similar to the Vigenere Cipher, in that it add letters together to encrypt and subtracts the same letters away to decrypt. The main, and very important, difference is that while the Vigenere Cipher gets its key value by looping through a finitely long key, the one-time pad generates its key values with a random number generator. If a strong random number is chosen, the period of this generation can be massive, and if this is case, any sort of patterns will be completely unidentifiable, making this implementation of the one-time pad

immune to the attack I used on the Vigenere Cipher up above. As long as the encrypt and decrypt functions are seeded with the same key, this cipher will work.

Basic Number Theory

This section contains a number of modular arithmetic functions, and more! These functions are used throughout the rest of this suite, but can also be quite useful by themselves. Many have been designed to work with numbers of variable length through MIPR.

GCD

This function finds the greatest common divisor of two integers. In other words, the largest integer that divides both of them evenly. If this number is 1, the input integers are coprime, meaning that they share no divisors greater than one. Even two non-prime numbers can be coprime. There are a number of ways to find the GCD of two numbers, and I've gone with a very simple method that has been set up to work with MIPR.

Extended GCD

This function finds the greatest common denominator and more! It also returns the values you must multiply your inputs by to produce the gcd.

Modular Inverse

Finds the modular inverse of an integer a mod an integer n using the extended Euclidean algorithm. This can only be found if a and n are coprime!

Modular Power

This function can find massive exponential values mod n with relatively little space required simply by performing multiplications one at a time and taking the mod after each multiplication. This has been set up to run with MIPR, so massive numbers are allowed, but the runtime is O(e), so very large exponents will cause this algorithm to run quite slowly.

Is Prime

This function generates a random number between the number being tested for primality, n, and 1, and then finds:

$$rand^{n-1} (mod\ n)$$

If this returns anything except for 1, it guarantees that n is not prime. If it returns one, it is quite likely that n is prime, especially for large values of n.

Random Prime

Generates random numbers between 2^{b+1} and 2^b and then checks primality. This repeats until a prime is found, which is then returned.

Factor

I've implemented a couple of methods for factoring large primes which can theoretically be used to crack RSA ciphers with non-massive n values. The first version works with Fermat's Method, and the second with Pollard p-1.

For Fermat's method, a counter starts at 1 and increases. It's squared and added to n at each step, and this sum is run through a perfect square test. If the sum is itself a perfect square, its prime factors can be found: (sqrt(sum) + counter) and (sqrt(sum) - counter).

The Pollard p-1 method take a different approach. It computes $a^{B!}(mod\ n)$ with a being 2, B being some chosen bound, and n being the value to be factored. The gcd of this number - 1 and n is taken, and if if it lies somewhere between 1 and n, it is a non trivial factor.

DES

This simple DES: Data Encryption Standard implementation only performs four rounds on a single 12 bit block, and the algorithm is now outdated, having been replaced by the AES: Advanced Encryption Standard. This cipher works by splitting the 12 bit data block into 2 6 bit blocks, and running them through a number of rounds.

In each round, the left bits are replaced by the right, and the right bits are run through a function with many steps. First, the 6 bit chunk is expanded into 8 bits. The 5th and 6th bits are moved into the 7th and 8th slots, and the 3rd bit is put into the 4th and 6th slots while the 4th bit is put into the 3rd and 5th. The first and second bits remain unchanged. This expanded chunk, now 8 bits in length, is then bitwise exclusive or'd with an 8 bit portion of the key that changes with each cycle.

This first 4 bits of this new 8 bit chunk are used to access a predefined S1 box and the last 4 are used to access an S2 box. Each returns 3 bits, which are combined to give you the new right bits.

In order to decrypt, you go through the same key segments in reverse, and move the left bits into the right and xor the right bits with the result of running the function on the left bits, which you store in the left bits.

RSA

This algorithm is quite simple! Most of the code I've written in the encrypt and decrypt functions simply converts text to numbers and vice versa, and makes everything play nice with MIPR. The actual algorithms for encryption and decryption are simply:

$$c = m^e(mod\ n)$$
$$m = c^d(mod\ n)$$

N is the product of two primes p and q, which are used to pick e. Namely, (p-1)(q-1) and e must be coprime. d is chosen to be the multiplicative inverse of e (mod (p-1)(q-1)). p, q, and d are kept secret, but n and e are openly distributed.