What is Cryptography?

Cryptography is a practice and study of techniques for secure communication in the presence of third parties

The basic concept is that we want to make sure that the message send by the sender is read only by the receiver exclusively

Some Important terms:

- Plain Text: The message itself that we want to encrypt
- Cipher Text: The encrypted message
- Key: This is a sequence that is needed for both Encryption and Decryption

Encryption: The process of encoding the plain text such a way that only authorized parties can access it
cipher_text=function(plaintext,key)

Decryption: The process of decoding a given cipher text
plain_text=function^{-1} (cipher_text,key)

Types of Cryptography:

1)Private Key Cryptography

Symmetric encryption

Same key is used for encryption and decryption

Decryption function is inverse of the encryption function

Disadvantage: The key must be exchange

2)**Public Key Cryptography**

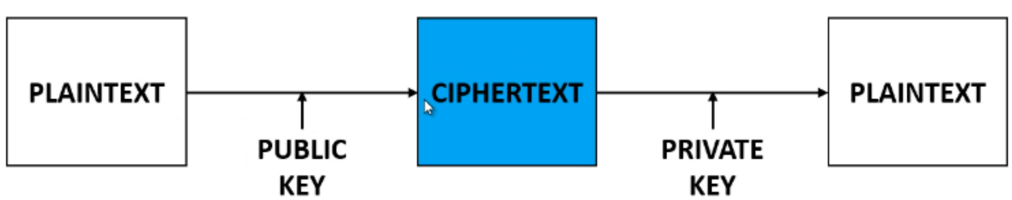
Asymmetric encryption

We use two keys: (public key and private key)

Keep the private key secret but everyone can know about the public key

Eg) If Alice wants to send a message to bob,

Alice will encrypt the message using Bob's public key and Bob will decrypt using private key



Private Key Cryptography

1)Caesar Cipher

ROT- Cipher: It is a type of substitution cipher: Where we shift ever single letter in plain text by a fixed number of lengths

Key itself is the number of letters used for shifting

Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char
0	0	[NULL]	32	20	[SPACE]	64	40	@	96	60	`
1	1	[START OF HEADING]	33	21	!	65	41	A	97	61	a
2	2	[START OF TEXT]	34	22	"	66	42	B	98	62	b
3	3	[END OF TEXT]	35	23	#	67	43	C	99	63	c
4	4	[END OF TRANSMISSION]	36	24	$	68	44	D	100	64	d
5	5	[ENQUIRY]	37	25	%	69	45	E	101	65	e
6	6	[ACKNOWLEDGE]	38	26	&	70	46	F	102	66	f
7	7	[BELL]	39	27	'	71	47	G	103	67	g
8	8	[BACKSPACE]	40	28	(	72	48	H	104	68	h
9	9	[HORIZONTAL TAB]	41	29	)	73	49	I	105	69	i
10	A	[LINE FEED]	42	2A	*	74	4A	J	106	6A	j
11	B	[VERTICAL TAB]	43	2B	+	75	4B	K	107	6B	k
12	C	[FORM FEED]	44	2C	,	76	4C	L	108	6C	l
13	D	[CARRIAGE RETURN]	45	2D	-	77	4D	M	109	6D	m
14	E	[SHIFT OUT]	46	2E	.	78	4E	N	110	6E	n
15	F	[SHIFT IN]	47	2F	/	79	4F	O	111	6F	o
16	10	[DATA LINK ESCAPE]	48	30	0	80	50	P	112	70	p
17	11	[DEVICE CONTROL 1]	49	31	1	81	51	Q	113	71	q
18	12	[DEVICE CONTROL 2]	50	32	2	82	52	R	114	72	r
19	13	[DEVICE CONTROL 3]	51	33	3	83	53	S	115	73	s
20	14	[DEVICE CONTROL 4]	52	34	4	84	54	T	116	74	t
21	15	[NEGATIVE ACKNOWLEDGE]	53	35	5	85	55	U	117	75	u
22	16	[SYNCHRONOUS IDLE]	54	36	6	86	56	V	118	76	v
23	17	[END OF TRANS. BLOCK]	55	37	7	87	57	W	119	77	w
24	18	[CANCEL]	56	38	8	88	58	X	120	78	x
25	19	[END OF MEDIUM]	57	39	9	89	59	Y	121	79	y
26	1A	[SUBSTITUTE]	58	3A	:	90	5A	Z	122	7A	z
27	1B	[ESCAPE]	59	3B	;	91	5B	[	123	7B	{
28	1C	[FILE SEPARATOR]	60	3C	<	92	5C	\	124	7C	|
29	1D	[GROUP SEPARATOR]	61	3D	=	93	5D	]	125	7D	}
30	1E	[RECORD SEPARATOR]	62	3E	>	94	5E	^	126	7E	~
31	1F	[UNIT SEPARATOR]	63	3F	?	95	5F	_	127	7F	[DEL]

Consider with example:

If we only consider Alphabets:

A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25

Algorithm

Encryption:

$$E_n(x)=(x + key) \bmod 26$$

Decryption:

$$E_n(x)=(x - key) \bmod 26$$

Why is Mod 26 important?

To make sure the encrypted text is within the range of (0, sizeOf(alphabets)-1)

21+8=29

But there is no 29 in English Alphabet, so 29 mod 26=3

-3%26=23

CODE

```python
ALPHABET="".join(chr(x) for x in range(128))
key=3
def caesar_encryption(plain_text,key):
    cipher_text='' #To store cipher Text
    plain_text=plain_text.upper() #To make case insensitive
    for c in plain_text:
        index=ALPHABET.find(c) #Find the index in ALPHABET
        index=(index+key)%len(ALPHABET)
        cipher_text=cipher_text+ALPHABET[index]
    return cipher_text

def caesar_decryption(cipher_text,key):
    plain_text=''
    for c in cipher_text:
        index=ALPHABET.find(c) #Find the index in ALPHABET
        index=(index-key)% len(ALPHABET)
        plain_text=plain_text+ALPHABET[index]
    return plain_text
```

Cracking of Caesar-cipher

Disadvantage

Limited sets of keys: 26(When we consider only English letters)

256(When we key when we consider ASCII)

1) **Brute Force Attack:** Take the advantage of limited set of keys

   Use all the possible set of values in the range (0, sizeof(ALPHABET)-1)

   Now use this output to match with **English dictionary** to check for correct key

```python
def bruteCeacer (cipher_text):
    for key in range(len(ALPHABET)):
        plain_text=caesar_decryption(cipher_text,key)
        check=languageDetectorAlgorithm(plain_text)
        if check is True:
            print('Key: '+ str(key)+ '\n\t'+plain_text)
```

```python
def languageDetectorAlgorithm(inputText):
    EnglishWords=[]
    match=0
    totalWords=len(inputText.split(' '))
    Checkwordsinput=[]
    dictionary = open('english_words.txt', 'r')
    for word in dictionary.read().split('\n'):
        EnglishWords.append(word)
    for word in inputText.split(' '):
        word=word.upper()
        if word in EnglishWords:
            match=match+1
    if (float(match)/totalWords)*100 >=50:
        return True
    else:
        return False
```

2) **Frequency-analysis**: Take advantage of information leaking

In English language there are some letters that more frequent than other
Blank space,E,A,O,I and T are the most frequent letters

Algorithm:

Possible value of keys:

key=value of ciphertext's most frequent letter – value of Blank Space

key=value of ciphertext's most frequent letter – value of E

key=value of ciphertext's most frequent letter – value of A

```
def frequencyAnalysis(text):
    text=text.upper()
    letterFrequencies={}

    for letter in ALPHABET:
        letterFrequencies[letter]=0

    for letter in text:
        letterFrequencies[letter] += 1
    return letterFrequencies
```

```
def crackCaesar(cipher_text):
    freq=frequencyAnalysis(cipher_text)
    freq = sorted(freq.items(), key=lambda x: x[1], reverse=True)
    possibleKeys=[]
    key=ALPHABET.find(freq[0][0])-ALPHABET.find(' ')
    possibleKeys.append(key)
    key=ALPHABET.find(freq[1][0])-ALPHABET.find('E')
    if key not in possibleKeys:
        possibleKeys.append(key)
    key=ALPHABET.find(freq[1][0])-ALPHABET.find('A')
    if key not in possibleKeys:
        possibleKeys.append(key)

    for key in possibleKeys:
        plain_text=caesar_decryption(cipher_text,key)
        check=languageDetectorAlgorithm(plain_text)
        if check is True:
            print('Key: '+ str(key)+ '\n\t'+plain_text)
```

Vigenere Cipher

Polyalphabetic Substitution: Similar to Caesar cipher but instead of one key we use many key

Why Vigenere Cipher is better than Caesar cipher

Uses a word as private key:

So we have more than 26 possible keys

Size of keys pace= $26^{\text{Size of key}}$

Algorithm

Encryption

$E_i(x_i)=(x_i + k_i)$ mod 26

We use i^{th} letter of the key for i^{th} encryption

Decryption

$D_i(x_i)=(x_i - k_i)$ mod 26

Code

```
def vigenere_encrypt(plain_text,key):
    plain_text = plain_text.upper() #Make text case insensitive
    key = key.upper()
    cipher_text = '' #To store the cipher text
    key_index = 0 #Index of keys
    for letter in plain_text:
        if ord(letter)>=65 and ord(letter)<=90:
            index = (ALPHABET.find(letter)+ALPHABET.find(key[key_index])) % len(ALPHABET
            cipher_text = cipher_text + ALPHABET[index]
            key_index = key_index + 1
            if key_index == len(key):
                key_index = 0
        else:
            cipher_text=cipher_text+' '
    return cipher_text
```

```
def vigenere_decrypt(cipher_text,key):
    cipher_text = cipher_text.upper() #Make text case insensitive
    key = key.upper()
    plain_text = '' #To store the cipher text
    key_index = 0 #Index of keys
    for letter in cipher_text:
        if ord(letter)>=65 and ord(letter)<=90:
            index = (ALPHABET.find(letter)-ALPHABET.find(key[key_index])) % len(ALPHABET
            plain_text = plain_text + ALPHABET[index]
            key_index = key_index + 1
            if key_index == len(key):
                key_index = 0
        else:
            plain_text=plain_text+' '
    return plain_text
```

Drawback:

1)Brute Force Attack:

This is highly unlikely because the number of possible key = $26^{\text{Size of the key}}$

2)Dictionarry Attack: Usually we use meaingfull words as the key of the cipher so we use all the known words in English Dictionary as key

3)**Kaiski-algorithm:**

If we know the size of the key, Then Vigenere cipher is as weak as Caesar Cipher

Step 1) Find the Length of the key

Step 2) If n= length of the key

Divide the ciphered text into n parts

Step 3) Use frequency analysis on each substring to find the subkey

Step 4) Combine the subkeys to get the main key and now use it to decrypt the text

Finding the Length of Key (Take Advantage of Information Leaking)

Let us considered cipher text,

Shift by 1 and calculate the NO of Matches

X= ABCDDXERVXX **No of matches = ABC**

=ABCDDXERVXX

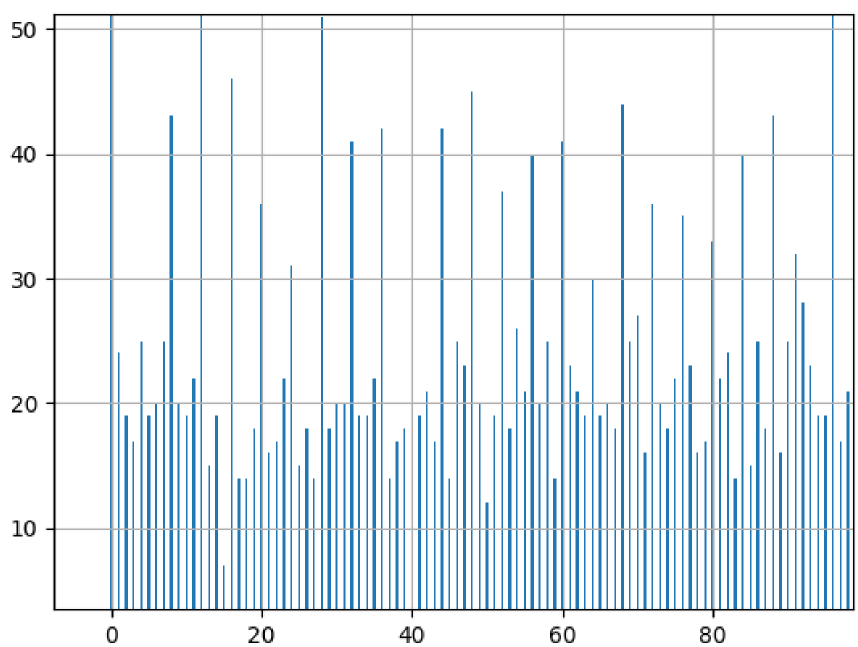
Shift by 2 and calculate the NO of Matches

X= ABCDDXERVXX **No of Matches = PQR**

ABCDDXERVXX

Continue this process until the end of the string.

Plot the graph from 0 shifts to len(cipher)



KeyLength = (Highest number of lower frequencies between two peaks)+1

Graphical method:

Plot the graph and manually observe

Mathematical Formula:

Identify all the peaks and store them in a list (Peak[ ])

Peak >= Std Dev + Mean

Now store

The difference of index of the peaks in peakdiff [ ]

KeyLength= Mode(peakDiff [ ])

```python
def guess_key_length(cipher: str):
    res = []
    for shift in range(len(cipher)):
        matches = 0
        for i in range(len(cipher)):
            if cipher[i] == cipher[(i - shift + len(cipher)) % len(cipher)]:
                matches=matches+1
        res.append(matches)
    dat_means = st.mean(res[1:])
    dat_std_dev   = math.sqrt(st.variance(res[1:]))
    peaks = []
    for i in range(len(res)):
        if res[i] >= dat_std_dev + dat_means: peaks.append(i)
    peak_diff = []
    for i in range(len(peaks) - 1):
        peak_diff.append(peaks[i+1] - peaks[i])

    return st.mode(peak_diff)
```