Cryptography Portfolio

Dana Thomson

December 2021

Description:
This document is designed to showcase the types of cryptographic systems that I have learned. The document covers the types of cryptosystems implemented and the math behind them. The algorithms of each system is described and has images detailing the process. The math section details different mathematical concepts and methods which are key to many developments in cryptography. This document will serve as a user guide and walkthrough.

Algorithms and Libraries used:
I used 2 python libraries random and numpy. Random is used for the random.randint function. This function is used for pseudo random number generation. The numpy library is used for dot and matmul functions.

Function and Program Structure:
My system has 5 files, main.py, classiccrypto.py, des.py, rsa.py and cryptomath.py. Main contains the user menu and input. It is a starting point for the system. Each of the other files is included in main so only main needs to be run to see the functionality of all files. The algorithms used throughout the application are described in more detail below.

1 Crypto Systems

Affine: When the user wants to run an affine cipher they have the option to encrypt, decrypt, a brute force attack, or a given plain text attack. The basis of an affine cipher is to use a linear one-to-one function to encrypt the plain text. To decrypt the modular inverse must be found of the alpha value used to encrypt. The modular inverse function is written in the cryptomath file.

$$x \mapsto \alpha x + \beta \pmod{26}.$$

The encryption and decryption functions ask the user for the alpha and beta used for encryption of the text. The encryption function will then take the plain

text and alter it using the one-to-one function. The decryption function takes the modular inverse of alpha and then uses the one-to-one function to alter the cipher text back to plain text. I have coded two types of attacks for text encrypted by an affine cipher. The first is a brute force attack, this attack tries all 312 possible alpha/beta combinations and prints a section of decrypted text to a file. The user then can look through the file to find the readable decrypted text and associated alpha/beta values. The second attack type is know plain text, this requires the user to enter two characters of cipher text with their associated two characters of plain text. Then the code uses these two values to find the alpha and beta values using basic algebra.

Vigenere: The Vigenere cipher is a shift cipher based on a key. One of the strengths of this cipher is that it doesn't require a one-to-one function. Since the text is shifted based on the key there isn't a set unique cipher text value for each unique plain text value. Another strength of the Vigenere cipher is that the key length is unknown to an attacker, making it harder to decrypt. The encryption and decryption methods ask the user to supply the text and the key. Then to encrypt the key value at a the current index is added to the text value mod 26. To decrypt the key value is subtracted instead of added.

(plaintext)	*h*	*e*	*r*	*e*	*i*	*s*	*h*	*o*	*w*	*i*	*t*	*w*	*o*	*r*	*k*	*s*
(key)	21	4	2	19	14	17	21	4	2	19	14	17	21	4	2	19
(ciphertext)	*C*	*I*	*T*	*X*	*W*	*J*	*C*	*S*	*Y*	*B*	*H*	*N*	*J*	*V*	*M*	*L*

The attack on a Vigenere cipher is dependent on finding the key length. To do this the encrypted text is compared against a shifted version of the encrypted text looking for coincidences. A coincidence occurs when the characters at index i are the same in the original and shifted text. The key length is most likely the number of shifts that gives the highest number of coincidences. In our algorithm the key length is not updated if the new max value is a multiple of the old, for instance 5 and 15. This is due to the cyclic nature of applying the key and the smallest multiple is most likely the true key. Once the key length is found the key value is found through frequency analysis. The cipher text is split into index values based on the key length and a frequency analysis using dot products is done for each letter of the alphabet, the known frequency table is shifted every time to have a new leading letter. By doing this the letter that produced the highest dot product value is most likely the letter of the key at that index. This is repeated for the length of the key to find the complete key value.

Hill Cipher: The hill cipher is a fairly simple cipher which uses an invertable matrix to encrypt and decrypt text. To encrypt the text is split into vectors of the same dimension as the matrix, then the vector is multiplied by the matrix mod 26. The result of this multiplication is the cipher text. To decrypt the same process is done with the inverse of the encryption matrix.

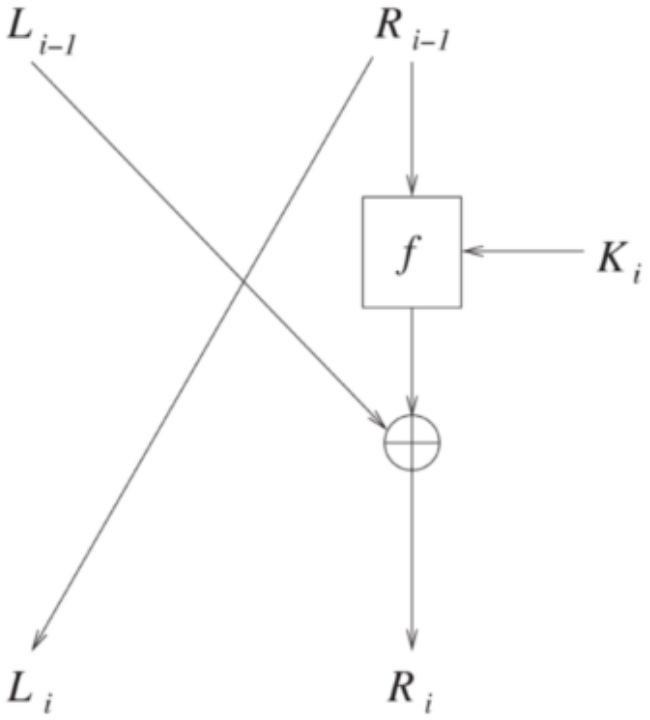
$$(0,\ 1,\ 2)\begin{pmatrix}1 & 2 & 3\\ 4 & 5 & 6\\ 11 & 9 & 8\end{pmatrix} \equiv (0,\ 23,\ 22) \pmod{26}.$$

One Time Pad: One time pad is a type of block cipher like the hill cipher. The one time pad though uses a key value to add to the message but the key values are never repeated. This makes it so that even if an attacker where to find the key from a message, that key couldn't decrypt any other messages. My solution for one time pad is implemented using random numbers generated from a user given seed. This allows us to be able to encrypt and decrypt if we know the seed value. The encryption works by adding the random value to the plain text value and taking the result mod 26. The decryption works the same way except subtracting the random value instead of adding.

Frequency Calculation: This is a utility function that calculates the frequency of each letter in a given text. The function counts the number of times each letter appears and divides it by the total number of letters in the text to provide a decimal value representing the percentage of the letters. A sample of the table is shown below.

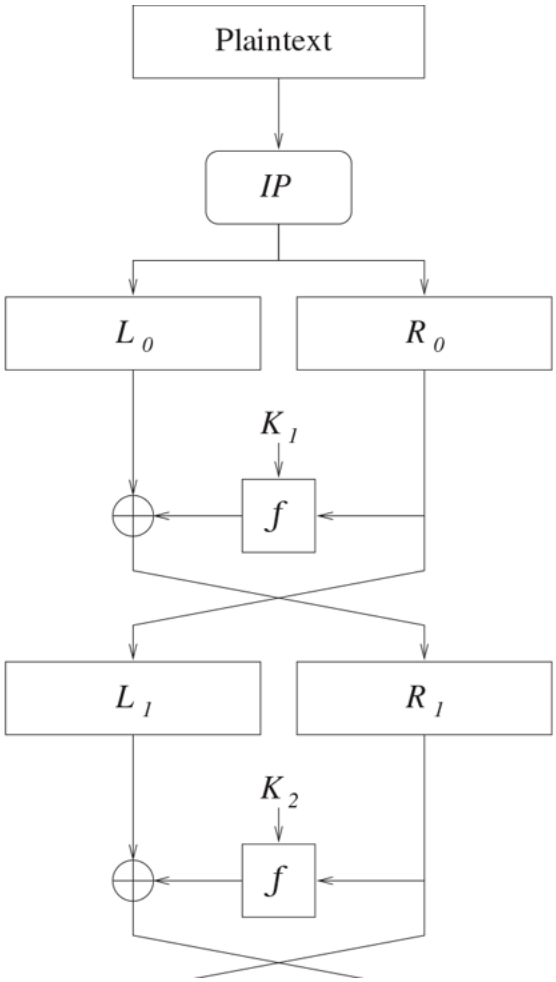
Letter	Frequency	Letter	Frequency
e	12.7020%	m	2.4060%
t	9.0560%	w	2.3600%
a	8.1670%	f	2.2280%
o	7.5070%	g	2.0150%
i	6.9660%	y	1.9740%
n	6.7490%	p	1.9290%
s	6.3270%	b	1.4920%
h	6.0940%	v	0.9780%
r	5.9870%	k	0.7720%
d	4.2530%	j	0.1530%
l	4.0250%	x	0.1500%
c	2.7820%	q	0.0950%
u	2.7580%	z	0.0740%

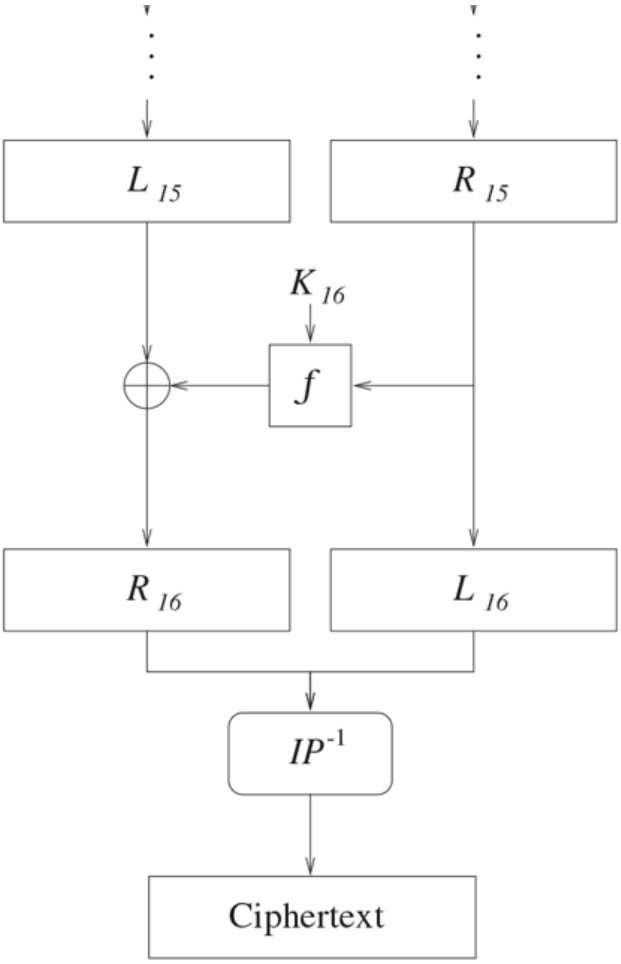
Baby DES: The baby DES is simplified to take in a 12 bit message, do four rounds of the algorithm and return the encrypted message. The algorithm is pictured below. The basis of the algo is to separate the message into left and right 6 bits. Then the right six bits are expanded to 8 bits and xored with a user given key. This value is then xored with the left six bits to create the next set of right six bits. The next set of left six bits are the previous right six bits. The key is 9 bits but only 8 are used in each round. After each round the key is rotated to the left one bit to create a new key. The final L values and R values are combined to form the encrypted 12 bit message. Decryption works in a similar pattern though the xors happen is slightly different orders to undo the pattern.



Three Round Cryptanlysis: The three round cryptanalysis is an attack on the DES system to determine the key. The user needs to know 4 initial and 4 end values for a simplified des machine, L_1R_1's and L_4R_4's. This attack focuses on weeknesses in xor, specifically that if two values are the same the xor is 0. The analysis uses this by having corresponding R_1 and R_1^* be the same values. By doing so we can learn what the s boxes xored inputs and outputs are. Once these are found an exhaustive search of all input possibilities will find the pairs which satisfy the input and output xors. If this process is repeated for 2 more sets of values then there should only be one overlap between the pairs of possibilities for right and left side of the key. By finding the left-right pair that is repeated in both possibilities 8 bits of the key are then known. Then the user can try filling in the missing bit with and 1 and a 0 on an input message of all zeros and see which version gives the actual encrypted message. By doing this the user now has the key to a simplified DES.

Full DES: The full version of DES uses a 64 bit message, in our case this is written as a 16 bit hex message. The key has 56 bits, but is given as 64 bits. The parity bits are not used which gets the key down to 56 bits. The system outputs a 64 bit cipher text in the form of a 16 bit cipher hex text. The flow of the algorithm is shown below.


Plaintext
IP
L 0
R 0
K 1
f
L 1
R 1
K 2
f



As shown by the graphic the full DES works in a similar way to the simplified DES. With the next right side being determined by the xor of the current left and the output of the f function. The key difference in the algorithm is in the initial and inverse permutations, key generation, and the f function itself.
The initial and inverse permutations are believed to make the algorithm load more efficiently into the chips that were available in the 1970s. The f function starts by expanding R to be 48 bits. The expanded Ri is then xored with Ki. This computed 48 bits is divided into 8 six bit chunks. Each chunk is fed into a corresponding S-Box and the outputs of these are combined to get the C value, which is 32 bits. Then the c value is sent through a permutation function. The result of this C permutation is the 32 bit string output for f(Ri, Ki).
This algorithm is run for 16 rounds each round uses a different key. The shift value is determined by the key shift table. After going through 16 rounds the text is passed in RL into the inverse permutation to get the cipher text.

One of the nicest things about DES is it's a Feistel cipher. This means the algorithm is symmetric. So to decrypt we simply need to apply the keys in reverse order. We do not need to make any other changes to the algorithm to decrypt.

RSA: RSA is a fairly simple public key crypto system. It is the only public key system implemented in this portfolio. RSA works by having each user, say Bob, choose two secret large prime numbers. These primes, p and q, are used to calculate $n = pq$. Then the Bob chooses an integer e such that gcd of $(p-1)(q-1)$ is 1. Then e is used to calculate d which is the modular inverse of e. Bob then makes their n and e values public and keep values p, q, and d private. If I want to send a message to this user, then I would take my message and calculate m^e (mod n). This is the cipher text I send to Bob. To decrypt this Bob will compute c^d (mod n).
Since there is only one user of my application the implementation is slightly different. When using the rsa implementation you can choose to generate public and private key data or to input your own. If you choose to generate then the system will run for little bit till it generates two random primes, p and q, in range $2^{20}+1, 2^{21}-1$. Then it will find a value between 5000 and p*q for which the gcd(val, (p-1)*(q-1)) = 1. Then the mod inverse of e is calculated. If there is no modular inverse then the systems keeps looking for a different e value. Once these values are found they are printed to the screen and the user has the choice to encrypt or decrypt using these values. If the user wants to encrypt/decrypt a message for which they already know their private key and public key values they will be prompted to input them once they decline for the system to generate data. Once this initial determination of public and private key data is found the algorithm runs like described in the first part of this section.

2 Math

GCD: This function takes two integers as input. First it checks that the first int, a, is larger than the second int, b. If it isn't then a and b are swapped. Then while b does not equal zero a becomes b and b becomes a mod b. Finally be return a which is the greatest common divisor.

Extended GCD: This function takes two integers a and b as input. First it checks that the first int, a, is larger than the second int, b. If it isn't then a and b are swapped. The egcd finds the gcd and the linear combination of a and b which will generate the gcd. To do this while b is not zero a becomes b, b becomes a mod b, linearX is calculated to be x0 minus x1 times the quotient of a and b, and linearY is calculate y0 minus y1 times the quotient. Then x0, y0 are set to x1, y1 and x1, y1 are set to linearX, linearY. Then a, x0, and y0 are returned. These provide the gcd and the linear combination values for a and b.

Mod Inverse: This function calls the extended_gcd function if the gcd is one

then there is no possible inverse. If it is not one then the inverse is returned as x mod n. Where n is the mod number.

Primitive Roots: This function takes a user given prime number and finds the primitive roots of it. A primitive root is a number whose power generates every non-zero class mod p.

Miller-Rabin: The Miller-Rabin algorithm is used to determine if a number is prime. The algorithm works by looking see if the value x is calculated to be 1 or -1. X is calculated by doing a power mod n. Each iteration an random value between 1 and n-4 is is raised to the power of d mod n. Where d is the fist value of integer division on n where the result is not an even number. These calculations are ran multiple times to double check that the algorithm didn't return a false positive. The iteration in my implementation is set to ten.

Random Prime: To generate a random prime I first aquire the range from the user. The user enters an integer b and the range of numbers looked at is $2^b+1, 2^{b+1}-1$. Then all numbers in this range are ran through our Miller-Rabin primality checker. If they are prime they are added to a list. Then using the random.choice function a random value from this list is selected, giving the user a random prime.

Factoring Numbers: In my application there are four seperate factoring algorithms Fermat's, Pollard Rho, Pollard p-1, and Shank's square.For the rest of this section let N denote the number we are trying to factor.
Fermat's Factorization uses perfects squares to find a factor. The algorithm looks for value x, y such that $x^2 - y^2 = N$. The algorithm start by setting up variable A to be the ceiling of the square root of N and variable B being A squared minus N. Then we set through increasing A by one until the B calculation is a perfect square. When B is a perfect square we know that a factor of N is $A - \sqrt{B}$.
Pollard Rho works by looking for a gcd not equal to 1 between a calculated value D and N. The value D is calculated using a function g. This function uses modular powers to give a new value. When the loop exits if the calculated D value is N then the algorithm failed to find a factor. If it's not N the the factor is D.
Pollard p-1 is similar to Pollard rho in that it is using gcd to determine if a factor has been found. Starting at 2 and going to a user input upper bound, B is updated to be the B^i mod N where i is the loop value. Then the gcd is calculated of B-1 and N. If the gcd is not 1 or N then a factor was found and the algorithm exits.
Shank's Square Factorization: Shank's algorithm was devised by Daniel Shanks as an improvement to Fermat's method. Instead of looking for value x, y such that $x^2 - y^2 = N$, Shank's algorithm looks for x, y such that $x^2 \cong y^2$ (mod N). We look for these values because this implies $x^2 - y^2 = (x-y)(x+y)$ and there is a good chance that the prime divisors of N are distributed between these two

factors. Shank's algorithm is the most complicated of the factoring algorithm implemented as it uses a forward cycle and a reverse cycle to find two rounds which have the same calculated P value. Then the gcd of this P value and N will give a factor of N.

How to compile:
To run this program simply run the main.py file. Be sure classiccrypto.py, cryptomath.py, des.py, and rsa.py are in the same directory as main.py. I have been running the program through the command line using the command 'python3 main.py'.

Testing and Verification:
To test this program I have ran each encryption and decryption algorithm on small then large input texts to verify that they are working correctly. For each attack algorithm I ran them multiple times to determine that they would find the correct answers in different situations. For the crypto math functions I tested agaisnt known values provided in the book and online.