South Dakota School of Mines and Technology

Cryptography, Fall 2021

CSC 512 - M01

Portfolio

Sherwyn Braganza

Classical Cryptosystems

- Shift Ciphers

  One of the simplest Cryptosystems out there, **Shift Ciphers** are formed by taking the numerical value of each character in the plaintext and then adding a constant value (**key**) to it, mod n (where n is the size of the character set). The orders are then translated back to their character values.

  What does order mean?
  It is the position/rank of the character w.r.t to the whole set - 1. In the set of the English Alphabet, **A** will have a rank of 0. Similarly **Z** will have a rank of 25.

  As most things are best explained by an example, consider the plaintext -

  “I LOVE CRYPTOSYSTEMS”

I	L	O	V	E	C	R	Y	P	T	O	S	Y	S	T	E	M	S
8	11	14	21	5	2	17	24	15	19	14	18	24	18	19	4	12	18

  The order of each of its characters is as follows.

  Adding a **key = 5**, and reverting numbers to characters, we get the following encryption.

I	L	O	V	E	C	R	Y	P	T	O	S	Y	S	T	E	M	S
8	11	14	21	5	2	17	24	15	19	14	18	24	18	19	4	12	18
13	16	19	0	10	7	22	3	20	24	18	23	3	23	24	9	17	23
N	Q	T	A	K	H	W	D	U	Y	S	X	D	X	Y	J	R	X

- # Affine Ciphers

The Affine Cipher is another simple Cryptosystem. It basically stems from regular **Shift Ciphers** and is governed by the following equation:

$$c \equiv \alpha * p + \beta \pmod{N}$$

Where

- **p** is the numeric value of the plaintext alphabet.
- **β** can be any arbitrary number.
- **α** is an any number who's **gcd(α, N) =1**
- **c** is the outputted cipher text letter
- **N** for the set of the English alphabet is **26.**

This system can be extended beyond the English alphabet system and can include other symbols too. **N** correspondingly needs to change to encompass the new range.

Few key things to keep in mind about this encryption system is

- It encrypts one character at a time; thereby the complexity of this encryption and decryption function is linear.
- Spaces are not encrypted but simply skipped over.
- It is susceptible to frequency attacks as every alphabet in the plaintext has only one ciphertext value/encryption.
- The key for this cryptosystem is 2 character combination of **(α, β).**

Decryptions of Affine Ciphers work in a similar way. Except the new function, the decrypting function is given by:

$$p \equiv \alpha^{-1} * (c - \beta) \quad \text{mod } N$$

Where α^{-1} is the the modular inverse of **α**.

Attacks on Affine Ciphers

As mentioned above, attacks on Affine Ciphers can be carried out using a simple frequency analysis. This works because each character has a unique cipher text encoding.

This link contains the code for a simple program that simulates the encryption and decryption of an affine cipher as well the case of an attack on it :

Affine_Cipher_Encrypt_Decrypt

- # Vigenère Ciphers

Vigenère Ciphers work in a similar manner as Shift Ciphers.

1) You first start with a **key (K)** that is a simple plaintext word (ex. sherwyn).
2) Write out the plaintext in a single line.
3) Directly below the plaintext, corresponding to each letter of the plaintext, write out the key in a cyclical fashion (as shown below)

```
T h e   q u i c k   b r o w n   f o x   j u m p e d   o v e r
s h e   r w y n s   h e r w y   n s h   e r w y n s   h e r w
```

4) The cipher text value/character is the sum of the plaintext character and the

character of the key right below it, mod 26.

In other words, each individual character encoding is treated as a Shift Cipher with the **β** value equal to character right below it.

The Cipher Text version of the above shown example is

```
l o i h q g p c i v f s l s g e n l i n r v v z v n
```

Decryption of Vigenère Ciphers works in a similar way as encryption. You lay down the a cyclical version of the key against each character of the cipher ciphertext and then take the sum of it mod 26 (since in this case we are working in the set of the Common English Alphabet).

Attacks on Vigenère Ciphers

Attacks on Vigenère Ciphers involve broad steps:

1) Finding the key length.

2) Finding the key.

Both these steps involve employing the same method - comparing the product of the set of shifted letter frequencies with the set of letter frequencies in the common English Alphabet.

This link contains a program that simulates a Vigenère Cipher as well as an attack on it to guess the key - Vigenère Cipher

- # ADFGX Ciphers

ADFGX ciphers belong to the class of Matrix Ciphers. You start out with a predetermined matrix encoding each letter in the English Alphabet to its **row-column combination**. Consider this predetermined **ADFGX matrix.**

	A	D	F	G	X
A	p	g	c	e	n
D	b	q	o	z	r
F	s	l	a	f	t
G	m	d	v	i	w
X	k	u	y	x	h

From this we know that **k** encrypts to the **row-column** combination of ***XA.***

***** j and I are treated as the same character and encrypt to the same character in the matrix.

Encryption

The Encryption process contains 2 steps and involves the predetermined ADFGX matrix and a **keyword**.

The encryption process is best described with an example.

Consider encrypting the plaintext: ***Kaiser Wilhelm***

With the keyword: ***RHEIN***

A simple ADFGX row-column transformation of the plaintext yields the following ***transformation text***.

XA FF GG FA AG DX GX GG FD XX AG FD GA.

This is then arranged in a matrix headed by the keyword, as shown below. The transformation text is arranged from left to right, wrapping around when it reaches the last character of the keyword.

R	H	E	I	N
X	A	F	F	G
G	F	A	A	G
D	X	G	X	G
G	F	D	X	X
A	G	F	D	G
A				

The columns are then reshuffled so that the keyword headers are in alphabetical order (shown below).

E	H	I	N	R
F	A	F	G	X
A	F	A	G	G
G	X	X	G	D
D	F	X	X	G
F	G	D	G	A
				A

Each individual column is then outputted (ignoring the keyword header), from top to bottom, to give you the encrypted ciphertext:

FAGDFAFXFGFAXXDGGGXGXGDGAA.

Few interesting things to note about this is that ciphertext is built as a permutation of just 5 letters - **A, D, F, G and X.**

A, D, F, G, X were specifically chosen because their Morse Code equivalents are easily distinguishable from each other to prevent confusing one with the other. Which leads us to the inference that this was probably used in WW1, when Morse code was one of the most utilized means of communication.

Decryption

Decryption is basically done by following the encryption algorithm from bottom to top.

1) Our initial aim is to calculate the number of rows in each “header column”. TO achieve this, we do an integer division of the size of the ciphertext by 5. This gives us the mean # of rows for each column.

Performing this step on our above example, we get the mean # of rows to be 5. This leaves us with the following number of columns.

E = 5

H = 5

I = 5

N = 5

R = 5

2) Now we take the modulo of the size of the ciphertext with 5

sizeof(ciphertext) % 5 = 1

The value we get tells us how many header columns, with respect to the original keyword ordering needs to get padded with an extra row value.

In our example here, the **'R'** header column, will get padded with an extra row resulting in the final row count for each column being show as below.

E = 5

H = 5

I = 5

N = 5

R = 6

Considering all the other cases in which **sizeof(ciphertext) % 5** returns 2, 3, 4; the modified rows numbers for each column is shown in following table

sizeof(ciphertext)%5 =	2	3	4
E	5	5	5
H	6	6	6
I	5	6	6
N	5	5	6
R	6	6	6

3) The next step is to grab characters corresponding to the column size, in the rearranged alphabetical order of the keyword (the polar opposite to the last step of the encryption method).
This leaves you with the matrix looking like this

E	*H*	*I*	*N*	*R*
F	*A*	*F*	*G*	*X*
A	*F*	*A*	*G*	*G*
G	*X*	*X*	*G*	*D*
D	*F*	*X*	*X*	*G*
F	*G*	*D*	*G*	*A*
				A

4) The next step is to reshuffle the columns to resemble the original keyword.

R	H	E	I	N
X	A	F	F	G
G	F	A	A	G
D	X	G	X	G
G	F	D	X	X
A	G	F	D	G
A				

5) Print from this matrix, reading from left to right (starting from row 1), with wrap arounds. Performing this step on the example we are discussing, leaves us with this ***transformation text.***

XA FF GG FA AG DX GX GG FD XX AG FD GA.

6) Our job now is to look at our predetermined ADFGX Matrix and write the English alphabet character corresponding to each 2 letter combination.

XA FF GG FA AG DX GX GG FD XX AG FD GA.

The source code for a simple ADFGX encryption decryption simulator in python can be found at this link: **ADFGX Simulator**.
It does used a preset ADFGX matrix which is the same one discussed in the example. It has an Encryption and Decryption Mode: both modes require you to provide the key.

- # Block Ciphers

Block Ciphers belong to the class on Matrix Ciphers, i.e. a matrix of characters/ values is used to encrypt a given plaintext. In Block Ciphers, the key is a $m_x m$ matrix/block, that is filled with integers mod 26 (if using the English Alphabet as you working set). Integers should be chosen in such a fashion that the resulting **key matrix** is not singular.

An example key would be:

$$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 11 & 9 & 8 \end{pmatrix}$$

Encryption

Matrix multiplication mod 26 is performed using the **key matrix**, on a plaintext vector of length m (vector representing the number encoding/order of the English character/letter). The resulting/matrix vector is the numerical encoding/ ordering of the ciphertext.

Decryption

Decryption of block ciphers follow the similar rules. The ciphertext is multiplied by the **inverse of the key matrix** to produce the plaintext.

Few things to note about Block Ciphers are

- It encrypts ‘m’ characters at a time.
- The key has to always be a square matrix.
- Since Block Ciphers encrypt multiple characters at a time, a change in one character in the plaintext would result in a multi-character change in the ciphertext.

Due to this design benefit, Block Ciphers are kind of resilient to simple frequency analysis ciphertext only attacks. It however does succumb to known ciphertext attacks.

The python code for a simple Block Cipher Simulator (Encrypt/Decrypt) can be found at this link: **Block Cipher Emulator**

• Attacks on Cryptosystems

There are 4 broad categories for classifying attacks on Cryptosystems. The end goal of all attacks is to find the key.

1) **Ciphertext Only Attacks** - The challenger has a copy of the ciphertext only.
2) **Known Plaintext Attacks** - The challenger has copy of the plaintext and the ciphertext it encrypts to.
3) **Chosen Plaintext Attacks** - The challenger has temporary access to the encryption side of the cryptosystem and uses this advantage to deduce the key.
4) **Chose Ciphertext Attacks** - The challenger has temporary access to the decryption side of the crypto system and uses this advantage to deduce the key.

- # Frequency Analysis

One of the most common methods that fall under the category of **ciphertext only attacks** is Frequency Analysis.

As is pretty evident from the name itself, this method involves doing a sweep of the ciphertext and recording the occurrences of either the characters by them self or two letter combinations. Higher order combinations (3 letter and 4 letter) are pretty inconsistent in terms of giving valid uniform results.

The next step is to compare them with the well known letter frequencies in common literature. The table below shows frequency ratios of the single letters in an common piece of English Literature.

a	b	c	d	e	f	g	h	i	j
.082	.015	.028	.043	.127	.022	.020	.061	.070	.002
k	l	m	n	o	p	q	r	s	t
.008	.040	.024	.067	.075	.019	.001	.060	.063	.091
u	v	w	x	y	z				
.028	.010	.023	.001	.020	.001				

Cryptosystems that have a direct correspondence between ciphertext character and plaintext character, i.e. systems in which a plaintext character is encrypted to a unique ciphertext character and vice-versa, are the easiest to break (find the key) using this method. All you got to do in these cases is just compare calculated character frequencies in the ciphertext and compare them to the character frequencies in the above mentioned table. This leaves us with the knowledge of correspondence between each ciphertext and plaintext character, which can easily be used to solve for the key.

Shift Ciphers, Affine Ciphers and Vigenère Ciphers are prime targets for these type of attacks.

- # Linear Feedback Shift Registers

  Sequences that are generated as a product of some arbitrary functions on its previous state/sequence are called Feedback Sequences. Linear Feedback Shifting is a method that generates sequences from its previous/initial sequences using a linear function that involves shifting. The Hardware used to perform these functions are called Linear Feedback Shift Registers. These Sequences are self generating Markov like processes and can generate fairly long sequences before repeating themselves.

  Now for the heavy, Mathematics part:

  - A Linear Feedback Shifting Function can be represented as an irreducible/ reducible function in a Galois Field, GF($2^{\text{size of initial sequence in bits}}$).
  - Whats the difference between using irreducible and reducible polynomials?
    Short Answer: Periodicity.
    Irreducible polynomials are like prime numbers, they can act as number generators for all the numbers belonging to the set Z:{1, size-1} where size is the size of the Galois Field.
    Reducible Polynomials cannot do that as they are subject to periodicity amongst a small set.

  For example, consider the GF(2^8) or an 8 bit size sequence. Your starting sequence can be anything. However if you chose your function to be : $x^3 + x^2 + x + 1$ (3rd bit XORed with the 0th bit) you encounter periodicity way early and are not able to generate all the possible numbers in the set Z:{1 - 255}

Attacks on Linear Feedback Shift Registers

Attacks on Linear Feedback Shift Registers involves finding the Irreducible Polynomial the Galois Field in which it is working.

This sample Program simulates and Attack on an LFSR using Matrix Math to find the Polynomial - **LFSR_Decrypt.**

Linear Feedback Shift Registers are highly susceptible to pretty much all the 4 types of attacks. The strength in LFSRs rely on the initial sequence being large, thereby working in a Larger Galois Field with a Larger number of Irreducible Polynomials.

Number System

- ## Greatest Common Divisor

  Pretty evident from the name itself, the Greatest Common Divisor is the largest number that divides the numbers you are considering, without leaving any remainder.

  The algorithm to find said divisor involves the following steps:
  - Splitting the numbers into their prime factors
  - Combining the common prime factors (and their powers if they are common too)

  This link will take you to the implementation of this algorithm in Python.

- ## Extended GCD

  This Algorithm implements the Euclidean GCD algorithm (click link for more information). Apart from returning the Greatest Common Divisor, it also returns the modular inverse of the smaller of the pair (mod the larger of the pair)

  An implementation of this algorithm in Python can be found **here.**

- ## Diophantine Equations

  Equations of the form $ax + by = c$ in which x, y are unknowns, and the solutions to these equations are integer values, are known as Diophantine equations. Solutions to this equation can be found using modular arithmetic even though you have two unknowns and and just the one equation.

- ## Modular Inverse

  The modular inverse of a number 'a' mod(n) is defined as a number when multiplied with a, satisfies the following equation.

  $$\mathbf{a * x \equiv 1 \bmod(n)}$$

  In this case, **x** is defined as the modular inverse of **a**.

  One of the most efficient algorithms for calculating this is extended from Euler's GCD Algorithm. Working your way upwards from the final step of this algorithm, you end up with the equation

  $$\mathbf{- n * y + a * x = 1}$$

  or

  $$\mathbf{n * y + a * (-x) = 1}$$

  The case of (-x) involves an extra step in which you calculate the positive analog by adding n to (-x).

  An implementation of this algorithm can be found **here.**

- ## Matrix Modular Inverse

A matrix is invertible only if the modular inverse of its determinant mod(n) exists,.

For a 2x2 matrix, the Modular Inverse is calculated as follows:

$$\begin{pmatrix} a & b \\ c & d \end{pmatrix}^{-1} = \frac{1}{ad - bc} \begin{pmatrix} d & -b \\ -c & a \end{pmatrix}$$

The next step is calculating replacing 1 /(ad-bc) - The Determinant - by the modular inverse of itself.

This same method can me be extended to N x N matrices and can be simplified as normally inverting a matrix and instead of dividing it by the determinant, you multiply it by the modular inverse.

The **mod_inv.py** file in the library csc514-crypto/Number_Theory contains code for a function that provides the modular inverse for a given matrix. It does this by first calculating the regular inverse and the determinant of the matrix.
It goes through each of the elements of the matrix, dividing the elements by the modular inverse of the determinant.
The result is a matrix that is the modular inverse of the Matrix provided.

- ## Primitive Roots mod(n)

A primitive root mod(n) is defined as a number who's powers yield all non zero classes mod(n). More intuitively, 'a' is defined as a primitive root m if a when raised to powers mod(n) acts a number generator for all numbers less than n.

$a^k \equiv Z \bmod(n)$ **where Z is the set of all integers [1, n)**

Checking whether a number is a little complicated but possible. It involves factoring (n-1) into its prime factors and checking if a raised (n-1)/each prime factor is congruent to 1 mod (n)

$a^P \equiv 1 \bmod(n)$ **for all P that belongs to set of Prime factors of (n-1)**

Below is a table of all the primitive roots from mod(1) to mod(10)

1	0
2	1
3	2
4	3
5	2, 3
6	5
7	3, 5
8	
9	2, 5
10	3, 7

DES and AES

- ## Feistel Ciphers

  Feistel Ciphers are symmetrical block ciphers.

  Block Ciphers - Ciphers that work on blocks of ciphertext/plaintext. These block sizes are predetermined (usually blocks of bits)
  Symmetrical - The Algorithm is the same for encryption and decryption.

  Feistel Ciphers are tiered or include multiple rounds. Each round performs the same basic Feistel function; for example XORing blocks together or Permuting bits, etc.
  Unlike Hashing functions, Feistel functions are not one-way functions or rather, the inverse of the function is existent and most usually involves sending in the cipher text blocks through the same mechanism that was used to produce it. The figure above shows a simple Feistel System.

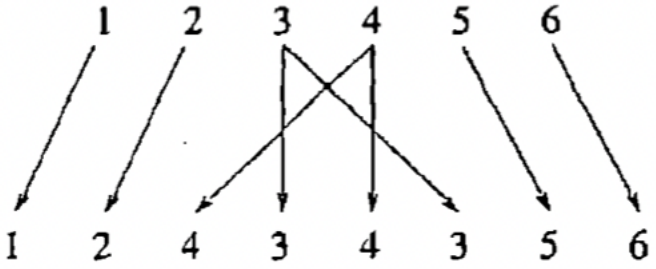
  At the center of this system is the function F that works on the R block and the K block (the key block). This function can be anything who's inverse exists (from simple XORs to really really complex permutations).

- ## DES - Data Encryption Standard

  DES is a ciphering algorithm that was the standard for encrypting data till it was replaced in 1997. At its core, it employed a Feistel System as its main Encryption Algorithm.

DES works on 12 bit blocks. It splits this block into two 6 bit chunks - L_n and R_n where n denotes the round. Before we dive deep into the algorithm, lets take a loot at a few functions and substitutions used in the encryption decryption process.

The Expander Function - This function expands a 6 bit number to a 8 bit number. The figure below shows how the expander function works.

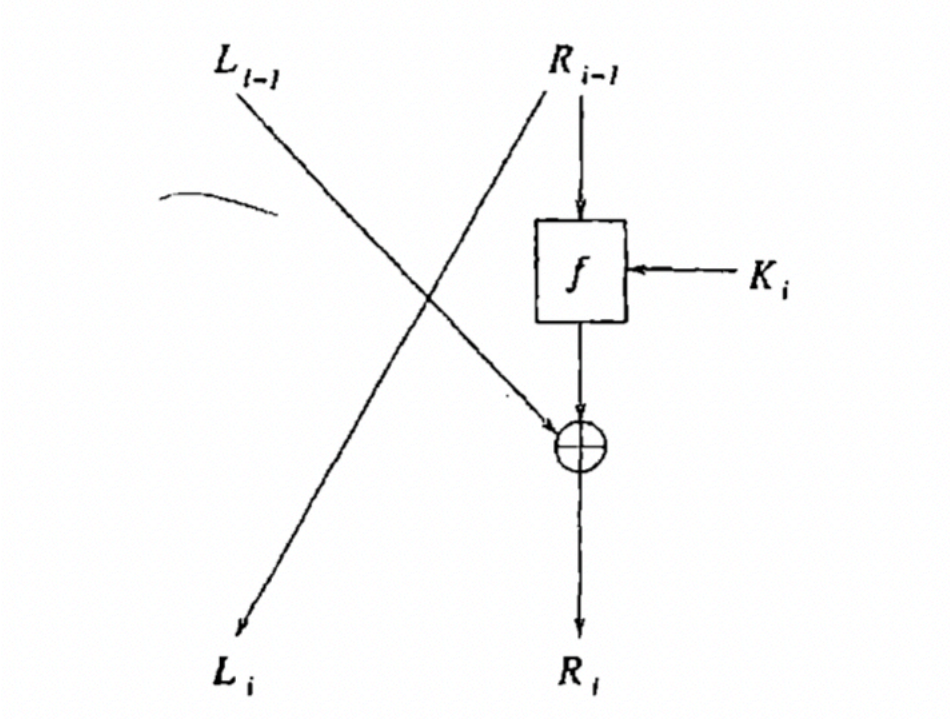


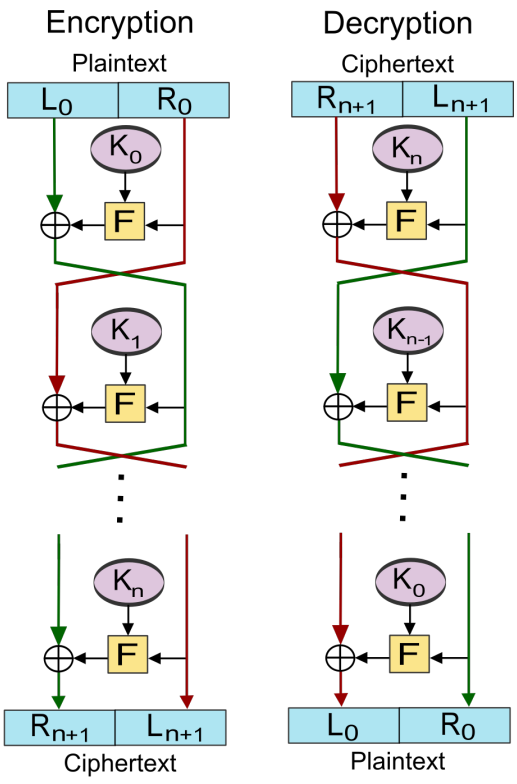
The S-Box - The S-Boxes are basically one-one matching from a 4 bit input to a 3 bit output. In simpler words, transforming a 4 bit input to a 3 bit output by looking through a 'look up table'. The pictures below show the S-boxes. The one on top is S-Box1 and the one on the bottom is S-Box2.

$$S_1 \quad \begin{bmatrix} 101 & 010 & 001 & 110 & 011 & 100 & 111 & 000 \\ 001 & 100 & 110 & 010 & 000 & 111 & 101 & 011 \end{bmatrix}$$

$$S_2 \quad \begin{bmatrix} 100 & 000 & 110 & 101 & 111 & 001 & 011 & 010 \\ 101 & 011 & 000 & 111 & 110 & 010 & 001 & 100 \end{bmatrix}.$$

Now the Algorithm; our small version of DES is made up of this basic algorithmic block shown below. This exact block is performed multiple times and the ciphertext/encrypted block basically ends up as a permutation of the original plaintext.





The ***f*** function is defined as follows:

$$R_2 = L_1 \oplus f(R_1, K_2)$$
$$L_3 = R_2 = L_1 \oplus f(R_1, K_2)$$
$$R_4 = L_3 \oplus f(R_3, K_4) = L_1 \oplus f(R_1, K_2) \oplus f(R_3, K_4).$$

for the upper 4 bits : S1 ($E(R_{i-1}) \oplus K_i$)

for the lower 4 bits : S2 ($E(R_{i-1}) \oplus K_i$)

A implementation of this simplistic DES algorithm on 4 rounds in Python can be found at this link - **Baby_DES**

For more detailed information on the Feistel Algorithm it uses as well as the entire encryption-decryption algorithm, check this link - **Data Encryption Standard**

- ## Differential Cryptanalysis

One of the simplest ways to attack DES is using a method called Differential Cryptanalysis. Differential Cryptanalysis makes a few assumptions:

- You have access to the encryption/decryption machine
- You are allowed an unlimited amount of encryptions and decryptions
- You know the number of rounds the machine performs

Differential Cryptanalysis at its core is a chosen plaintext kind of attack. For our example on how differential cryptanalysis works, let's just assume we have access to a four round machine.

Let the plaintext we inject in be L_0R_0 and the expected cipher text after four rounds be L_4R_4. Let's now, temporarily analyze 3 rounds; starting from L_1R_1. Working out the Math for the encryption process we get the following

Now lets assume we have a secondary input or plaintext $L_1^* R_1^*$; such that $R_1 = R_1^*$. Moreover, let $L_1` R_1`$ denote the difference between $L_1^* R_1^*$ and L_1R_1 where the difference between the the two is defined as the XOR of the two.

$$R_4' = R_4 \oplus R_4^* = L_1' \oplus f(R_3, K_4) \oplus f(R_3^*, K_4).$$
$$R_4' \oplus L_1' = f(R_3, K_4) \oplus f(R_3^*, K_4).$$
$$R_4' \oplus L_1' = f(L_4, K_4) \oplus f(L_4^*, K_4).$$

Working through the math we end up with the above equality. Now our job is to reverse engineer the S-boxes and equate the RHS to the LHS. Solving this equalities for multiple different plaintexts, we can zone in at the value of K.

This algorithm is implemented in code at this link - **3 round differential cryptanalysis**

4 Round Differential Cryptanalysis is basically an extension of this. We can now start from L_0R_0. Our aim now is to chose plaintexts (L_0R_0) to encrypt such that our $L_1^* R_1^*$ and $L_1 R_1$ conditions our met so we can use the same derivations we got in the 3 round differential cryptanalysis. Fortunately for us no matter what we choose as L_0R_0 and $L_0^*R_0^*$, as long as $L_0`R_0`$ equals 011010001100, it will fulfill our $L_1` R_1`$ conditions 3/8 of the time. Even in the times when it doesn't fulfill our criterion, after working through our whole algorithm, the correctly calculated key should appear more than others in a frequency analysis. The working of this can be found at - **4 round differential cryptanalysis**

RSA

- Rivest-Shamir-Adleman

RSA is one of the most common public-key private-key Cryptosystems used today who's workings and security are based on a simple, yet difficult to attack algorithm. RSA strongly follows Kerckhoffs's principle as everything about it is public information except the key (private key). The strength and security lies in the fact that it is super difficult, mathematically, to factor a number that is made up of two large primes.

The Algorithm

- You first start off by picking two really large prime numbers (in the order of 2^{1024}) - **'p'** and **'q'**
- Multiplying these 2 numbers then gives you a new number **'n'** (n = p*q). This is part of the "public key scheme".
- The next next step involves choosing an encryption exponent 'e', such that the **gcd(e, (p-1)(q-1) = 1**
- Next, calculate **'d'** such that **d*e = 1 mod((p-1)(q-1)).**
- You make the values, **n** and **e** public.
- Whenever someone wants to send an encrypted message to you, they calculate $\mathbf{m^e}$ **mod(n)** and send that resulting ciphertext to you.

$$\textbf{ciphertext} = (\textbf{message})^{\textbf{e}} \quad \textbf{mod (n)}$$

- Since only you know the factorization of n, you can calculate d (or you may already know d), and can use this to decrypt the ciphertext.

$$\textbf{message} = (\textbf{ciphertext})^{\textbf{d}} \quad \textbf{mod (n)}$$

Working through the Math and using Euler's Theorem we can see why this works. The beauty about this algorithm is that once encrypted, only you can decrypt the message because you know your private key (**d**) and know the factorization of **n** to compute **d**. Evidently one can see why the security of RSA lies with our inability to factor large primes in real time.

A program that encrypts a message using the RSA Scheme and then decrypts it can be found at this link - **RSA**

Applications of RSA

- RSA by itself is a much slower algorithm than DES or AES. But the beneficial part of RSA is its public-key private-key architecture as it helps us solve the **chicken and egg problem** encountered with cryptosystems like DES and AES (you want to share your key with the person decrypting, but you need another key to encrypt the first key you want to send …..)
- Due to the above reason, RSA is first used to send the key for faster/more secure encryption algorithms like DES and AES
- RSA is used as a form of **Digital Signatures.**

Digital Signatures

Digital Signatures are used to verify senders of information. Like a traditional signature, they need to be unique to the person they are associated with. The public and private key in RSA are interchangeable - in other words, a key that is used to decrypt can also be used to encrypt a message, with its pair being used to decrypt.

How do Digital Signatures work then?

The person in need of verification tries to get access of the public key combo of the person its trying to verify. They then agree on a common plaintext, which the signer encrypts using his private key and sends it to the verifier. The verifier then

uses the public key to decrypt the message and if it matches the agreed upon message, then the verifier knows exactly what it has been signed by.

- # Primality Testing

Fermat's Primality Test

Fermat's Primality Test states that if for any integer in [1, n-1], if $a^{n-1} \equiv 1 \bmod(n)$; then n is **probably prime** and a is a witness for n's primality. If the converse is true, then n is surely composite. The key statement here is - **probably prime**. The only way one can make sure if n is prime, is by checking for all the witnesses in the set [1, n-1].

Miller-Rabin Primality Test

Let $n > 1$ be an odd integer. Write $n - 1 = 2^k m$ with m odd. Choose a random integer a with $1 < a < n-1$.

Compute $b_0 = a^m \pmod n$. If $b_0 = \pm 1 \pmod n$, then stop and declare that n is probably prime. Otherwise, let $b_1 = b_0^2 \pmod n$. If by $b_1 = 1 \pmod n$, then n is composite (and $\gcd(b_0 - 1, n)$ gives a nontrivial factor of n). If $b_1 = -1 \pmod n$, then stop and declare that n is probably prime.

Otherwise, let $b_2 = b_1^2 \pmod n$. If $b_2 = 1 \pmod n$, then n is composite. If $b_2 = -1 \pmod n$, then stop and declare that n is probably prime. Continue in this way until stopping or reaching b_{k-1}. If b_{k-1} is not equal to $-1 \pmod n$, then n is composite.

Solovay-Strassen Primality Test

The Solovay-Strassen Primality Test is defined as follows:
If n is an odd integer, choose any number a, between [1, n-1]. Compute the Legendre Symbol (a/n).

If

$$\frac{a}{n} \equiv a^{\frac{n-1}{2}} mod(n)$$

then n is **probably prime**. If the converse is true, then n is composite for sure.

Implementation of these three Primality Tests can be found at - **is_prime.py**

About me

Hi! Congratulations on making it this far! Just wanted to leave you with a brief background about who I am, what I like to do and where I wanna reach.

My name's Sherwyn Braganza, but you probably already know that. I'm a student at South Dakota School of Mines and Technology, working on getting my Masters Degree in Computer Science and Engineering. As far as undergrad studies goes, I graduated in Dec 20 (yeah, COVID semester) from South Dakota School of Mines and Technology with a B.Sc. in Computer Engineering and a minor in Computer Science, with emphasis on Computer Hardware Design/ Development.

Cryptography is an amazing field of study and there's so much more that I would have loved to include in this document (I will work on updating and adding more). Needless to say, it has become one of the Primary Fields of Study I am interested in (shout out to Dr. Christer Karlsson for getting me interested in this). Leaving all the technical aspects aside, some of the few philosophical takeaways from this class are:

1) Always use two-factor authorization.

2) If you want anything to be secure, everything of it should be public and open source except the key that is used to secure it.

3) Doing math on the whiteboard is hard.

4) Never run live simulations during a presentation.

5) Listening to Minecraft Music while coding helps.

6) We might be living in a simulation