▦differential XORing

I. Passphrase: a string used as the first "previous encrypted block" for the encryption of first block

II. Algorithm

1. Encypted
    1. Given: passphrase, blocksize, key
    2. Reduce the passphrase to a bit array of size BLOCKSIZE

```
bv_iv = BitVector(bitlist = [0]*BLOCKSIZE)
for i in range(0,len(PassPhrase) // numbytes):
    textstr = PassPhrase[i*numbytes:(i+1)*numbytes]
    bv_iv ^= BitVector( textstring = textstr )
```

    3. Reduce the key to a bit array of size BLOCKSIZE

```
key_bv = BitVector(bitlist = [0]*BLOCKSIZE)
for i in range(0,len(key) // numbytes):
    keyblock = key[i*numbytes:(i+1)*numbytes]
    key_bv ^= BitVector( textstring = keyblock )
```

    4. Carry out differential XORing of bit blocks and encryption

```
previous_block = bv_iv
bv = BitVector( filename = sys.argv[1] )
while (bv.more_to_read):
    bv_read = bv.read_bits_from_file(BLOCKSIZE)
    if len(bv_read) < BLOCKSIZE:
        bv_read += BitVector(size = (BLOCKSIZE - len(bv_read)))
    bv_read ^= key_bv
    bv_read ^= previous_block
    previous_block = bv_read.deep_copy()
    msg_encrypted_bv += bv_read
```

    5. Convert the encrypted bitvector into a hex string:

```
outputhex = msg_encrypted_bv.get_hex_string_from_bitvector()
```

2. Decrypted
    1. Given: passphrase, blocksize, key
    2. Reduce the passphrase to a bit array of size BLOCKSIZE:
    3. Create a bitvector from the ciphertext hex string:

```
FILEIN = open(sys.argv[1])
encrypted_bv = BitVector( hexstring = FILEIN.read() )
```

    4. Reduce the key to a bit array of size BLOCKSIZE
    5. Carry out differential XORing of bit blocks and decryption:

```
previous_decrypted_block = bv_iv
for i in range(0, len(encrypted_bv) // BLOCKSIZE):
    bv = encrypted_bv[i*BLOCKSIZE:(i+1)*BLOCKSIZE]
    temp = bv.deep_copy()
    bv ^=  previous_decrypted_block
    previous_decrypted_block = temp
    bv ^=  key_bv
    msg_decrypted_bv += bv
```

6. Extract plaintext from the decrypted bitvector

▦plaintext: what you want to encrypt

▦ciphertext: The encrypted output

▦enciphering or encryption: The process by which plaintext is converted into ciphertext

▦encryption algorithm: The sequence of data processing steps that go into transforming plaintext into ciphertext. Various parameters used by an encryption algorithm are derived from a secret key. The encryption and decryption algorithms are placed in the public domain. Secret algorithm is less likely to be subject to the same level of testing and scrutiny that a public algorithm is.

▦secret key: used to set some or all of the various parameters used by the encryption algorithm.

1. symmetric key cryptography: the same secret key is used for encryption and decryption.
2. asymmetric key cryptography / public key cryptography: encryption and decryption keys are different, one of them is placed in the public domain.

▦deciphering or decryption: Recovering plaintext from ciphertext

▦decryption algorithm: The sequence of data processing steps that go into transforming ciphertext back into plaintext.

▦cryptography: The many schemes available today for encryption and decryption

▦cryptographic system / cipher: Any single scheme for encryption and decryption

▦block cipher: processes a block of input data at a time and produces a ciphertext block of the same size.

▦stream cipher: encrypts data on the fly, usually one byte at a time.

▦cryptanalysis (breaking the code): relies on a knowledge of the encryption algorithm and some knowledge of the possible structure of the plaintext.

The precise methods used for cryptanalysis depend on whether the attacker has just a piece of ciphertext, or pairs of plaintext and ciphertext, how much structure is possessed by the plaintext, and how much of that structure is known to the attacker.
▩key space: total number of all possible keys that can be used in a cryptographic system. For example, DES uses a 56-bit key. So the key space is of size 2^{56}
▩brute-force attack: When encryption and decryption algorithms are publicly available, a brute-force attack means trying every possible key on a piece of ciphertext until an intelligible translation into plaintext is obtained.
▩codebook attack: mapping from the plaintext symbols to the ciphertext symbols. In a codebook attack, the attacker tries to acquire as many as possible of the mappings between the plaintext symbols and the corresponding ciphertext symbols. You can think of a codebook as the mapping between the plaintext bit blocks and the ciphertext bit blocks, with a ciphertext bit block being related to the corresponding plaintext bit block through an encryption key.
▩algebraic attack: express the plaintext-to-ciphertext relationship as a system of equations. Given a set of (plaintext, ciphertext) pairs, you try to solve the equations for the encryption key.
▩time-memory tradeoff in attacking ciphers: The brute-force and the codebook attacks represent two opposite cases in terms of time versus memory needs of the algorithms. Pure brute-force attacks have very little memory needs, but can require inordinately long times to scan through all possible keys. Codebook attacks can in principle yield results
instantaneously, but their memory needs can be humongously large.
▩time-memory tradeoff attacks: reduce the time taken by a brute-force attack if we use memory to store intermediate results obtained from the current computational steps
▩backdoor: allows an intruder to get inside a networked device without user uthentication credentials. Backdoors may be created by malware or by exploiting vulnerabilities in the security protocols used in a networked device.
▩commercial spyware: application that transmits sensitive information off the device without user consent and does not display a persistent notification that this is happening.

▦denial of service: prevent legitimate users from accessing a network resource. Malware in a machine may turn it into a devicefor mounting a denial-of-service attack on a network resource.
▦hostile downloader: application that is not in itself potentially harmful, but downloads other potentially harmful apps.
▦mobile billing fraud: application that charges the user in an intentionally misleading way.

1. sms fraud: application that charges users to send premium SMS without consent, or tries to disguise its SMS activities by hiding disclosure agreements or SMS
   messages from the mobile operator notifying the user of charges or confirming subscription.
2. call fraud: application that charges users by making calls to premium-rate telephone numbers without user consent.
3. toll fraud: application that tricks users to subscribe or purchase content via

   their mobile phone bill. Toll Fraud includes any type of billing except Premium SMS and premium calls. WAP fraud is one of the most prevalent types of Toll fraud. WAP fraud can include tricking users to click a button on a silently loaded transparent WebView. Upon performing the action, a recurring subscription is initiated, and the confirmation SMS or email is often hijacked to prevent users from noticing the financial transaction.

▦phishing: An application that pretends to come from a trustworthy source, requests a users authentication credentials and/or billing information, and sends the data to a third party.
▦mobile unwanted software (MUwS): application that collects at least one of the following without user consent: • Information about installed applications • Information about third-party accounts • Names of files on the device
▦privilege escalation: application that compromises the integrity of the system by breaking the application sandbox, or changing or disabling access to core security-related functions. Allow an app to steal credentials from other apps and to prevent its own removal. Privilege escalation apps that root devices without user permission are classified as rooting apps.

1. Non-malicious rooting apps: let the user know in advance that they are going to root the device and they do not execute other

potentially harmful actions.

2. Malicious rooting apps: do not inform the user that they will root the device, or they inform the user about the rooting in advance but also execute other harmful actions.

▦ransomware: makes your computer unusable by encrypting all your files

▦spam: unsolicited, unwanted, and frequently annoying email messages that land in your computer or mobile device

▦spyware: application that transmits sensitive information off the device.

▦SSL (Secure Socket Layer) /TLS (Transport Layer Security): certificate based client and
server authentication made possible by the SSL/TLS protocol that makes e-commerce possible. An SSL/TLS certificate for an e-commerce website makes available the public key used by the website.

▦TCP/IP: two different foundational protocols that govern how information is exchanged between two different hosts in the internet. TCP as sitting on top of IP. TCP protocol adds handshaking to this interaction in order to make sure that every data packet sent by a host was actually received by the other host.

▦tor: route anonymizing protocol that makes it easier for folks in countries with heavy censorship and controls to access foreign websites like Google and Facebook.

▦trojan: application that appears to be benign and performs undesirable actions against the user. A trojan will have an innocuous app component and a hidden harmful component.

▦VPN (Virtual Private Network): overlay network that allows a set of hosts to communicate with one another confidentially using IPSec, which is a secure version of the IP protocol.

▦Two building blocks of all classical encryption techniques are substitution and transposition.

1. Substitution: replacing an element of the plaintext with an element of ciphertext. The same overall substitution rule may be applied to every element of the plaintext, or the substitution rule may vary from position to position in the plaintext.
2. Transposition/permutation: rearranging the order of appearance of the elements of the plaintext. Transposition may be carried out after or before substitution

▦CAESAR CIPHER (Substitution): Each character of a message is replaced by a haracter x position down in the alphabet. Encryption and decryption formula for replacing each character p of the plaintext with a character c of the ciphertext can be expressed as: (k would be the secret key)

$$\begin{aligned} c &= E(k,p) &= (p+k) \; mod \; 26 \\ p &= D(k,c) &= (c-k) \; mod \; 26 \end{aligned}$$

```
plaintext:    are you ready

ciphertext:   DUH BRX UHDGB
```

▦save "hello" as a file, the file will have 6 bytes. Due to the new line added before EOF

▦Base64 encoding: replaced every consecutive 6 bits with one of 64 possible cipher

Characters (3 bytes into 4 64-base characters as printable characters). Non-printable character are control characters so cannot be used.

characters *M*, *a*, and *n* are stored as the byte values `77`, `97`,

and `110`, which are the 8-bit binary values `01001101`, `01100001`,

and `01101110`. These three values are joined together into a 24-bit

string, producing `010011010110000101101110`. Groups of 6 bits (6

bits have a maximum of 2^6 = 64 different binary values) are converted into individual numbers from left to right (in this case, there are four numbers in a 24-bit string), which are then converted into their corresponding Base64 character values.

Source	Text (ASCII)	M								a								n							
	Octets	77 (0x4d)								97 (0x61)								110 (0x6e)							
Bits		0	1	0	0	1	1	0	1	0	1	1	0	0	0	0	1	0	1	1	0	1	1	1	0
Base64 encoded	Sextets	19						22						5						46					
	Character	T						W						F						u					
	Octets	84 (0x54)						87 (0x57)						70 (0x46)						117 (0x75)					

If there are only two significant input octets (e.g., 'Ma'), or when the last input group contains only two octets, all 16 bits will be captured in the first three Base64 digits (18 bits); the two least significant bits of the last content-bearing 6-bit block will turn out to be zero, and discarded on decoding (along with the following `=` padding characters)

Source	Text (ASCII)	M								a													
	Octets	77 (0x4d)								97 (0x61)													
Bits		0	1	0	0	1	1	0	1	0	1	1	0	0	0	0	1	0	0						
Base64 encoded	Sextets	19						22						4						Padding					
	Character	T						W						E						=					
	Octets	84 (0x54)						87 (0x57)						69 (0x45)						61 (0x3D)					

If there is only one significant input octet (e.g., 'M'), or when the last input group contains only one octet, all 8 bits will be captured in the first two Base64 digits (12 bits); the four least significant bits of the last content-bearing 6-bit block will turn out to be zero, and discarded on decoding (along with the following `=` padding characters):

Source	Text (ASCII)	M																					
	Octets	77 (0x4d)																					
Bits		0	1	0	0	1	1	0	1	0	0	0	0												
Base64 encoded	Sextets	19						16						Padding						Padding					
	Character	T						Q						=						=					
	Octets	84 (0x54)						81 (0x51)						61 (0x3D)						61 (0x3D)					

Index	Binary	Char	Index	Binary	Char	Index	Binary	Char	Index	Binary	Char
0	000000	A	16	010000	Q	32	100000	g	48	110000	w
1	000001	B	17	010001	R	33	100001	h	49	110001	x
2	000010	C	18	010010	S	34	100010	i	50	110010	y
3	000011	D	19	010011	T	35	100011	j	51	110011	z
4	000100	E	20	010100	U	36	100100	k	52	110100	0
5	000101	F	21	010101	V	37	100101	l	53	110101	1
6	000110	G	22	010110	W	38	100110	m	54	110110	2
7	000111	H	23	010111	X	39	100111	n	55	110111	3
8	001000	I	24	011000	Y	40	101000	o	56	111000	4
9	001001	J	25	011001	Z	41	101001	p	57	111001	5
10	001010	K	26	011010	a	42	101010	q	58	111010	6
11	001011	L	27	011011	b	43	101011	r	59	111011	7
12	001100	M	28	011100	c	44	101100	s	60	111100	8
13	001101	N	29	011101	d	45	101101	t	61	111101	9
14	001110	O	30	011110	e	46	101110	u	62	111110	+
15	001111	P	31	011111	f	47	101111	v	63	111111	/
padding		=									

▦When you increase the size of a number by a factor of 10, you are increasing the size by
one order ofmmagnitude. So when we say that the keyspace is 10 orders of 10 magnitude larger, that means that the keyspace is larger by a factor of 1
▦monoalphabetic cipher, you use the same substitution rule to find the replacement ciphertext letter for each letter of the alphabet in the plaintext message.

1. EX. random permutation, encryption key is the sequence of substitution letters, key space = 26!
2. any monoalphabetic substitution cipher, regardless of the size of the key space, can be easily broken with a statistical attack.
3. When the plaintext is plain English, a simple form of statistical attack consists measuring the frequency distribution for single characters, for pairs of characters, for triples of characters, and so on, and comparing those with similar statistics for English.
    1. Equally powerful statistical inferences can be made by comparing the in the cipher relative frequencies for pairs and triples of characters text and the language believed to be used for the plaintext.
    2. Digrams: Pairs of adjacent characters. can represent this table by the joint probability p(x,y) where denotes the first letter of a digram and y the second letter.
    3. Trigrams: triples of characters

▦PLAYFAIR CIPHER (multiple-character substitution)

1. Algorithm ()
    1. choose an encryption key, make sure there are no duplicate characters in key. Key = smythework
    2. enter the characters in the key in the cells of a 5*5 matrix in a left-to-right and top-to-down fashion starting with the first cell at the top-left corner.
    3. fill the rest of the cells of the matrix with the remaining characters in the alphabet and do so in alphabetic order.
    4. for any given pair of plaintext characters, you use the following three rules to determine the corresponding pair of ciphertext characters:

S	M	Y	T	H
E	W	O	R	K
A	*B*	*C*	*D*	*F*
G	*I/J*	*L*	*N*	*P*
Q	*U*	*V*	*X*	*Z*

        a. Two plaintext letters that fall in the same row: replaced by letters to the right of each in the row. “bf”-> “ CA”
        b. Two plaintext letters that fall in the same column: replaced by the letters just below them in the column. “ol” -> “CV”
        c. Otherwise, for each plaintext letter in a pair, replace it with the letter that is in the same row but in the column of the other letter. “gf”-> “PA”
    5. Before the substitution rules are applied, you must insert a chosen filler letter (let say it is x) between any repeating letters in the plaintext. So a plaintext word such as “hurray” becomes “hurxray”.
2. the cipher does alter the relative frequencies associated with the individual letters and with digrams and with trigrams, but not sufficiently
3. The cryptanalysis of the Playfair cipher is also aided by the fact that a digram and its reverse will encrypt in a similar fashion. That is, if AB encrypts to XY, then BA will encrypt to YX. So by looking for words that begin and end in reversed digrams, EX. receiver, departed, repairer, redder, denuded

▦HILL CIPHER(multiple-character substitution)

$$\mathbf{K} = \begin{bmatrix} k_{11} & k_{12} & k_{13} \\ k_{21} & k_{22} & k_{23} \\ k_{31} & k_{32} & k_{33} \end{bmatrix}$$

1. Algorithm:
    1. assign an integer to each letter of the alphabet. Ex. integers 0 through 25 to the letters a through z of the plaintext
    2. encryption key, call it K, consists of a 33 matrix of integers
    3. transform three letters at a time from the plaintext, the letters being represented by the numbers p1, p2, p3 into three ciphertext letters c1, c2, c3

$$\begin{aligned} c_1 &= (k_{11}p_1 + k_{12}p_2 + k_{13}p_3) \bmod 26 \\ c_2 &= (k_{21}p_1 + k_{22}p_2 + k_{23}p_3) \bmod 26 \\ c_3 &= (k_{31}p_1 + k_{32}p_2 + k_{33}p_3) \bmod 26 \end{aligned}$$

2. formula
    1. encryption: $\vec{\boldsymbol{C}} = [\boldsymbol{K}]\vec{\boldsymbol{P}}\ \boldsymbol{mod}\ \mathbf{26}$
    2. decryption: $\vec{\boldsymbol{P}} = [\boldsymbol{K}^{-1}]\vec{\boldsymbol{C}}\ \boldsymbol{mod}\ \mathbf{26}$
3. keyspace can be made extremely large by choosing the matrix elements from a large set of integers or larger matrices
4. But it has zero security when the plaintext-ciphertext pairs are known. The key matrix can be calculated easily from a set of

known $\vec{C}, \vec{P}$ pairs

▦monoalphabetic cipher, the same substitution rule is used at every character position in the plaintext message.

▦polyalphabetic cipher: the substitution rule changes continuously from one character position to the next in the plaintext according to the elements of the encryption key.

▦Vigenere cipher (polyalphabetic cipher)

1. Algorithm: align the encryption key with the plaintext message. Consider each letter of the encryption key denoting a shifted Caesar cipher, the shift orresponding to the letter of the key.

```
key:          abracadabraabracadabraabracadabraab
plaintext:    canyoumeetmeatmidnightihavethegoods
ciphertext:   CBEYQUPEFKMEBK.....................
```

encryption key	*plain text letters*				
letter	a	b	c	d	....
	substitution letters				
a	A	B	C	D	....
b	B	C	D	E	....
c	C	D	E	F	....
d	D	E	F	G	....
e	E	F	G	H	....
.	.	.	.	.	.
.	.	.	.	.	.
z	Z	A	B	C	....

2. Since there exist in the output multiple ciphertext letters for each plaintext letter, you would expect that the relative frequency distribution would be effectively destroyed. The longer the encryption key, the greater the masking of the structure of the plaintext. The best possible key is as long as the plaintext message and consists of a purely random permutation of the 26 letters of the alphabet
3. to break the Vigenere cipher

    1. estimate the length of the encryption key by using Kasiski Examination: examining the ciphertext for sequences of characters that are repeated. The distances between the repeated occurrences of character strings in the ciphertext can serve as possible candidates for the length of the encryption key. If there are several such candidates, one works with the greatest common divisor all possible values as the most likely choice for the key length.
    2. if the estimated length of the key is N, then the cipher consists of N monoalphabetic substitution ciphers and the plaintext letters at positions 1, N, 2N, 3N, etc., will be encoded by the same monoalphabetic cipher.
    3. accumulate the ciphertext characters separately at intervals of N, 2N, 3N, etc., and subject each of the accumulations separately to a statistical analysis.
4. rotors are used in the electromechanical hardware for implementing a polyalphabetic cipher, such machines are commonly referred to as rotor machine

▦pure permutation cipher: write your plaintext message along the rows of a matrix of some size. You generate ciphertext by reading along the columns. The order in which you read the columns is determined by the encryption key

```
key:                2 5 3 1 6 4

plaintext:          m e e t m e
                    a t m i d n
                    i g h t f o
                    r t h e g o
                    d i e s x y

ciphertext:         ETGTIMDFGXEMHHEMAIRDENOOYTITES
```

▦A, B, and C are bit arrays, ⊕ denotes the XOR operator

1. [A⊕B] ⊕ C = A ⊕ [B⊕C]
2. A⊕A = 0
3. A⊕0 = A

▦ideal block cipher: replace a block of N bits from the plaintext with a block of N bits from the ciphertext.

1. the relationship between the input blocks and the output block is completely random. But it must be invertible for decryption to

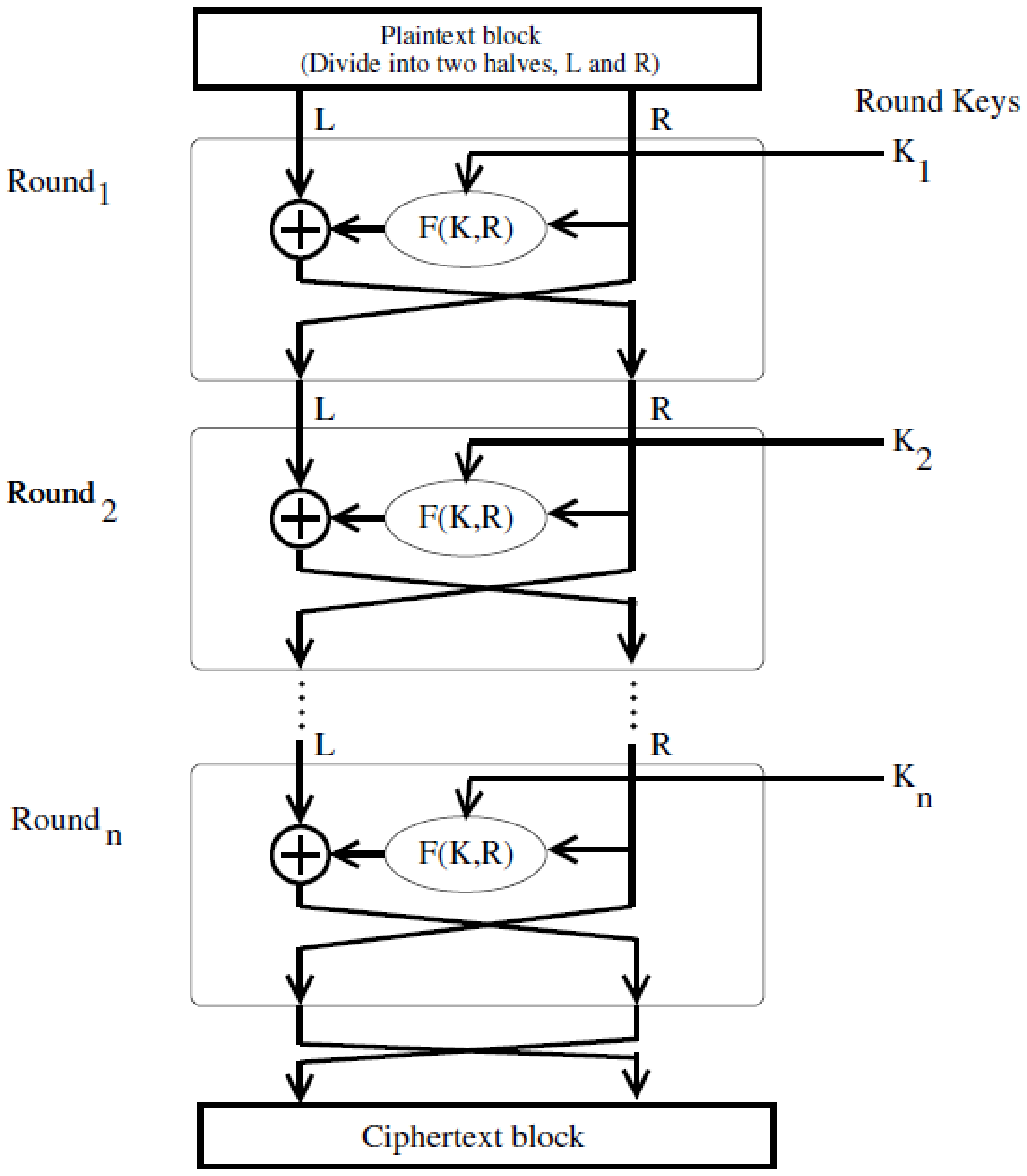
work. Therefore, it has to be one-to-one mapping.

2. The mapping from the input bit blocks to the output bit blocks can also be construed as a mapping from the integers corresponding to the input bit blocks to the integers corresponding to the output bit blocks. EX. 4-bit is 0~$\mathbf{2^4}$
3. encryption key for the ideal block cipher is the codebook itself, meaning the table that shows the relationship between the input blocks and the output blocks.
4. construct the codebook by displaying just the output blocks in the order of the integers corresponding to the input blocks. Ex. 64-bit block -> codebook size (size of the encryption key) is 64$*\mathbf{2^{64}}$
5. The size of the encryption key would make the ideal block cipher an impractical idea. Think of the logistical issues related to the transmission, distribution, and storage of such large keys.

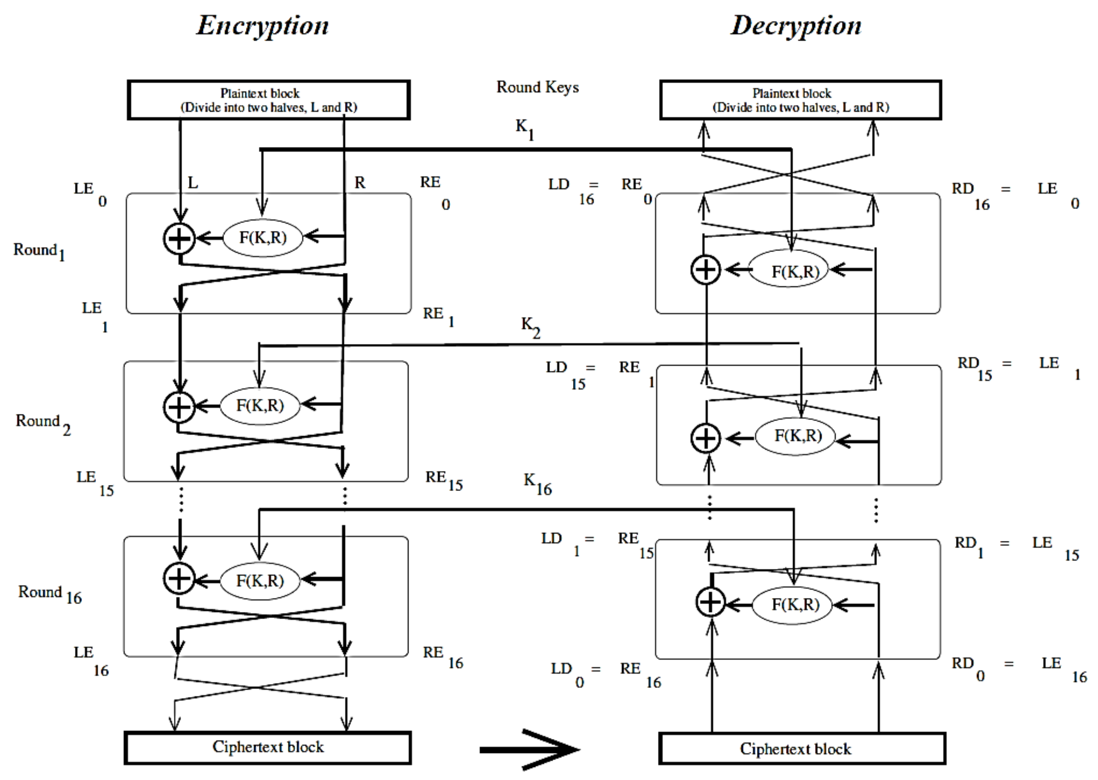
▦Feistel structure consists of multiple rounds of processing of the plaintext, with each round consisting of a substitution step followed by a permutation step.

1. Algorithm
    1. input block to each round is divided into two halves L and R.
    2. In each round, the right half of the block, R, goes through unchanged. But the left half, L, goes through an operation (Function Feistel) that depends on R and the encryption key.
    3. permutation step at the end of each round consists of swapping the modified L and R. Therefore, the L for the next round would be R of the current round. And R for the next round be the output L of the current round.
    4. Feistel Structure: Encryption: relationship between the output of $\mathbf{i}^{th}$ round and the output of the $\mathbf{i-1}^{th}$ round

$$\begin{array}{lcl} LE_i & = & RE_{i-1} \\ RE_i & = & LE_{i-1} \oplus F(RE_{i-1}, K_i) \end{array}$$



2. The output of each round during decryption is the input to the corresponding round during encryption except for the left-right switch between the two halves. The above result is independent of the precise nature of the Feistel function
   LD0 = RE16, LE16 = RD0
3. decryption algorithm is exactly the same as the encryption algorithm with the only difference that the round keys are used in the reverse order. KE1 = KD16



$$\begin{aligned} LD_1 &= RD_0 \\ &= LE_{16} \\ &= RE_{15} \end{aligned}$$

$$\begin{aligned} RD_1 &= LD_0 \oplus F(RD_0, K_{16}) \\ &= RE_{16} \oplus F(LE_{16}, K_{16}) \\ &= [LE_{15} \oplus F(RE_{15}, K_{16})] \oplus F(RE_{15}, K_{16}) \\ &= LE_{15} \end{aligned}$$

▦DES (Data Encryption Standard): based on Structure Feistel, 16 rounds, 56-bit encryption key (The key itself is specified with 8 bytes, but one bit of each byte is used as
a parity check)

1. What is specific to DES is the implementation of the F function in the algorithm and how the round keys are derived from the main encryption key.
2. DEA (Data Encryption Algorithm): algorithmic implementation of DES
3. single round of processing in DES:
    1. expansion permutation (E-step): 32-bit right half of the 64-bit input data block is expanded by into a 48-bit block.
        a. divide the 32-bit block into eight 4-bit words

b. attach an additional bit on the left to each 4-bit word (last bit of the previous 4-bit word) (circular)

c. attach an additional bit to the right of each 4-bit word (beginning bit of the next 4-bit word)

Round Number	*Number of left shifts*
1	1
2	1
3	2
4	2
5	2
6	2
7	2
8	2
9	1
10	2
11	2
12	2
13	2
14	2
15	2
16	1

2. 56-bit key is divided into two halves, each half shifted separately, and the combined 56-bit key permuted/contracted to yield a 48-bit round key. To ensure that each bit of the original encryption key is used in roughly 14 of the 16 rounds.
   a. 56-bit encryption key is represented by 8 bytes, with the last bit (the least significant bit) of each byte used as a parity bit.
   b. Encrypt key (Key Permutation 1): Extract the first 7 bits from each of the 8 bytes and permute them in the order of table key_permutation_1
   c. Generate round keys:
      1. At the beginning of each round, we divide the 56 relevant key bits into two 28 bit halves and circularly shift to the left each half by one or two bits, depending on the round
      2. join together the two halves and apply a 56-bit to 48-bit contracting permutation. The resulting 48 bits constitute our round key.

Key Permutation 1						
56	48	40	32	24	16	8
0	57	49	41	33	25	17
9	1	58	50	42	34	26
18	10	2	59	51	43	35
62	54	46	38	30	22	14
6	61	53	45	37	29	21
13	5	60	52	44	36	28
20	12	4	27	19	11	3

Key Permutation 2							
13	16	10	23	0	4	2	27
14	5	20	9	22	18	11	3
25	7	15	6	26	19	12	1
40	51	30	36	46	54	29	39
50	44	32	47	43	48	38	55
33	52	45	41	49	35	28	31

```
key_permutation_1 = [56,48,40,32,24,16,8,0,57,49,41,33,25,17,
                      9,1,58,50,42,34,26,18,10,2,59,51,43,35,
                     62,54,46,38,30,22,14,6,61,53,45,37,29,21,
                     13,5,60,52,44,36,28,20,12,4,27,19,11,3]

key_permutation_2 = [13,16,10,23,0,4,2,27,14,5,20,9,22,18,11,
                      3,25,7,15,6,26,19,12,1,40,51,30,36,46,
                     54,29,39,50,44,32,47,43,48,38,55,33,52,
                     45,41,49,35,28,31]

shifts_for_round_key_gen = [1,1,2,2,2,2,2,2,1,2,2,2,2,2,2,1]

def generate_round_keys(encryption_key):
    round_keys = []
    key = encryption_key.deep_copy()
    for round_count in range(16):
        [LKey, RKey] = key.divide_into_two()
        shift = shifts_for_round_key_gen[round_count]
        LKey << shift
        RKey << shift
        key = LKey + RKey
        round_key = key.permute(key_permutation_2)
        round_keys.append(round_key)
    return round_keys

def get_encryption_key():
    key = ""
    while True:
        if sys.version_info[0] == 3:
            key = input("\nEnter a string of 8 characters for the key: ")
        else:
            key = raw_input("\nEnter a string of 8 characters for the key: ")
        if len(key) != 8:
            print("\nKey generation needs 8 characters exactly.  Try again.\n")
            continue
        else:
            break
    key = BitVector(textstring = key)
    key = key.permute(key_permutation_1)
    return key

encryption_key = get_encryption_key()
round_keys = generate_round_keys(encryption_key)
print("\nHere are the 16 round keys:\n")
for round_key in round_keys:
    print(round_key)
```

   d. The two halves of the encryption key generated in each round are fed as the two halves going into the next round.

3. key mixing: 48 bits of the expanded output produced by the E-step are XORed with the round key.
4. output produced by the previous step is broken into eight six-

bit words. Each 6-bit word fed into a separate S-box. Each S-box produces a 4-bit output. Therefore, the 8 S-boxes together generate a 32-bit output

a. Each of the eight S-boxes consists of a 4*16 table lookup for an output 4-bit word.
b. The first and the last bit of the 6-bit input word are decoded into one of 4 rows
c. the middle 4 bits decoded into one of 16 columns for the table lookup.
d. This step introduce diffusion in the generation of the output from the input. the row lookup for each of the eight S-boxes becomes a function of the input bits for the previous S-box and the next S-box.

```
expansion_permutation = [31,  0,  1,  2,  3,  4,
                          3,  4,  5,  6,  7,  8,
                          7,  8,  9, 10, 11, 12,
                         11, 12, 13, 14, 15, 16,
                         15, 16, 17, 18, 19, 20,
                         19, 20, 21, 22, 23, 24,
                         23, 24, 25, 26, 27, 28,
                         27, 28, 29, 30, 31,  0]

s_boxes = {i:None for i in range(8)}

s_boxes[0] = [ [14,4,13,1,2,15,11,8,3,10,6,12,5,9,0,7],
               [0,15,7,4,14,2,13,1,10,6,12,11,9,5,3,8],
               [4,1,14,8,13,6,2,11,15,12,9,7,3,10,5,0],
               [15,12,8,2,4,9,1,7,5,11,3,14,10,0,6,13] ]

def substitute( expanded_half_block ):
    '''
    This method implements the step "Substitution with 8 S-boxes" step you see inside
    Feistel Function dotted box in Figure 4 of Lecture 3 notes.
    '''
    output = BitVector (size = 32)
    segments = [expanded_half_block[x*6:x*6+6] for x in range(8)]
    for sindex in range(len(segments)):
        row = 2*segments[sindex][0] + segments[sindex][-1]
        column = int(segments[sindex][1:-1])
        output[sindex*4:sindex*4+4] = BitVector(intVal = s_boxes[sindex][row][column], size = 4)
    return output
```

```
# For the purpose of this illustration, let's just make up the right-half of a
# 64-bit DES bit block:
right_half_32bits = BitVector( intVal = 800000700, size = 32 )

# Now we need to expand the 32-bit block into 48 bits:
right_half_with_expansion_permutation = right_half_32bits.permute( expansion_permutation )

print("expanded right_half_32bits: %s" % str(right_half_with_expansion_permutation))

# The following statement takes the 48 bits back down to 32 bits after carrying
# out S-box based substitutions:
output = substitute(right_half_with_expansion_permutation)
print(output)
```

e. Diffusion: a change in any plaintext bit must propagate out to as many ciphertext bits as possible.
f. creating the different round keys from the main key is meant to introduce confusion into the encryption process.
g. Confusion: the relationship between the encryption key and the ciphertext must be as complex as possible. Each bit of the key must affect as many bits as possible of the output ciphertext block.
h. Diffusion and confusion are the two cornerstones of block cipher design.

5. 32-bits of the previous step then go through a P-box based permutation

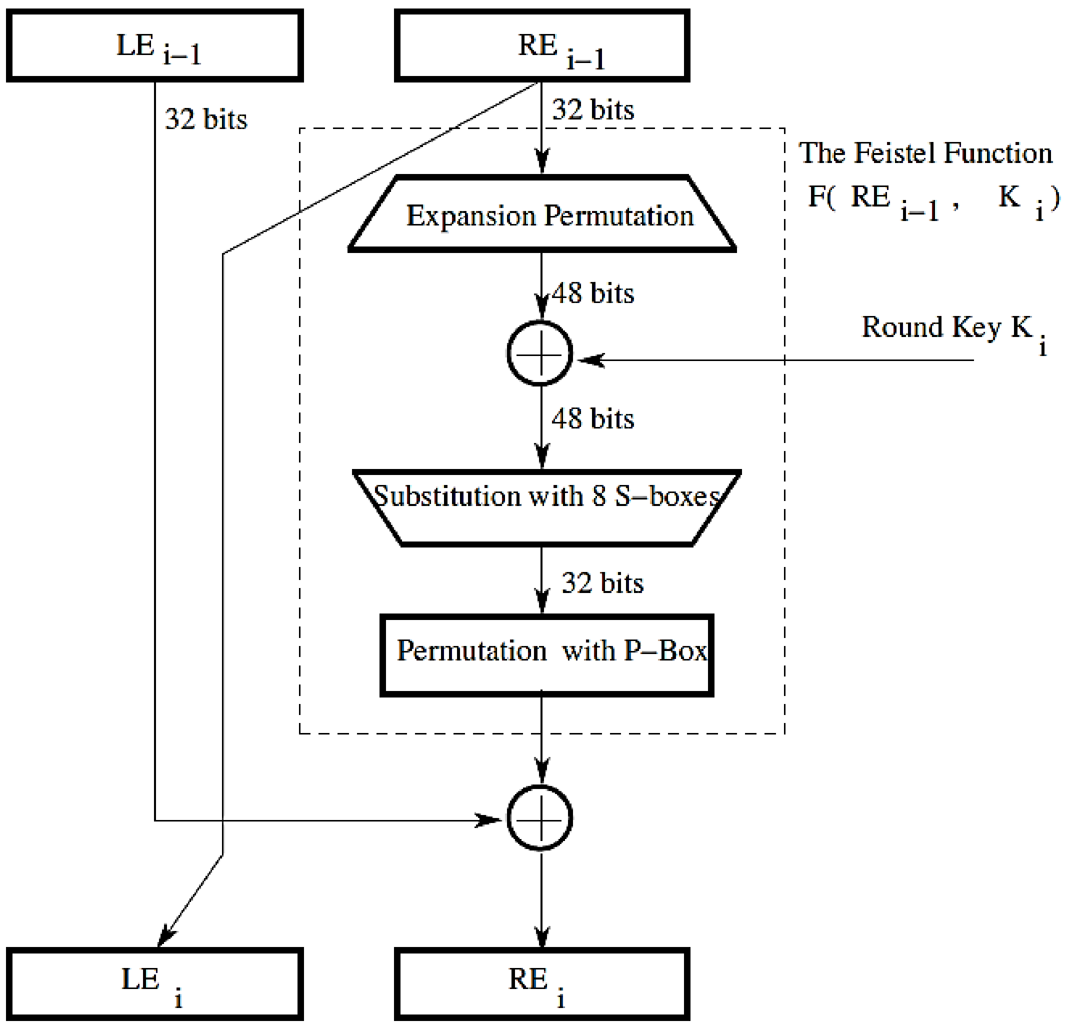
P-Box Permutation							
15	6	19	20	28	11	27	16
0	14	22	25	4	17	30	9
1	7	23	13	31	26	2	8
18	12	29	5	21	10	3	24

a. 0th output bit will be the 15th bit of the input, the 1st output bit will be the 6th bit of the input, and so on, for all of the 32 bits. 0th bit of the second output
byte (the 8th bit of the output) will be the 0th bit of the 32-bit input.

```
sboxes_output =  BitVector representation of the
                        output of the S-Boxes
right_half = sboxes_output.permute( pbox_permutation )
```

6. What comes out of the P-box is then XORed with the left half of the 64-bit block that we started out with. The output of this XORing operation gives us the right half block for the

next round.



4. WHAT MAKES DES A STRONG CIPHER
    1. The substitution step is very effective as far as diffusion is concerned. if you change just one bit of the 64-bit input data block, on the average it propagates out to affect 34 bits of the ciphertext block.
    2. The manner in which the round keys are generated from the encryption key is also very effective as far as confusion is concerned. if you change just one bit of the encryption key, on the average that affects 35 bits of the ciphertext.
    3. Both effects mentioned above are referred to as the avalanche effect.
    4. 56-bit encryption key means a key space of 2^{56}size

▦In the design of the DES, the S-boxes were tuned to enhance the resistance of DES to what is known as the differential cryptanalysis-attack (13)

1. one plaintext bit block X = [X1,X2, ...,Xn] and corresponding output bit block is Y = [Y1, Y2, ..., Yn].
2. Y

▦finite field: a finite set of numbers in which you can carry out the operations of addition, subtraction, multiplication, and division without error. Division is error prone and what you see is a high-precision approximation to the true result.

1. Group:
   - I. A set of objects, along with a binary operation (operation that is applied to two objects at a time) on the elements of the set, must satisfy the following four properties for the set of objects to be called a group:
     - i. Closure: a and b are in the set, then the element a○b = c is also in the set.
     - ii. Associativity: (a○b)○c = a○(b○c)
     - iii. existence of a unique identity element: for every a in the set, a○i= a.
     - iv. existence of an inverse element for each element: for every a in the set, the set must also contain an element b such that a○b= i (i is identity element).
   - II. a group is denoted by {G,○} or {G, + }where G is the set of objects and ○/+ is the group operator. When a group is denoted {G, + } it is common to use the symbol 0 for denoting the group identity element.
   - III. If the Group Operation is Referred tonas Addition, then the Group Also Allows for Subtraction
     - i. $\mathbf{p_1} + \mathbf{p_2} = \mathbf{0}$, then $\mathbf{p_2}$ is the additive inverse of $\mathbf{p_1}$ and denoted as $-\mathbf{p_1}$
     - ii. $\mathbf{p_1} - \mathbf{p_2} = \mathbf{p_1} + (-\mathbf{p_2})$, We may now refer to this expression as representing subtraction.
   - IV. Infinite groups: groups based on sets of infinite size. EX. The set of all integers, the set of all N x N matrices over real numbers under the operation of arithmetic addition, The set of all even integers (odd can't, because odd + odd =even), The set of all 33 nonsingular matrices, along with the matrix multiplication as the operator, denoted GL(3), GL stands for General Linear
   - V. finite groups (Permutation Groups): Pn along with the operation

of composition forms a finite group.

i. $\mathbf{S^n} = <\mathbf{1, 2, \ldots n}>$ denote a sequence of integers 1 through n. the order in which the items appear in a sequence is important. A sequence is typically shown delimited by angle brackets

ii. The set of all permutations of the sequence $\mathbf{S^n}$. Denote this set by $\mathbf{P^n}$. Each element of the set $\mathbf{P^n}$ stands for a permutation $<\mathbf{p_1},\ \mathbf{p_2},\ \mathbf{p_3} \cdots \mathbf{p_n}>$ of the sequence $\mathbf{S_p}$. given a set of n distinct labels, the total number of permutations of the labels is n!. EX. $\mathbf{S_3} = <\mathbf{1, 2, 3}>$, $\mathbf{P_3} = \{<\mathbf{1, 2, 3}>, <\mathbf{1, 3, 2}>, <\mathbf{2, 1, 3}>, <\mathbf{2, 3, 1}>, <\mathbf{3, 1, 2}>, <\mathbf{3, 2, 1}>\}$, cardinality of P3 is 6. $\mathbf{P_3} = \{<\mathbf{p_1},\ \mathbf{p_2},\ \mathbf{p_3}> \mid \mathbf{p_1}, \mathbf{p_2}, \mathbf{p_3} \in \mathbf{S_3}\ \boldsymbol{with}\ \mathbf{p_1} \neq \mathbf{p_2}, \neq \mathbf{p_3}\}$

iii. let the binary operation on the elements of Pn be that of composition of permutations. For any two elements $\boldsymbol{\rho}$ and $\boldsymbol{\pi}$ of the set Pn, the composition means that we want to re-permute the elements of $\boldsymbol{\rho}$ a according to the elements of $\boldsymbol{\pi}$

iv. composition of permutations: $\boldsymbol{\pi} = <\mathbf{3, 2, 1}>, \boldsymbol{\rho} = <\mathbf{1, 3, 2}>$, $\boldsymbol{\pi} \bigcirc \boldsymbol{\rho} = <\mathbf{2, 3, 1}>$. that is accomplished by first choosing the third element of$\boldsymbol{\rho}$, followed by the second element of $\boldsymbol{\rho}$, followed by the first element of $\boldsymbol{\rho}$.

2. abelian group
   I. operation on the set elements is commutative: a○b = b○a
   II. Is the permutation group {Pn. ○} an abelian group? Only when n = 2
   III. Is the set of all integers, positive, negative, and zero, along with the operation of arithmetic addition an abelian group? Yes
3. Ring
   I. define one more operation on an abelian group, we have a ring, provided the elements of the set satisfy some properties with respect to this new operation.
   II. A ring is typically denoted $\{\mathbf{R}, +, \times\}$ where R denotes the set of objects, + the operator with respect to which R is an abelian group, the × the additional operator needed for R to form a ring.
      i. Closure: R must be closed with respect to the additional operator .

ii. Associativity: R must exhibit associativity with respect to the additional operator ×.

iii. Distribution: Distribute over the group addition operator.

1. a × (b + c) = a × b + a × c
2. (a+b) × **c** = a × c + a × c

III. the set of all N×N square matrices over the real numbers under the operations of matrix addition and matrix multiplication constitutes a ring.

IV. The set of all even integers, positive, negative, and zero, under the operations arithmetic addition and multiplication is a ring.

V. The set of all integers under the operations of arithmetic addition and multiplication is a ring.

VI. The set of all real numbers under the operations of arithmetic addition and multiplication is a ring.

4. commutative ring

I. A ring is commutative if the multiplication operation is commutative for all elements in the ring. ab = ba

II. The set of all even integers, positive, negative, and zero, under the operations arithmetic addition and multiplication.

III. The set of all integers under the operations of arithmetic addition and multiplication.

IV. The set of all real numbers under the operations of arithmetic addition and multiplication.

5. integral domain

I. a commutative ring {**R**, +, ×} that obeys the following two additional properties:

i. The set R must include an identity element for the multiplicative operation. a1 = 1a = a

ii. Let 0 denote the identity element for the addition operation (+). If a multiplication of any two elements a and b of R results in 0, then either a or b must be 0.

II. The set of all integers under the operations of arithmetic addition and multiplication.

III. The set of all real numbers under the operations of arithmetic addition and multiplication.

6. Field

I. {**F**, +, ×}, an integral domain whose elements satisfy the following additional property:

i. For every element a in F, except the element 0 (identity element for + operator), there must exist its multiplicative inverse in F.
   $\mathbf{a} \in \mathbf{F}, \mathbf{a} \neq \mathbf{0}$, **such that** $\mathbf{ab} = \mathbf{ba} = \mathbf{1}$ (identity element for $\times$)
ii. field has a multiplicative inverse for every element except the element that serves as the identity element for the group operator.

II. The set of all real numbers under the operations of arithmetic addition and multiplication is a field.
III. The set of all rational numbers under the operations of arithmetic addition and multiplication is a field.
IV. The set of all complex numbers under the operations of complex arithmetic addition and multiplication is a field.
V. The set of all even integers, positive, negative, and zero, under the operations arithmetic addition and multiplication is NOT a field.
VI. The set of all integers under the operations of arithmetic addition and multiplication is NOT a field.

▦a mod n: Given any integer a and a positive integer n, and given a division of a by n that leaves the remainder between 0 and n−1

1. remainder must be between 0 ~ n−1, both ends inclusive, even if we must use a negative quotient when dividing a by n. EX. −8 mod 3 = 1 (quotient = −3)
2. modulo n arithmetic maps all integers into the set {0, 1, 2, 3, ..., n . 1}.

...	0	1	2	0	1	2	0	1	2	0	1	2	0	1	2	0	1	2	0	1	2	0	...
...-	9	-8	-7	-6	-5	-4	-3	-2	-1	0	1	2	3	4	5	6	7	8	9	10	11	12	...

3. congruence : if a mod n = b mod n, a and b is congruent modulo n, expressed as a ≡ b (mod n) or a = b (mod n) or a = b mod n
4. a non-zero integer a is a divisor of another integer b provided the remainder is zero when we divide b by a, expressed as a | b.
5. prove any of the above equalities, you write a as mn + $\mathbf{r}_a$ and b as pn + $\mathbf{r}_b$, where ra and rb are the residues (the same thing as remainders) for a and b

$$
\begin{aligned}
[(a \; mod \; n) \; + \; (b \; mod \; n)] \; mod \; n &= (a \; + \; b) \; mod \; n \\
[(a \; mod \; n) \; - \; (b \; mod \; n)] \; mod \; n &= (a \; - \; b) \; mod \; n \\
[(a \; mod \; n) \; \times \; (b \; mod \; n)] \; mod \; n &= (a \; \times \; b) \; mod \; n
\end{aligned}
$$

6. In mod n arithmetic, any time you see n or any of its multiples (n, 2n, 3n) think them as 0. Anytime you see −1 in mod n arithmetic, you should think n−1.
7. set of residues: Zn is the set of remainders in arithmetic modulo n. Zn = {0, 1, 2, 3, ..., n . 1}
   I. Zn is a group: group operator is modulo n addition
   II. Zn is an abelian group: modulo n addition community (3+ 4)mod3 = (4+ 3) mod 3
   III. Zn is a ring: ring operator is modulo n multiplication, closure, associativity, distribute over addition
   IV. Zn is a commutative ring: modulo n multiplication community
   V. Zn is more than a commutative ring, but not quite an integral domain: Zn possesses a multiplicative identity, but it does NOT satisfy the other condition of integral domains which says that if ab = 0 then either a or b must be zero. Consider modulo 8 arithmetic. 2∗4 = 0, which is a clear violation of the second rule for integral domains
   VI. Zn is not a field:
      i. For every element of Zn, there exists an additive inverse in Zn. But there does not exist a multiplicative inverse for every non-zero element of Zn.

Z_8	:	0	1	2	3	4	5	6	7
additive inverse	:	0	7	6	5	4	3	2	1
multiplicative inverse	:	-	1	-	3	-	5	-	7

      ii. multiplicative inverses exist for only those elements of Zn that are relatively prime to n.
      iii. two integers a and b are relatively prime to each other if Greatest Common Divisor. gcd(a, b) = 1
   VII. Zn has 5 properties:
      i. Commutativity

$$(w + x)\ mod\ n = (x + w)\ mod\ n$$
$$(w \times x)\ mod\ n = (x \times w)\ mod\ n$$

      ii. Associativity

$$[(w + x) + y]\ mod\ n = [w + (x + y)]\ mod\ n$$
$$[(w \times x) \times y]\ mod\ n = [w \times (x \times y)]\ mod\ n$$

iii. Distributivity of Multiplication over Addition

$$[w \times (x + y)] \bmod n = [(w \times x) + (w \times y)] \bmod n$$

iv. Existence of Identity Elements

$$(0 + w) \bmod n = (w + 0) \bmod n$$
$$(1 \times w) \bmod n = (w \times 1) \bmod n$$

v. Existence of Additive Inverses

For each $w \in Z_n$, there exists a $z \in Z_n$ such that

$$w + z = 0 \bmod n$$

VIII. (a + b) ≡ (a + c) (mod n) implies b ≡ c (mod n)

IX. (a b) ≡ (a c) (mod n) does not imply b ≡ c (mod n), unless a and n are relatively prime to each other.

▦Euclids algorithm for GCD

1. Oberservations
   I. gcd(a, a) = a
   II. if b|a then gcd(a, b) = b
   III. gcd(a, 0) = a
   IV. Euclids GCD
2. Algorithm: gcd(a, b) = gcd(b, a mod b)

```
while b:
    a,b = b, a%b

print("\nGCD: %d\n" % a)
```

3. proof of Euclids algorithm
   I. Given any two non-negative integers a and b, with a > b, we can write a = qb + r for some non-negative quotient integer q and some non-negative remainder integer r.
   II. Every common divisor of a and b must therefore be a common divisor of qb + r and b. Since the product qb is trivially divisible by b, it is surely the case that every common divisor of a and b is a common divisor of r and b.
   III. all common divisors for a and b are the same as those for b and r. gcd(a, b) = gcd(b, r).

▦Binary GCD algorithm / Steins algorithm

1. observations
   I. shifting a binary code word to the left by one bit position means multiplication by 2.
   II. Shifting by one bit position to the right means division by 2.
   III. an even integer: whose LSB (least significant bit) is no

2. consider the following five cases
   I. If both the integers a and b are even, then 2 is a common factor of the two integers. So gcd(a, b) = 2 gcd(a/2, b/2). The new arguments a/2 and b/2 are obtained by shifting the binary word representations for each integer to the right by one bit position.
   II. If a is even and b is odd, then gcd(a, b) = gcd(a/2, b). So we shift a to the right by one bit position and call gcd again.
   III. If a is odd and b is even, then gcd(a, b) = gcd(a, b/2). So we shift b to the right by one bit position and call gcd again.
   IV. If both a and b are odd and, at the same time, a > b, then we can show that the gcd recursion takes the following form gcd(a, b) = gcd(a−b, b) = gcd((a−b)/2, b), and back to the second situation
   V. If both a and b are odd and, at the same time, a < b, gcd(a, b) = gcd(b−a, a) = gcd((b−a)/2, a).
3. Implementation

```
a,b = int(sys.argv[1]),int(sys.argv[2])

def bgcd(a,b):
    if a == b: return a                                              #(A)
    if a == 0: return b                                              #(B)
    if b == 0: return a                                              #(C)
    if (~a & 1):                                                     #(D)
        if (b &1):                                                   #(E)
            return bgcd(a >> 1, b)                                   #(F)
        else:                                                        #(G)
            return bgcd(a >> 1, b >> 1) << 1                         #(H)
    if (~b & 1):                                                     #(I)
        return bgcd(a, b >> 1)                                       #(J)
    if (a > b):                                                      #(K)

        return bgcd( (a-b) >> 1, b)                                  #(L)
    return bgcd( (b-a) >> 1, a )                                     #(M)

gcdval = bgcd(a, b)
print("\nBGCD: %d\n" % gcdval)
```

▦PRIME FINITE FIELDS

1. The main reason for why Zn is only a commutative ring and not a finite field is because not every element in Zn is guaranteed to have a multiplicative inverse. an element a of Zn does not have a multiplicative inverse if a is not relatively prime to the modulus n.

2. For prime n, every non-zero element a∈Zn will be relatively prime to n. That implies that there will exist a multiplicative inverse for every non-zero a∈Zn for prime n.
3. prime finite field GF(p) (GF stands for Galois Field): Zp is a finite field if we assume p denotes a prime number.
4. Proving that, for prime p, every non-zero element of Zp possess a unique MI (multiplicative inverse): assume that a non-zero element a∈Zp possesses two different MIs b and c. That would imply ab = 1 (mod p) and ac = 1 (mod p). That would mean that a (b−c) ≡ 0 (mod p) ≡ p (mod p). But that is impossible since the prime number p cannot be so factorized. The integer p only possesses only trivial factors, 1 and itself.
5. a b = 0 for general Zn occurs only when non-zero a and b are factors of the modulus n. When n is a prime, its only factors are 1 and n. So with the elements of Zn being in the range 0 through n.1, the only time we will see a b = 0 is when either a is 0 or b is 0.

▦finding the value of the multiplicative inverse of a given integer a in modulo n arithmetic

1. Bezouts Identity: when a and n are any pair of positive integers, the following must always hold for some integers x and y (that may be positive or negative or zero): gcd(a, n)=x a+ y n. x and y do not have to be unique for given a and n.
2. Proof of Bezouts Identity:
   I. S = {am + bn | am + bn > 0, m, n∈Z }
   II. by its definition, S can only contain positive integers. When a = 8 and b = 6, we have S={2, 4, 6, 8....}
   III. let d denote the smallest element of S
   IV. a=qd+ r, 0 . r < d
   V. expressed the residue r as a linear sum of a and b. But that is only possible if r equals 0. If r is not 0 but actually a non-zero integer less than d that would violate the fact that d is the smallest positive linear sum of a and b.
$$\begin{aligned} r &= a \ mod \ d \\ &= a - qd \\ &= a - q(am + bn) \\ &= a(1 - qm) + b(-n) \end{aligned}$$
   VI. Since r is zero, it must be the case that a = qd for some integer q. Similarly, we can prove that b is sd for some integer s. This

proves that d is a common divisor of a and b.

VII. assume that some other integer c is also a divisor of a and b. Then it must be the case that c is a divisor of all linear combinations of the form ma + nb. Since d is of the form ma + nb, then c must be a divisor of d. This fact applies to any arbitrary common divisor c of a and b. That is, every common divisor c of a and b must also be a divisor of d.

VIII. Hence it must be the case that d is the GCD of a and b.

3. Extended Euclids Algorithm Algorithm
   I. use the same Euclid algorithmas before to find the gcd(a, n),
   II. at each step we write the expression in the form a x + n y for the remainder
   III. before we get to the remainder becoming 0, when the remainder becomes 1 (which will happen only when a and n are relatively prime), x will automatically be the multiplicative inverse we are looking for.

$$\begin{array}{llll|l}
gcd(b_1, b_2) & & & & assume\ \ b_1 > b_2 \\
 & = gcd(b_2, b_1\ mod\ b_2) & = gcd(b_2, b_3) & & b_3 = b_1 - q_1 \times b_2 \\
 & = gcd(b_3, b_2\ mod\ b_3) & = gcd(b_3, b_4) & & b_4 = b_2 - q_2 \times b_3 \\
 & = gcd(b_4, b_3\ mod\ b_4) & = gcd(b_4, b_5) & & b_5 = b_3 - q_3 \times b_4 \\
 & .... & .... & & \\
 & .... & .... & & \\
 & gcd(b_{m-1}, b_m) & & & b_m = b_{m-2} - q_{m-2} \times b_{m-1}
\end{array}$$

until b_m is either 0 or 1.

IV. If bm= 0 and bm exceeds 1, then there does NOT exist a multiplicative inverse for b1 in arithmetic modulo b2.

V. If bm = 1, then there exists a multiplicative inverse for b1 in arithmetic modulo b2.

VI. find the multiplicative inverse of 32 modulo 17, which is 8

```
gcd( 32, 17 )
    = gcd( 17, 15 )     | residue  15  = 1x32 - 1x17
    = gcd( 15, 2 )      | residue   2  = 1x17 - 1x15
                        |              = 1x17 - 1x(1x32 - 1x17)
                        |              = (-1)x32 + 2x17
    = gcd(  2, 1 )      | residue   1  = 1x15 - 7x2
                        |              = 1x(1x32 - 1x17)
                        |                    - 7x( (-1)x32 + 2x17 )
                        |              = 8x32 - 15x17
```

```
#!/usr/bin/env python

## FindMI.py

import sys

if len(sys.argv) != 3:
    sys.stderr.write("Usage: %s   <integer>   <modulus>\n" % sys.argv[0])
    sys.exit(1)

NUM, MOD = int(sys.argv[1]), int(sys.argv[2])

def MI(num, mod):
    '''
    This function uses ordinary integer arithmetic implementation of the
    Extended Euclid's Algorithm to find the MI of the first-arg integer
    vis-a-vis the second-arg integer.
    '''
    NUM = num; MOD = mod
    x, x_old = 0L, 1L
    y, y_old = 1L, 0L

    while mod:
        q = num // mod
        num, mod = mod, num % mod
        x, x_old = x_old - q * x, x
        y, y_old = y_old - q * y, y
    if num != 1:
        print("\nNO MI. However, the GCD of %d and %d is %u\n" % (NUM, MOD, num))
    else:
        MI = (x_old + MOD) % MOD
        print("\nMI of %d modulo %d is: %d\n" % (NUM, MOD, MI))

MI(NUM, MOD)
```

Stein's Algorithm:

$$\begin{aligned} gcd(1344, 752) &= 2 * gcd(672, 376) \\ &= 4 * gcd(336, 188) \\ &= 8 * gcd(168, 94) \\ &= 16 * gcd(84, 47) \\ &= 16 * gcd(42, 47) \\ &= 16 * gcd(21, 47) \\ &= 16 * gcd(13, 21) \\ &= 16 * gcd(4, 13) \\ &= 16 * gcd(2, 13) \\ &= 16 * gcd(1, 13) \\ &= 16 * gcd(6, 1) \\ &= 16 * gcd(3, 1) \\ &= 16 * gcd(1, 1) \\ &= 16 \end{aligned}$$

▦polynomial arithmetic

1. we can represent a bit pattern by a polynomial in the variable x. Each power of x in the polynomial can stand for a bit position in a bit pattern. 101 -> $x^2 + 1$
2. zeroth-degree polynomial is called a constant polynomial.
3. Polynomial arithmetic deals with the addition, subtraction, multiplication, and division of polynomials.
   I. Add

$$\begin{aligned} f(x) &= a_2x^2 + a_1x + a_0 \\ g(x) &= b_1x + b_0 \\ f(x) + g(x) &= a_2x^2 + (a_1 + b_1)x + (a_0 + b_0) \end{aligned}$$

   II. Subtract

$$\begin{aligned} f(x) &= a_2x^2 + a_1x + a_0 \\ g(x) &= b_3x^3 + b_0 \\ f(x) - g(x) &= -b_3x^3 + a_2x^2 + a_1x + (a_0 - b_0) \end{aligned}$$

   III. Multiply

$$\begin{aligned} f(x) &= a_2x^2 + a_1x + a_0 \\ g(x) &= b_1x + b_0 \\ f(x) \times g(x) &= a_2b_1x^3 + (a_2b_0 + a_1b_1)x^2 + (a_1b_0 + a_0b_1)x + a_0b_0 \end{aligned}$$

   IV. Divide: x2 dividend is $8x^2$ + 3x + 2 and the divisor is 2x + 1, quotient is 4x . 0.5, remainder is 2.5

$$\frac{8x^2 + 3x + 2}{2x + 1} = 4x - 0.5 + \frac{2.5}{2x + 1}$$

4. polynomials whose coefficients belong to the finite field Z_7, the

additive and the multiplicative inverses in this set can be displayed as:

Z_7:	0	1	2	3	4	5	6
AI:	0	6	5	4	3	2	1
MI:	—	1	4	5	2	3	6

I. Add: 1+6 mod 7=0

$$\begin{array}{rcl} f(x) & = & 5x^2 + 4x + 6 \\ g(x) & = & 2x + 1 \\ f(x) + g(x) & = & 5x^2 + 6x \end{array}$$

II. Subtract

$$\begin{array}{rcl} f(x) & = & 5x^2 + 4x + 6 \\ g(x) & = & 2x + 1 \\ f(x) - g(x) & = & 5x^2 + 2x + 5 \end{array}$$

III. Multiply: 5*2 mod 7=3

$$\begin{array}{rcl} f(x) & = & 5x^2 + 4x + 6 \\ g(x) & = & 2x + 1 \\ f(x) \times g(x) & = & 3x^3 + 6x^2 + 2x + 6 \end{array}$$

IV. Divide: 2*6 mod 7 =5 or 5/2 MI = 4 -> 5*4=20 -> 20 mod 7 =6

$$\begin{array}{rcl} f(x) & = & 5x^2 + 4x + 6 \\ g(x) & = & 2x + 1 \\ f(x) \,/\, g(x) & = & 6x + 6 \end{array}$$

Dividing 5 by 2 is the same as multiplying 5 by the multiplicative inverse of 2. Multiplicative inverse of 2 is 4 since 2×4 mod 7 is 1. Therefore, the first term of the quotient is 6x.

$$\frac{5}{2} = 5 \times 2^{-1} = 5 \times 4 = 20 \ mod\ 7 = 6$$

Product of 6x and 2x + 1 is **5x^2** + 6x, subtract **5x^2** + 6x from the dividend **5x^2** +4x + 6, result is (4−6)x + 6, which (since the additive inverse of 6 is 1) is the same as (4 + 1)x + 6, and that is the same as 5x + 6.

As a polynomial defined over the field GF(7), **5x^2** +4x + 6 is a product of two factors, 2x + 1 and 6x + 6.

$$5x^2 + 4x + 6 = (2x + 1) \times (6x + 6)$$

5. polynomial over a field: a polynomial is defined over a field if all its coefficients are drawn from the field.
6. POLYNOMIALS OVER A FIELD CONSTITUTE A RING
   I. The group operator is polynomial addition,
   II. The polynomial 0 is obviously the identity element with

respect to polynomial addition.
   - III. Polynomial addition is associative and commutative.
   - IV. The set of all polynomials over a given field is closed under polynomial addition.
   - V. Polynomial multiplication distributes over polynomial addition.
   - VI. polynomial multiplication is associative.
   - VII. the set of all polynomials over a field constitutes a polynomial ring.
   - VIII. polynomial multiplication is commutative, the set of polynomials over a field is actually a commutative ring.
   - IX. it does not make sense to talk about multiplicative inverses of polynomials in the set of all possible polynomials that can be defined over a finite field. (Recall that our polynomials do not contain negative powers of x.)
   - X. it is possible for a finite set of polynomials, whose coefficients are drawn from a finite field, to constitute a finite field.
7. Polynomial division is obviously not allowed for polynomials that are not defined over fields. You cannot divide **$4x^2$+** 5 by the polynomial 5x. If you tried, the first term of the quotient would be (4/5)x where the coefficient of x is not an integer.
8. You can always divide polynomials defined over a field. Operation of division is legal when the coefficients are drawn from a finite field.
9. for polynomials defined over a field, the division of a polynomial f(x) of degree m by another polynomial g(x) of degree n . m can be expressed by where q(x) is the quotient and r(x) the remainder.
$$\frac{f(x)}{g(x)} = q(x) + \frac{r(x)}{g(x)}$$
$$f(x) = q(x)g(x) + r(x)$$
When r(x) is zero, we say that g(x) divides f(x), g(x) is a divisor of f(x), g(x)|f(x). When g(x) divides f(x) without leaving a remainder, we say g(x) is a factor of f(x).
10. prime polynomial / irreducible polynomial: A polynomial f(x) over a field F, if f(x) cannot be expressed as a product of two polynomials, both over F and both of degree lower than that of f(x).
11. 2 is the first prime. For a number to be prime, it must have exactly two distinct divisors, 1 and itself. GF(2) consists of the set {0, 1}. The two elements of this set obey the following addition and

multiplication rules: (-1 is equivalent to +1)

```
0  +  0  =  0      0  -  0  =  0                  0  X  0  =  0
0  +  1  =  1      1  -  0  =  1                  0  X  1  =  0
1  +  0  =  1      0  -  1  =  0  +  1  =  1      1  X  0  =  0
1  +  1  =  0      1  -  1  =  1  +  1  =  0      1  X  1  =  1
```

addition over GF(2) is equivalent to the logical XOR operation, and multiplication to the logical AND operation.

I. Adding

$$\begin{aligned} f(x) &= x^2 + x + 1 \\ g(x) &= x + 1 \\ f(x) + g(x) &= x^2 \end{aligned}$$

II. Subtracting

$$\begin{aligned} f(x) &= x^2 + x + 1 \\ g(x) &= x + 1 \\ f(x) - g(x) &= x^2 \end{aligned}$$

III. Multiplying

$$\begin{aligned} f(x) &= x^2 + x + 1 \\ g(x) &= x + 1 \\ f(x) \times g(x) &= x^3 + 1 \end{aligned}$$

IV. Dividing

$$\begin{aligned} f(x) &= x^2 + x + 1 \\ g(x) &= x + 1 \\ f(x) / g(x) &= x + \frac{1}{x + 1} \end{aligned}$$

V. GF(2) is a finite field consisting of the set {0, 1}, with modulo 2 addition as the group operator and modulo 2 multiplication as the ring operator.

VI. [Does $23x^5+ 1$ belong to the set of polynomials defined over GF(2)? How about $-3x^7+ 1$? The answer to both questions is yes.

VII. the number of such polynomials is infinite.

VIII. there exist only two irreducible polynomials of degree 3 over GF(2), $\boldsymbol{x^3 + x + 1}$ ***and*** $\boldsymbol{x^3 + x^2 + 1}$.

IX. polynomial arithmetic modulo the irreducible polynomial: when polynomial multiplication results in a polynomial whose degree equals or exceeds that of the irreducible polynomial, we will take for our result the remainder modulo the irreducible polynomial.

$$(x^2 + x + 1) \times (x^2 + 1) \ mod \ (x^3 + x + 1)$$

$$\begin{aligned} &= (x^4 + x^3 + x^2) + (x^2 + x + 1) \ mod \ (x^3 + x + 1) \\ &= (x^4 + x^3 + x + 1) \ mod \ (x^3 + x + 1) \\ &= -x^2 - x \\ &= x^2 + x \end{aligned}$$

$$\frac{(x^4 + x^3 + x + 1)}{(x^3 + x + 1)} = x + 1 + \frac{-x^2 - x}{x^3 + x + 1}$$

8. GF(2^3): 3 is the degree of the modulus polynomial. GF(2^3) maps all of the polynomials over GF(2) to the eight polynomials shown above.
   I. each power of x representing a specific position in a bit string.
   II. With multiplications modulo $x^3 + x + 1$, we have only the following eight polynomials in the set of polynomials over GF(2): the highest degree of the remainder is always 1 less of the irreducible polynomial

$$\begin{aligned} &0 \\ &1 \\ &x \\ &x^2 \\ &x + 1 \\ &x^2 + 1 \\ &x^2 + x \\ &x^2 + x + 1 \end{aligned}$$

   III. the crucial difference between GF(2^3) and Z_8 is that GF(2^3) is a field, whereas Z_8 is NOT.
   IV. GF(2^3) is an abelian group because of the operation of polynomial addition satisfies all of the requirements on a group operator and because polynomial addition is commutative. [Every polynomial in GF(2^3) is its own additive inverse because of how the two numbers in GF(2) behave addition with respect to modulo 2.]
   V. GF(2^3) is also a commutative ring because polynomial multiplication distributes over polynomial addition (and because polynomial ultiplication meets all the other stipulations on the ring operator: closedness, associativity, commutativity).
   VI. GF(2^3) is an integral domain because of the fact that the set contains the multiplicative identity element 1 and because if for a $\in$GF(2^3), b $\in$GF(2^3), a$\times$ $\mathbf{b} = \mathbf{0}$ **mod** $(\boldsymbol{x^3 + x + 1})$ then either a=0 or b=0. Proof:
      i. Assume that neither a nor b is zero when a$\times$ $\mathbf{b} =$ $\mathbf{0}$ **mod** $(\boldsymbol{x^3 + x + 1})$
      ii. Then a$\times$ $\mathbf{b} = (\boldsymbol{x^3 + x + 1})$
      iii. Then 0 $\equiv (\boldsymbol{x^3 + x + 1})$ mod $(\boldsymbol{x^3 + x + 1})$
      iv. above implies that the irreducible polynomial $\boldsymbol{x^3 + x + 1}$ can be factorized, which by definition cannot be done.

VII. GF($\mathbf{2^3}$) is a finite field because it is a finite set and because it contains a unique multiplicative inverse for every non-zero element.

  i. for every non-zero element a∈GF($\mathbf{2^3}$) there is always a unique element b ∈GF($\mathbf{2^3}$) such that a× **b** = 1.

  ii. if you multiply a non-zero element a with each of the eight elements of GF($\mathbf{2^3}$), the result will the eight distinct elements of GF($\mathbf{2^3}$). The results of such multiplications must equal 1 for exactly one of the non-zero element of GF($\mathbf{2^3}$). So if a× **b** = 1, then b must be the multiplicative inverse for a.

  iii. if you multiply a non-zero element a of GF($\mathbf{2^3}$) with every element of the same set, no two answers will be the same.
  Proof:
    1. assume the existence of two distinct b and c in the set such that
    $$\mathbf{a \times b \equiv a \times c \bmod (x^3 + x + 1)}$$
    2. that implies $\mathbf{a \times (b - c) \equiv 0 \bmod (x^3 + x + 1)}$
    3. that implies either a is 0 or that b equals c. In either case, we have a contradiction.

▦GF($\mathbf{2^n}$) IS A FINITE FIELD FOR EVERY n

1. GF($\boldsymbol{P^n}$) is a finite field for any prime p. The elements of GF($\boldsymbol{P^n}$) are polynomials over GF(p) (which is the same as the set of residues Zp)
2. Given any n at all, exactly the same approach can be used to $\mathbf{2^n}$ come up with bit patterns, each pattern consisting of n bits, for a set of integers that would constitute a finite field, provided we have available to us an irreducible polynomial of degree n.
3. The order of a finite field refers to the number of elements in the field. So the order GF($\mathbf{2^n}$) is $\mathbf{2^n}$
4. polynomial coefficients in GF($\mathbf{2^n}$) must obey the arithmetic rules that apply to GF(2) (which is the same as $\boldsymbol{Z_2}$, the set of remainders modulo 2).
   I. GF($\mathbf{2^n}$) adding the bit patterns in simply amounts to taking the bitwise XOR of the bit patterns.
   5 + 13 = 0000 0101 + 0000 1101 = 0000 1000 = 8
   76 + 22 = 0100 1100 + 0001 0110 = 0101 1010 = 90
   II. subtracting is the same as adding in GF($\mathbf{2^8}$), because each

number in GF(2^8) is its own additive inverse. For every x∈GF(2^8) we have .x = −x or x+ x=0.

$$7 - 3 = 0000\,0111 - 0000\,0011 = 0000\,0100 = 4$$
$$7 + 3 = 0000\,0111 + 0000\,0011 = 0000\,0100 = 4$$

III. the bitwise operations needed for directly multiplying two bit patterns in GF(2n) are specific to the irreducible polynomial that defines a given GF(2n). irreducible polynomial of degree 8:

$$m(x) = x^8 + x^4 + x^3 + x + 1$$
$$x^8 \ mod \ m(x) = x^4 + x^3 + x + 1$$
$$f(x) = b_7x^7 + b_6x^6 + b_5x^5 + b_4x^4 + b_3x^3 + b_2x^2 + b_1x + b_0$$

stands for the bit pattern b7b6b5b4b3b2b1b0.

$$f(x)\times x = b_7x^8 + b_6x^7 + b_5x^6 + b_4x^5 + b_3x^4 + b_2x^3 + b_1x^2 + b_0x$$

if b7 equals 1,

$$(f(x) \times x) \ mod \ m(x)$$

$$\begin{aligned}
&= (b_7x^8 + b_6x^7 + b_5x^6 + b_4x^5 + b_3x^4 + b_2x^3 + b_1x^2 + b_0x) \ mod \ m(x) \\
&= (b_6x^7 + b_5x^6 + b_4x^5 + b_3x^4 + b_2x^3 + b_1x^2 + b_0x) \ + \ (x^8 \ mod \ m(x)) \\
&= (b_6x^7 + b_5x^6 + b_4x^5 + b_3x^4 + b_2x^3 + b_1x^2 + b_0x) + (x^4 + x^3 + x \ + 1) \\
&= (b_6b_5b_4b_3b_2b_1b_00) \quad \otimes \quad (00011011)
\end{aligned}$$

If the bit b7 of f(x) is equals 0, then the right hand above is already in the set of polynomials in GF(28), the output bit pattern is b6b5b4b3b2b1b0

multiply two bit patterns B1 and B2, each 8 bits long

i. If B2 is the bit pattern 00000001, The result is B1 itself.
ii. If B2 is the bit pattern 00000010, then we are multiplying B1 by x. If B1's MSB is 0, the result is obtained by shifting the B1 bit pattern to the left by one bit and inserting a 0 bit from the right. If B1's MSB is 1, first we again shift the B1 bit pattern to the left, then we take the XOR of the shifted pattern with the bit pattern 00011011 for the final answer.
iii. If B2 is the bit pattern 00000100, multiplying B1 by $\boldsymbol{x^2}$. first multiplying B1 by x, and then multiplying B1 the result

again by x.

iv. if B2 is 10000011, we can write

$$B_1 \times 10000011$$

$$\begin{aligned} &= B_1 \times (00000001 + 00000010 + 10000000) \\ &= (B_1 \times 00000001) + (B_1 \times 00000010) + (B_1 \times 10000000) \\ &= (B_1 \times 00000001) \otimes (B_1 \times 00000010) \otimes (B_1 \times 10000000) \end{aligned}$$

IV. Dividing a bit pattern B1 by the B2 would mean multiplying B1 by the multiplicative inverse of B2

	Additive Inverse	Multiplicative Inverse
000	000	-----
001	001	001
010	010	101
011	011	110
100	100	111
101	101	010
110	110	011
111	111	100

```
##   GF_Arithmetic.py
##   Author: Avi Kak
##   Date:   February 13, 2011

##   Note: The code you see in this file has already been incorporated in
##         Version 2.1 and above of the BitVector module.  If you like
##         object-oriented approach to scripting, just use that module
##         directly.  The documentation in that module shows how to make
##         function calls for doing GF(2^n) arithmetic.

from BitVector import *

def gf_divide(num, mod, n):
    '''
    Using the arithmetic of the Galois Field GF(2^n), this function divides
    the bit pattern 'num' by the modulus bit pattern 'mod'
    '''
    if mod.length() > n+1:
        raise ValueError("Modulus bit pattern too long")
    quotient = BitVector( intVal = 0, size = num.length() )
    remainder = num.deep_copy()
    i = 0
    while 1:
        i = i+1
        if (i==num.length()): break
        mod_highest_power = mod.length() - mod.next_set_bit(0) - 1
        if remainder.next_set_bit(0) == -1:
            remainder_highest_power = 0
        else:
            remainder_highest_power = remainder.length() \
                                      - remainder.next_set_bit(0) - 1
        if (remainder_highest_power < mod_highest_power) \
              or int(remainder)==0:
            break
        else:
            exponent_shift = remainder_highest_power - mod_highest_power
            quotient[quotient.length() - exponent_shift - 1] = 1
            quotient_mod_product = mod.deep_copy();
            quotient_mod_product.pad_from_left(remainder.length() - \
                                               mod.length() )
            quotient_mod_product.shift_left(exponent_shift)
            remainder = remainder ^ quotient_mod_product
    if remainder.length() > n:
        remainder = remainder[remainder.length()-n:]
    return quotient, remainder

def gf_multiply(a, b):
    '''
    Using the arithmetic of the Galois Field GF(2^n), this function multiplies
    the bit pattern 'a' by the bit pattern 'b'.
    '''
    a_highest_power = a.length() - a.next_set_bit(0) - 1
    b_highest_power = b.length() - b.next_set_bit(0) - 1
    result = BitVector( size = a.length()+b.length() )
    a.pad_from_left( result.length() - a.length() )
    b.pad_from_left( result.length() - b.length() )
    for i,bit in enumerate(b):
        if bit == 1:
            power = b.length() - i - 1
            a_copy = a.deep_copy()
            a_copy.shift_left( power )
            result ^=  a_copy
    return result

def gf_multiply_modular(a, b, mod, n):
    '''
    Using the arithmetic of the Galois Field GF(2^n), this function returns 'a'
    divided by 'b' modulo the bit pattern in 'mod'.
    '''
    a_copy = a.deep_copy()
    b_copy = b.deep_copy()
    product = gf_multiply(a_copy,b_copy)
    quotient, remainder = gf_divide(product, mod, n)
    return remainder

def gf_MI(num, mod, n):
    '''
    Using the arithmetic of the Galois Field GF(2^n), this function returns the
    multiplicative inverse of the bit pattern 'num' when the modulus polynomial
    is represented by the bit pattern 'mod'.
    '''
    NUM = num.deep_copy(); MOD = mod.deep_copy()
    x = BitVector( size=mod.length() )
    x_old = BitVector( intVal=1, size=mod.length() )
    y = BitVector( intVal=1, size=mod.length() )
    y_old = BitVector( size=mod.length() )
    while int(mod):
        quotient, remainder = gf_divide(num, mod, n)
        num, mod = mod, remainder
        x, x_old = x_old ^ gf_multiply(quotient, x), x
        y, y_old = y_old ^ gf_multiply(quotient, y), y
    if int(num) != 1:
            return "NO MI. However, the GCD of ", str(NUM), " and ", \
                                       str(MOD), " is ", str(num)
    else:
        quotient, remainder = gf_divide(x_old ^ MOD, MOD, n)
        return remainder


mod = BitVector( bitstring = '100011011' )           # AES modulus

a = BitVector( bitstring = '10000000' )
result = gf_MI( a, mod, 8 )
print("\nMI of %s is: %s" % (str(a), str(result)))
```

```
a = BitVector( bitstring = '10010101' )
result = gf_MI( a, mod, 8 )
print("\nMI of %s is: %s" % (str(a), str(result)))

a = BitVector( bitstring = '00000000' )
result = gf_MI( a, mod, 8 )
print("\nMI of %s is: %s" % (str(a), str(result)))
```

When you run the above script, it returns the following result:

```
MI of 10000000 is: 10000011

MI of 10010101 is: 10001010

MI of 00000000 is: ('NO MI. However, the GCD of ', '00000000', ' and ', '100011011', ' is ', '100011011')
```

▦AES

1. AES is a block cipher with a block length of 128 bits.
2. AES allows for three different key lengths: 128, 192, or 256 bits. Encryption consists of 10 rounds of processing for 128-bit keys, 12 rounds for 192-bit keys, and 14 rounds for 256-bit keys.
3. Each round of processing includes one single-byte based substitution step, a row-wise permutation step, a column-wise mixing step, and the addition of the round key.
4. State array: think of a 128-bit block as consisting of a 4*4 array of bytes

$$\begin{bmatrix} byte_0 & byte_4 & byte_8 & byte_{12} \\ byte_1 & byte_5 & byte_9 & byte_{13} \\ byte_2 & byte_6 & byte_{10} & byte_{14} \\ byte_3 & byte_7 & byte_{11} & byte_{15} \end{bmatrix}$$

5. A word consists of four bytes, that is 32 bits. Therefore, each column of the state array is a word, as is each row.
6. Each round of processing works on the input state array and produces an output state array. The output state array produced by the last round is rearranged into a 128-bit output block.
7. DES was based on the Feistel network. AES uses a substitution-permutation network. Each round of processing in AES involves byte-level substitutions followed by word-level permutations. DES is a bit-oriented cipher, AES is a byte-oriented cipher.
8. Like DES, AES is an iterated block cipher in which plaintext is subject to multiple rounds of processing, with each round applying the same overall transformation function to the incoming block.
9. Unlike DES, AES is key-alternating block ciphers. In such ciphers,

each round first applies a diffusion-achieving transformation operation which may be a combination of linear and nonlinear steps to the entire incoming block, which is then followed by the application of the round key to the entire block. DES is based on the Feistel structure in which, for each round, one-half of the block passes through unchanged and the other half goes through a transformation that depends on the S-boxes and the round key. Key alternating ciphers lend themselves well to theoretical analysis of the security of the ciphers.

10. for the 128-bit key AES, the worst case time complexity for a brute-force attack would be $\mathbf{2^{128}}$. Such a brute-force attack would be considered to be an example of a theoretical attack since it is beyond the realm of any practical implementation. biclique attack improves upon this time complexity to $\mathbf{2^{126}}$
11. AES was designed using the wide-trail strategy (dispersal of the probabilities that one can associate with the bits at certain specific positions in a bit block as it propagates through the rounds): a block cipher involves: (1) A local nonlinear transformation (supplied by the substitution step in AES); and (2) A linear mixing transformation that provides high diffusion.
12. In DES, one bit of plaintext affected roughly 31 bits of ciphertext. But now we want each bit of the plaintext to affect every bit position of the ciphertext block of 128 bits (does NOT say that if you change one bit of the plaintext, the algorithm is guaranteed to change every bit of the ciphertext. Since a bit can take on only two values, on the average there will be many bits of the ciphertext that will be identical to the plaintext bits in the same positions after you have changed one bit of the plaintext.)
13. ENCRYPTION KEY AND ITS EXPANSION: Assuming a 128-bit key, the four column words of the key array are expanded into a schedule of 44 words. Each round consumes four words from the key schedule. The first four words are used for adding to the input state array before any round-based processing can begin, and the remaining 40 words used for the ten rounds of processing that are required for the case a 128-bit encryption key.

$$\begin{bmatrix} k_0 & k_4 & k_8 & k_{12} \\ k_1 & k_5 & k_9 & k_{13} \\ k_2 & k_6 & k_{10} & k_{14} \\ k_3 & k_7 & k_{11} & k_{15} \end{bmatrix}$$
$$\Downarrow$$
$$[\, w_0 \quad w_1 \quad w_2 \quad w_3 \,]$$

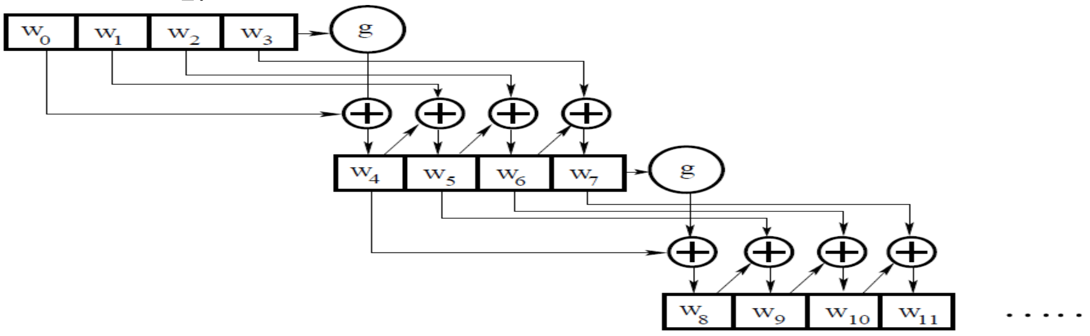
    I. AES Key Expansion: the logic of the key expansion algorithm is designed to ensure that if you change one bit of the encryption key, it should affect the round keys for several rounds.

i. arranges the 16 bytes of the encryption key in the form of a 4*4 array

ii. expands the words [w0,w1,w2,w3] into a 44-word key schedule, takes place on a four-word to four-word basis, each grouping of four words decides what the next grouping of four words will be.

$$w_{i+4} = w_i \otimes g(w_{i+3})$$

1.



2. g() consists of the following three steps:
   I. Perform a one-byte left circular rotation on the argument 4-byte word.
   II. Perform a byte substitution for each byte of the word returned by the previous step by using the same 16 16 lookup table as used in the SubBytes step of the encryption rounds.
   III. XOR the bytes obtained from the previous step with a round constant whose three rightmost bytes are always zero. Therefore, XORing with the round constant amounts to XORing with just its leftmost byte.
3. The only non-zero byte in the round constants, RC[i], obeys the following recursion: multiplication by 0x02 amounts to multiplying the polynomial corresponding to the bit pattern RC[j-1] by x. Addition of the round constants is for the purpose of destroying any symmetries that may have been introduced by the other steps in the key expansion algorithm.

$$\begin{aligned} RC[1] &= 0x01 \\ RC[j] &= 0x02 \times RC[j-1] \end{aligned}$$

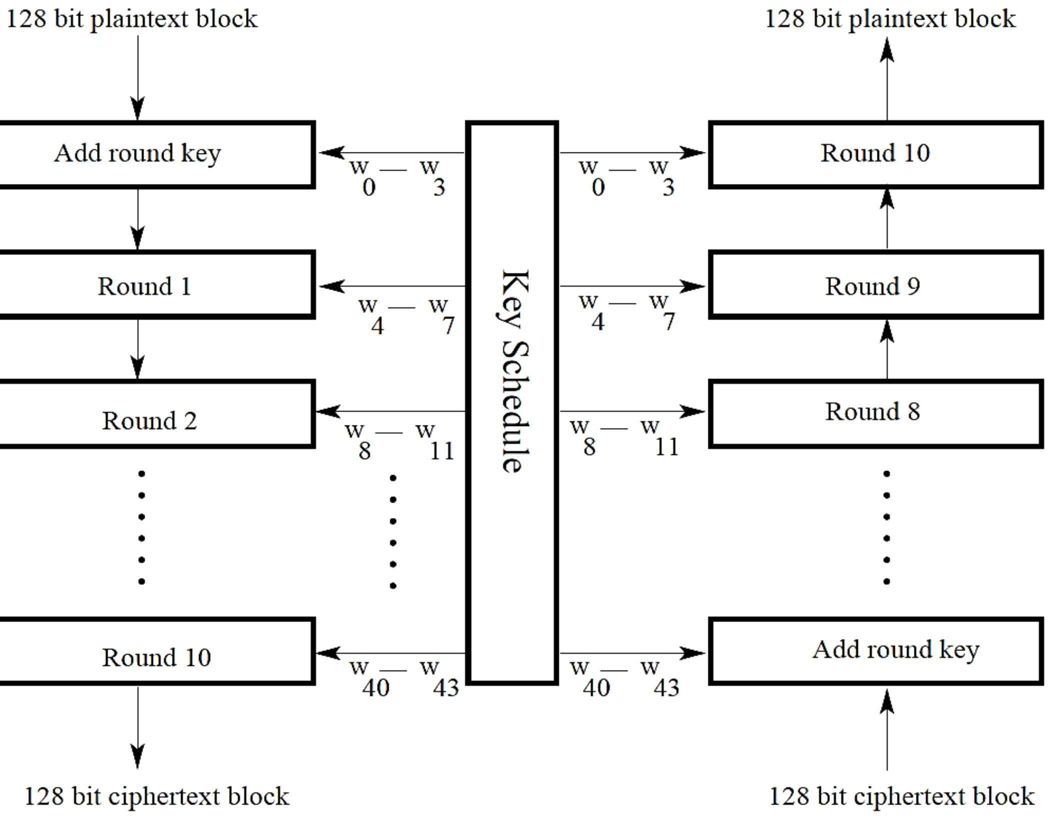
iii. words [w0,w1,w2,w3] are bitwise XORed with the input block before the round-based processing begins.

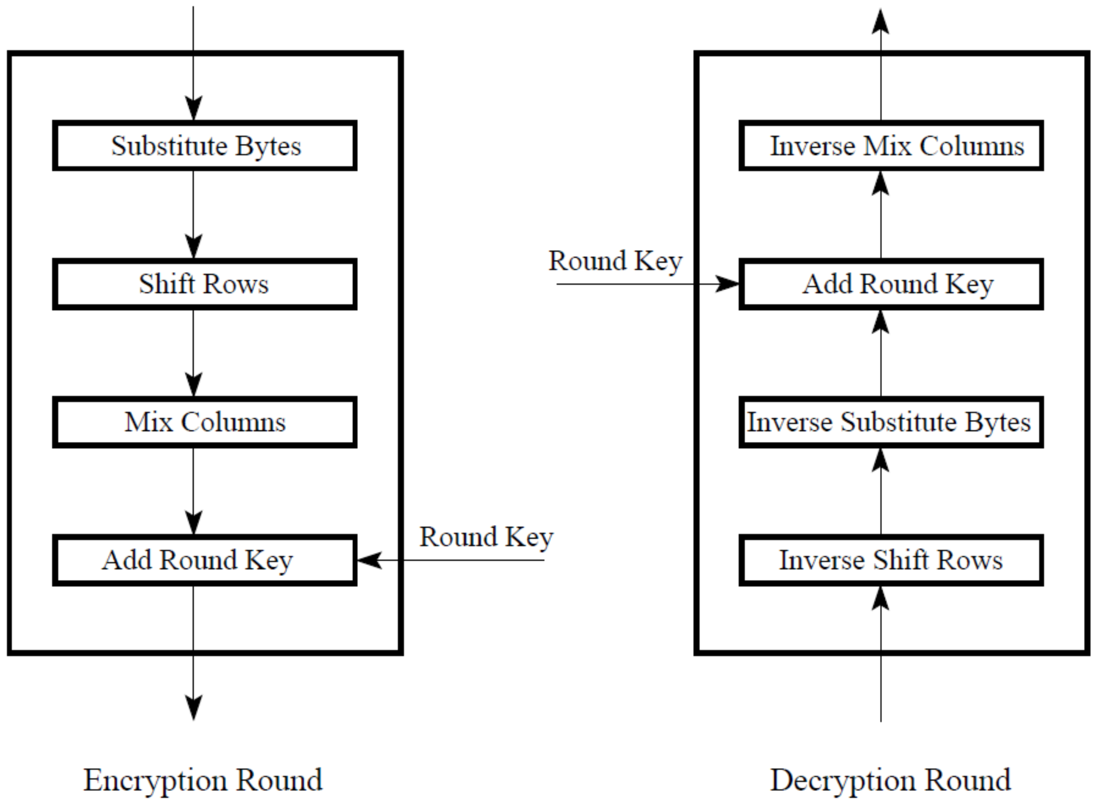
iv. remaining 40 words of the key schedule are used four words at a time in each of the 10 rounds.

II. key expansion algorithm ensures that AES has no weak keys. A weak key is a key that reduces the security of a cipher in a predictable manner. Weak keys of DES are those that produce identical round keys for each of the 16 rounds. Alternating ones and zeros causes the encryption to become self-inverting (plain text encrypted and then encrypted again will lead back to the same plain text).

14. AES structure

I. Before any round-based processing for encryption can begin, the input state array is XORed with the first four words of the key schedule. The same thing happens during decryption except that now we XOR the ciphertext state array with the last four words of the key schedule.





II. For encryption, each round consists of the following four steps:1) Substitute bytes, 2) Shift rows, 3) Mix columns, and 4) Add round key. The last step consists of XORing the output of the previous three steps with four words from the key schedule.

i. Substitute bytes (SubBytes): using a 16*16 lookup table (S-Box) to find a replacement byte for a given byte in the input state array. Entries in the lookup table are created by using the notions of multiplicative inverses in GF(2^8) and bit scrambling reduce the correlation between the input bits and the output bits at the byte level The bit scrambling part of the substitution step ensures that the substitution cannot be described in the form of evaluating a simple mathematical function.. let xin be a byte of the state array for which we seek a substitute byte xout, xout= f(xin):

1. Modern Explanation:
   I. find the multiplicative inverse x′= x_{in}^{-1} in GF(2^8)
   II. scramble the bits of x′by XORing x′with four different circularly rotated versions of itself and with a special constant byte c = 0x63. The four circular rotations are through 4, 5, 6, and 7 bit

positions to the right.

III. role played by the c byte of value 0x63:

i. for the byte substitution step to be invertible, the byte-to-byte mapping given to us by the 16 16 table must be one-one.

ii. No input byte should map to itself, since a byte mapping to itself would weaken the cipher. (byte 0x00 unchanged.)

2. Traditional Explanation

I. fill each cell of the 16*16 table with the byte obtained by joining together its row index and the column index.

	0	1	2	3	4	5	6	7	8	9 ....
0	00	01	02	03	04	05	06	07	08	09 ....
1	10	11	12	13	14	15	16	17	18	19 ....
2	20	21	22	23	24	25	26	27	28	29 ....
	....									

II. replace the value in each cell by its multiplicative inverse GF(2^8) based on the irreducible polynomial x8+ x4+ x3+ x+ 1. The hex value 0x00 is replaced by itself. the byte stored in the cell (9, 5) of the above table is the multiplicative inverse (MI) of 0x95, which is 0x8A

III. apply the following transformation to each bit bi of the byte stored in a cell of the lookup table. All of the additions in the product of the matrix and the vector are actually XOR operations. [Because $[A]\vec{x}+\vec{b}$ is affine transformation.]

$$b'_i = b_i \otimes b_{(i+4)\ mod\ 8} \otimes b_{(i+5)\ mod\ 8} \otimes b_{(i+6)\ mod\ 8} \otimes b_{(i+7)\ mod\ 8} \otimes c_i$$

$$\begin{bmatrix} 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \\ b_4 \\ b_5 \\ b_6 \\ b_7 \end{bmatrix} \otimes \begin{bmatrix} 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \\ 0 \end{bmatrix}$$

ii. Shift rows (ShiftRows): scramble byte order inside each 128-bit block

(i) not shifting the first row of the state array; (ii) circularly shifting the second row by one byte to the left; (iii) circularly shifting the third row by two bytes to the left; and (iv) circularly shifting the last row by three bytes to the left.

$$\begin{bmatrix} s_{0.0} & s_{0,1} & s_{0,2} & s_{0,3} \\ s_{1.0} & s_{1,1} & s_{1,2} & s_{1,3} \\ s_{2.0} & s_{2,1} & s_{2,2} & s_{2,3} \\ s_{3.0} & s_{3,1} & s_{3,2} & s_{3,3} \end{bmatrix} ===> \begin{bmatrix} s_{0.0} & s_{0,1} & s_{0,2} & s_{0,3} \\ s_{1.1} & s_{1,2} & s_{1,3} & s_{1,0} \\ s_{2.2} & s_{2,3} & s_{2,0} & s_{2,1} \\ s_{3.3} & s_{3,0} & s_{3,1} & s_{3,2} \end{bmatrix}$$

iii. Mix columns (MixColumns): replaces each byte of a column by a function of all the bytes in the same column. Shift-rows step along with the mix-column step causes each bit of the ciphertext to depend on every bit of the plaintext after 10 rounds of processing. Each byte in a column is replaced by two times that byte, plus three times the next byte, plus the byte that comes next, plus the byte that follows. For the bytes in 1st, 2nd,3rd,4th row

$$s'_{0,j} = (0x02 \times s_{0,j}) \otimes (0x03 \times s_{1,j}) \otimes s_{2,j} \otimes s_{3,j}$$

$$s'_{1,j} = s_{0,j} \otimes (0x02 \times s_{1,j}) \otimes (0x03 \times s_{2,j}) \otimes s_{3,j}$$

$$s'_{2,j} = s_{0,j} \otimes s_{1,j} \otimes (0x02 \times s_{2,j}) \otimes (0x03 \times s_{3,j})$$

$$s'_{3,j} = (0x03 \times s_{0,j}) \otimes s_{1,j} \otimes s_{2,j} \otimes (0x02 \times s_{3,j})$$

$$\begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix} \times \begin{bmatrix} s_{0.0} & s_{0,1} & s_{0,2} & s_{0,3} \\ s_{1.0} & s_{1,1} & s_{1,2} & s_{1,3} \\ s_{2.0} & s_{2,1} & s_{2,2} & s_{2,3} \\ s_{3.0} & s_{3,1} & s_{3,2} & s_{3,3} \end{bmatrix} = \begin{bmatrix} s'_{0.0} & s'_{0,1} & s'_{0,2} & s'_{0,3} \\ s'_{1.0} & s'_{1,1} & s'_{1,2} & s'_{1,3} \\ s'_{2.0} & s'_{2,1} & s'_{2,2} & s'_{2,3} \\ s'_{3.0} & s'_{3,1} & s'_{3,2} & s'_{3,3} \end{bmatrix}$$

iv. Add round key (AddRoundKey)

III. For decryption, each round consists of the following four steps: 1) Inverse shift rows, 2) Inverse substitute bytes, 3) Add round key, and 4) Inverse mix columns. The third step consists of XORing the output of the previous two steps with four words from the key schedule.

i. Inverse shift rows (InvShiftRows): The first row is left unchanged, the second row is shifted to the right by one byte, the third row to the right by two bytes, and the last row to the right by three bytes, all shifts being circular.

$$\begin{bmatrix} s_{0.0} & s_{0,1} & s_{0,2} & s_{0,3} \\ s_{1.0} & s_{1,1} & s_{1,2} & s_{1,3} \\ s_{2.0} & s_{2,1} & s_{2,2} & s_{2,3} \\ s_{3.0} & s_{3,1} & s_{3,2} & s_{3,3} \end{bmatrix} ===> \begin{bmatrix} s_{0.0} & s_{0,1} & s_{0,2} & s_{0,3} \\ s_{1.3} & s_{1,0} & s_{1,1} & s_{1,2} \\ s_{2.2} & s_{2,3} & s_{2,0} & s_{2,1} \\ s_{3.1} & s_{3,2} & s_{3,3} & s_{3,0} \end{bmatrix}$$

ii. Inverse substitute bytes (InvSubBytes)
   1. scramble the bits of x by XORing xwith four different circularly rotated versions of itself and with a special constant byte c = 0x05. The four circular rotations are through 4, 5, 6, and 7 bit positions to the right.

$$b'_i = b_{(i+2)\ mod\ 8} \otimes b_{(i+5)\ mod\ 8} \otimes b_{(i+7)\ mod\ 8} \otimes d_i$$

   2. find the multiplicative inverse x′= $\boldsymbol{x_{in}^{-1}}$ in GF($\mathbf{2^8}$)
   3. The bytes c and d are chosen so that the S-box has no fixed points. We do not want S box(a) = a for any a. Neither do we want S box(a) = a where a is the bitwise complement of a.

iii. Add round key (InvAddRoundKey)

iv. Inverse mix columns (nvMixColumns)

$$\begin{bmatrix} 0E & 0B & 0D & 09 \\ 09 & 0E & 0B & 0D \\ 0D & 09 & 0E & 0B \\ 0B & 0D & 09 & 0E \end{bmatrix} \times \begin{bmatrix} s_{0.0} & s_{0,1} & s_{0,2} & s_{0,3} \\ s_{1.0} & s_{1,1} & s_{1,2} & s_{1,3} \\ s_{2.0} & s_{2,1} & s_{2,2} & s_{2,3} \\ s_{3.0} & s_{3,1} & s_{3,2} & s_{3,3} \end{bmatrix} = \begin{bmatrix} s'_{0.0} & s'_{0,1} & s'_{0,2} & s'_{0,3} \\ s'_{1.0} & s'_{1,1} & s'_{1,2} & s'_{1,3} \\ s'_{2.0} & s'_{2,1} & s'_{2,2} & s'_{2,3} \\ s'_{3.0} & s'_{3,1} & s'_{3,2} & s'_{3,3} \end{bmatrix}$$

IV. The last round for encryption does not involve the Mix columns step. The last round for decryption does not involve the Inverse mix columns step

▦double DES: has two DES-based encryption stages using two different keys.

$$C = E(K_2,\ E(K_1,\ P))$$

$$P = D(K_1,\ D(K_2,\ C))$$

1. Since the plaintext-to-ciphertext mapping must be one-one, single application of DES encryption can be thought of as a specific permutation of the $\mathbf{2^{64}}$different possible integer values for a plaintext block. Relationship between the two keys K1 and K2 of 2DES and some single key K3 is:

$$E(K_2,\ E(K_1,\ P)) = E(K_3,\ P)$$

So any number of multiple encryptions of plaintext would amount to a single encryption of regular DES. Therefore, a cipher consisting of three applications of DES encryption, as in 3DES, would be no stronger than regular DES.

2. If we said that 2DES with the two keys (K1 K2) is equivalent to a single application of DES with some key K3, set of permutations achieved with different possible 56-bit DES encryptions is closed, so the set of permutations corresponding to DES encryption forms a group. However, as it turns out, the set of permutations corresponding to DES encryptions/decryptions does not constitute a group. The set of all possible permutations over 64-bit words is of size $\mathbf{2^{64}}!$ (possible mappings between the input words and the output words.). With a key size of 56 bits, we have a total of different $\mathbf{2^{56}}$ keys. Each key corresponds to one of the different$\mathbf{2^{64}}!$ possible mappings. This implies that the permutation produced by 2DES (or by, say, 3DES) is not guaranteed to belong to the set of size that corresponds to a single application of DES. So 2DES would be equivalent to single DES for some K3 is not guaranteed and the probability of finding such a triple (K1 k2 k3) by searching only through the permutations created by the 56-bit DES keys is negligibly small.
3. Any double block cipher, that is a cipher that carries out double encryption of the plaintext using two different keys in order to increase the cryptographic strength of the cipher, is open to the meet-in-the-middle attack.
   I. there exists an X such that
      $$X = E(K_1, P) = D(K_2, C)$$
   II. attacker creates a sorted table of all possible value for X for a given P by trying all possible $\mathbf{2^{56}}$keys. This table will have $\mathbf{2^{56}}$entries and another X-sorted table of all possible X by decrypting C using every one of the keys. This table also has $\mathbf{2^{56}}$entries.
   III. make a total of $\mathbf{2^{112}}$ comparisons to figure out which entries in the tables are the same. These comparisons involve only $\mathbf{2^{64}}$different possible values for X. (X is a 64-bit word.) Then it must be case that$\frac{2^{112}}{2^{64}} = \mathbf{2^{48}}$of the comparisons must involve identical values in the two tables.

IV. So on the average we are likely to run into 2^{48}false alarms.

V. suppose the attacker has another (P',C′) pair of 64-bit words. This time, we will only try the 2^{48} (K1,K2)pairs. We will see negative redundancy$\frac{2^{48}}{2^{64}} = 2^{-16}$ which implies that there will only be a single key pair (K1,K2) with the same X value in the tables

▦An obvious defense against the meet-in-the-middle attack is to use triple DES.

1. One way to use triple DES is with just two keys: one stage of encryption is followed by one stage of decryption, followed by another stage of encryption. This is also referred to as EDE encryption,
$$C = E(K_1, D(K_2, E(K_1, P)))$$
2. There is an important reason for juxtaposing a stage of decryption between two stages of encryption: it makes the triple DES system easily usable by those who are only equipped to use regular DES. This backward compatibility with regular DES can be achieved by setting k1=k2 in triple DES.
3. juxtaposing a decryption stage between two encryption stages does not weaken the resulting cryptographic system in any way. If you encrypt data with one key and try to decrypt with a different key, the final output will be still be an encrypted version of the original input.
4. encryption equation for two-key 3DES can rewrite as
$$\begin{aligned} A &= E(K_1, P) \\ B &= D(K_2, A) \\ C &= E(K_1, B) \end{aligned}$$
If the attacker had some way of knowing the intermediate value A for a given plaintext P, breaking the 3DES cipher becomes the same as breaking 2DES with the meet-in-the-middle attack.
5. In the absence of knowledge of A, the attacker can assume some arbitrary value for A and can then try to find a known (P, C)
   I. procures n pairs of (P, C) arranged in a two-column table 1. For a given pair (P,C), the probability of guessing the correct intermediate A is $1/2^{64}$. Therefore, given the n pairs of (P, C) values in Table I, the probability that a particular chosen value for A will be correct is $n/2^{64}$
   II. chooses an arbitrary A., figures out the plaintext that will result in that A for every possible key K1. If a P calculated in

this manner is found to match one of the rows in Table 1, B value and its corresponding key K1 is entered as a row in Table II

III. Given all the available (P, C) pairs, we now fill Table II with (B,K1) pairs where the set of K1's constitutes our candidate pool for the K1 key.

IV. sort Table II on the B values.

V. try, one at a time, all possible values for the K2 key in this equation for the assumed value for A. (2^56 possible values for K2) When we get a B that is in one of the rows of Table II, we have found a candidate pair (K, K2). Implies that the running time of the attack would be $\mathbf{2^{56} * \left(\frac{2^{64}}{n}\right) = 2^{120-\log n}}$

VI. candidate pair of keys (K1,K2)is tested on the remaining (P, C) pairs. If the test fails, we try a different value for A in Step 2 and the process is repeated.

6. TRIPLE DES WITH THREE KEYS

$$C = E(K_3, D(K_2, E(K_1, P)))$$

I. When all three keys are the same, that is when K1 =K2=K3, 3DES with three keys become identical to regular DES. When K1=K3,we have 3DES with two keys.

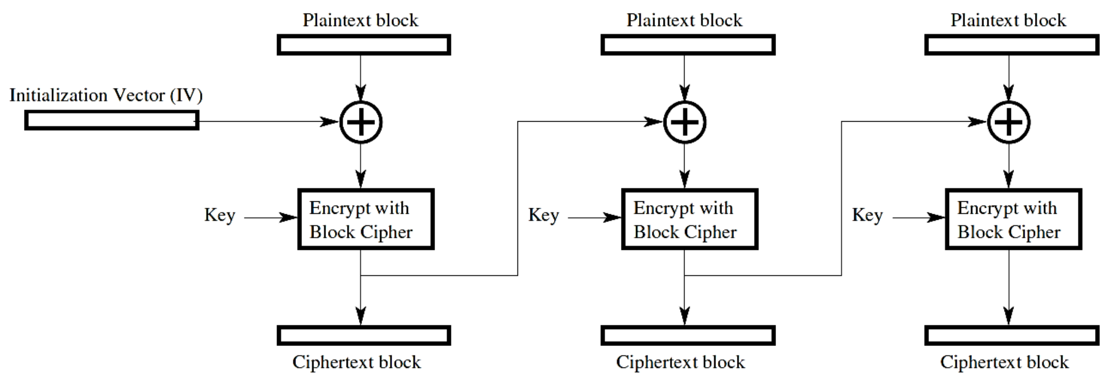
▦block cipher has been demonstrated to be strong (not vulnerable to brute-force, meet-in-the-middle, typical statistical, and other such attacks), does not imply that it will be sufficiently secure if you are using it to transmit long messages (longer than the block length). Interaction between the block-size based periodicity of such ciphers and any repetitive structures in the plaintext may still leave too many clues in the ciphertext that compromise its security. Five different modes in which any block cipher can be used

1. Electronic Code Book (ECB): scanning a long document one block at a time and enciphering it independently of the blocks seen before or the blocks to be seen next. Encryption process can be represented by a fixed mapping between the input blocks of plaintext and the output blocks of cipher text. For this mode to work correctly, either the message length must be an integral multiple of the block size or you must use padding so that the condition on the length is satisfied. It is not suitable for long messages, but 4 other modes variate from this.

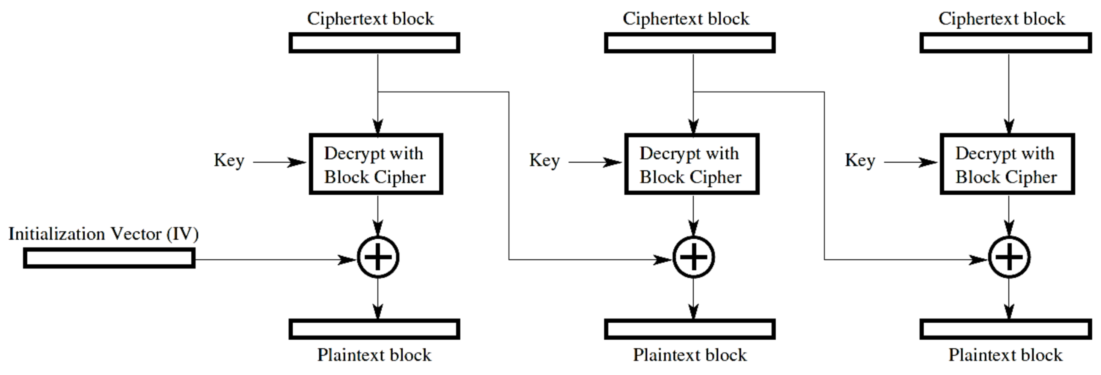
I. When a block cipher is used in ECB mode, each block of plaintext is coded independently. This makes it not very secure for long segments of plaintext, especially plaintext containing repetitive information. Used primarily for secure transmission of short pieces of information, such as an encryption key.

II. Another shortcoming of ECB is that the length of the plaintext message must be integral multiple of the block size. When that condition is not met, the plaintext message must be padded appropriately.

2. Cipher Block Chaining Mode (CBC): The input to the encryption algorithm is the XOR of the next block of plaintext and the previous block of ciphertext. This is obviously more secure for long segments of plaintext. However, this mode also requires that length of the plaintext message be an integral multiple of the block size. When that condition is not satisfied, the message must be suitably padded.



CBC Encryption



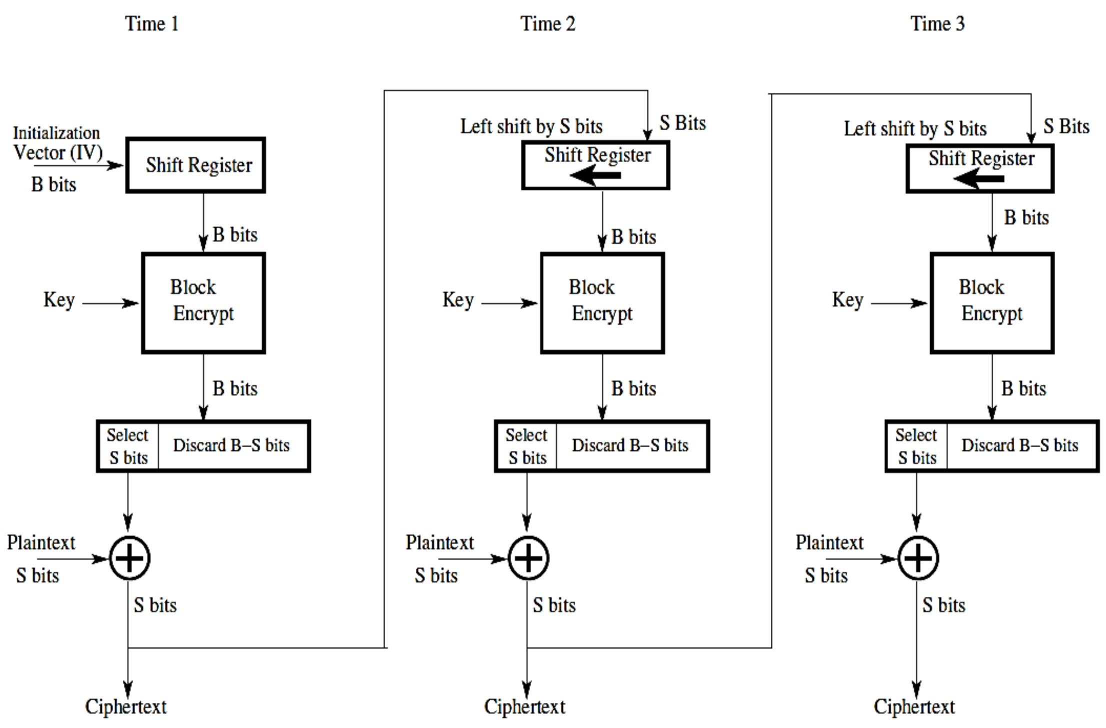
CBC Decryption

I. initialization vector is needed for the first invocation of the encryption algorithm.

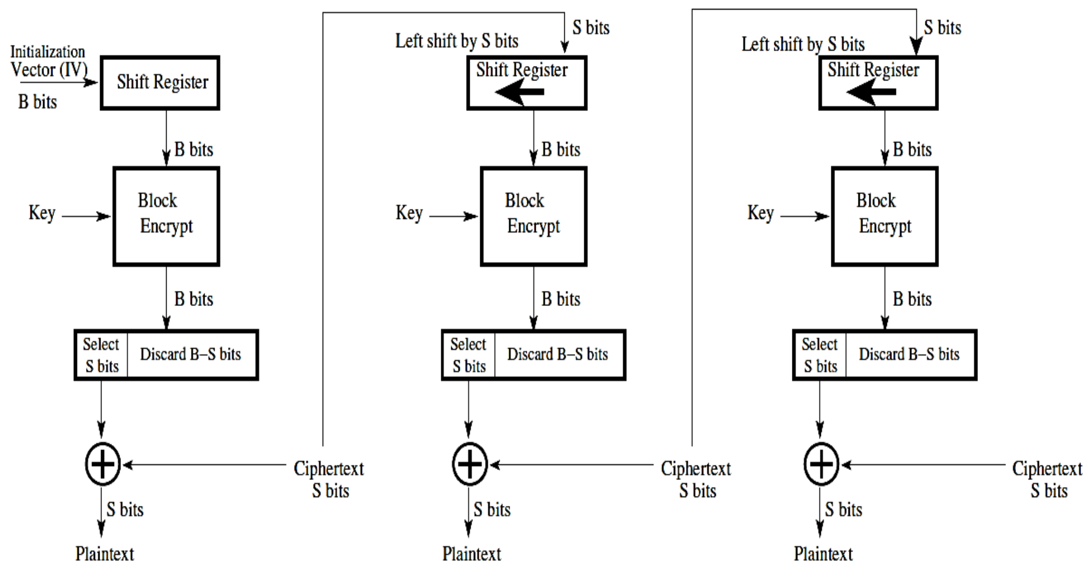
II. With this chaining scheme, the ciphertext block for any given plaintext block becomes a function of all the previous iphertext blocks.

3. Cipher Feedback Mode (CFB): uses only a fraction of the previous ciphertext block to compute the next ciphertext block, the encryption system digests only $s < b$ (b is the blocksize used by the block cipher) number of plaintext bits at a time even though the encryption algorithm itself carries out a b-bits to b-bits transformation. Since s can be any number, including one byte, that makes CFB suitable as a stream cipher. The ciphertext byte produced for any plaintext byte depends on all the previous plaintext bytes in the CFB mode.

   I. Start with an initialization vector, IV, of the same size as the blocksize expected by the block cipher.
   II. Encrypt the IV with the block cipher encryption algorithm.
   III. Retain only one byte from the output of the encryption algorithm. Let this be the most significant byte.
   IV. XOR the byte retained with the byte of the plaintext that needs to be transmitted. Transmit the output byte produced.
   V. Shift the IV one byte to the left (discarding the leftmost byte) and insert the ciphertext byte produced by the previous step as the rightmost byte. So the new IV is still of the same length as the block size expected by the encryption algorithm.
   VI. Go back to the step Encrypt the IV with the block cipher encryption algorithm



CFB Encryption



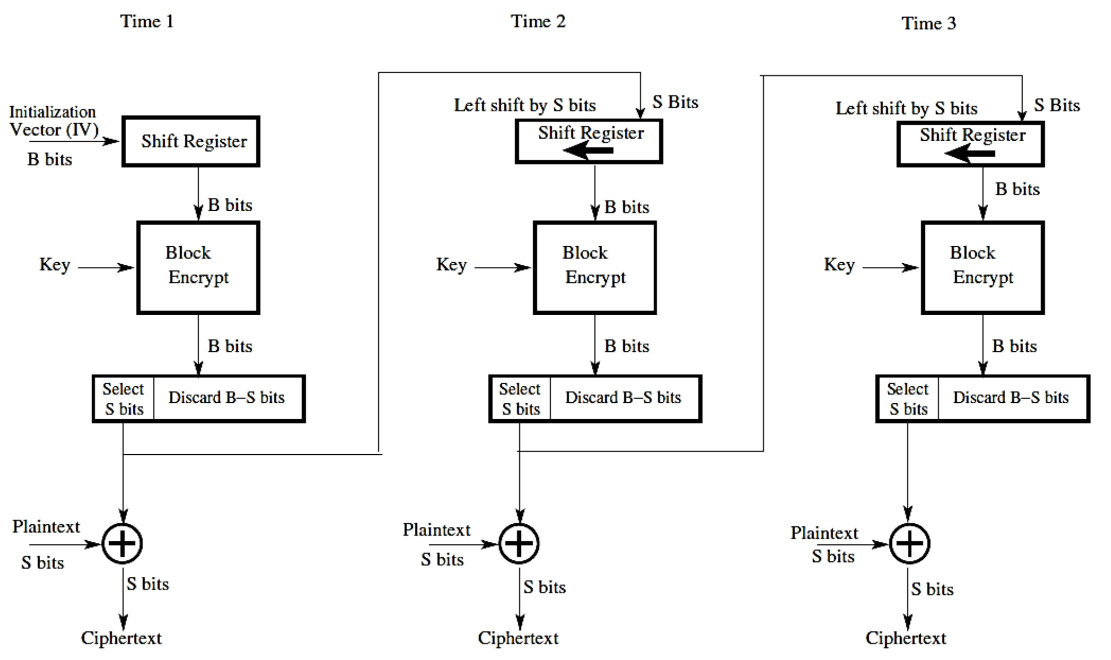
CFB Decryption

4. Output Feedback Mode (OFB): feed s bits from the output of the transformation itself. This mode of operation is also suitable if you want to use a block cipher as a stream cipher.
   I. The only difference between CFB and OFB is that we feed back one byte (the most significant byte) from the output of the block cipher encryption algorithm, as opposed to feeding back the actual ciphertext byte. This makes OFB more resistant to

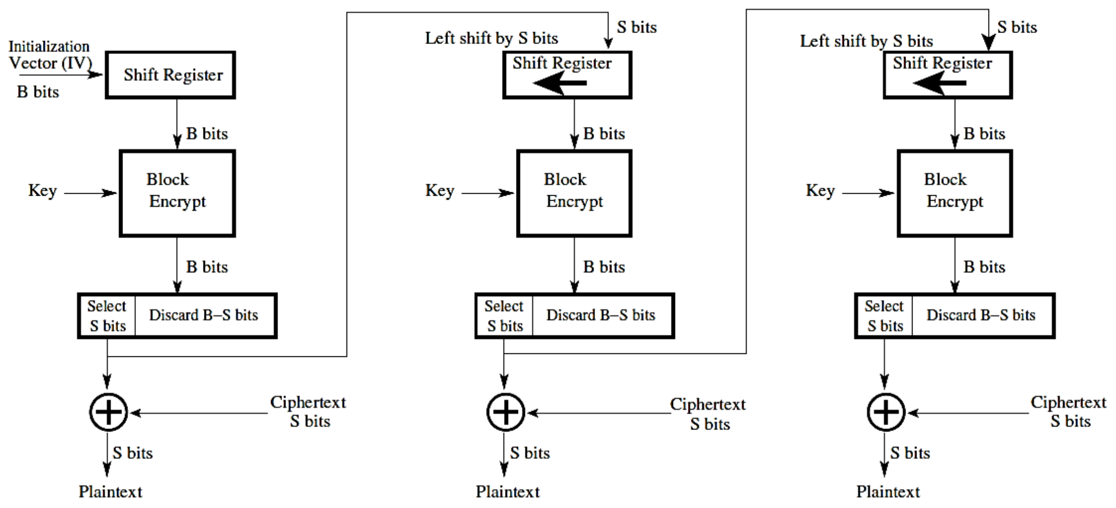
transmission bit errors.

II. Considering CFB, you have encrypted and transmitted the first byte of plaintext. Now suppose this byte is received with a one or more bit errors that error will also propagate to downstream decryptions because the received ciphertext byte is also fed back into the decryption of the next byte.

III. what is fed back in OFB is completely locally generated at the receiver. The information that is fed back is not exposed to the possibility of transmission errors in OFB.



OFB Encryption

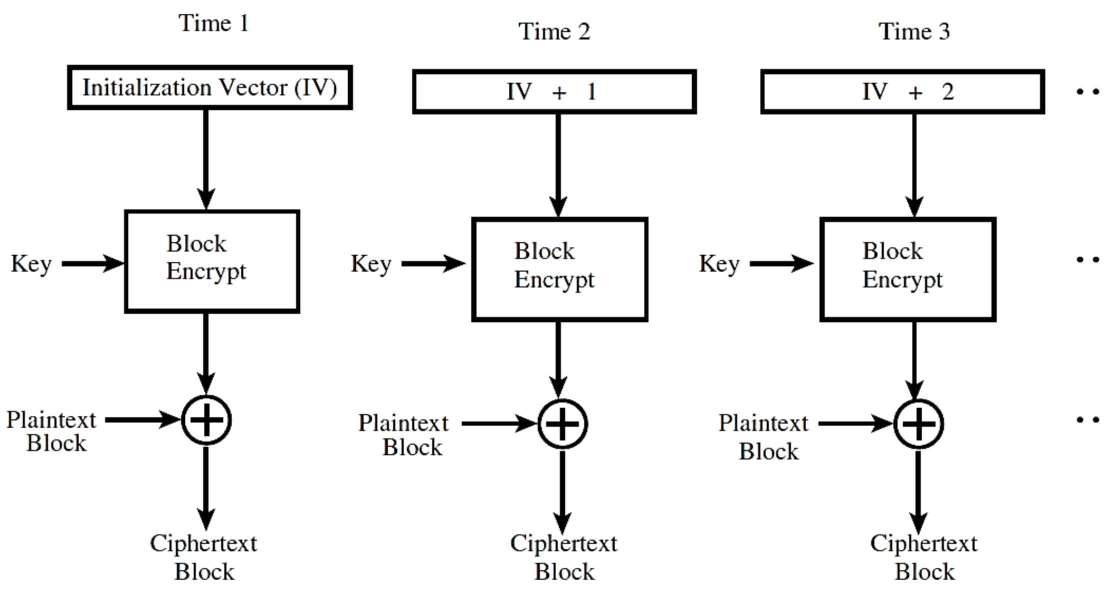


OFB Decryption

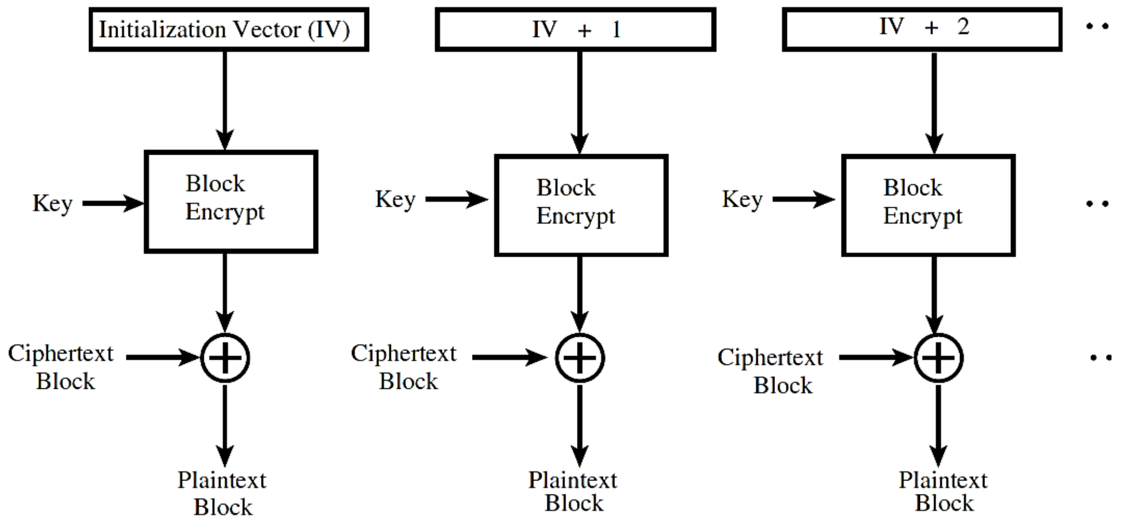
5. Counter Mode (CTR): applying the encryption algorithm not to the plaintext

directly, but to a b-bit number (and its increments modulo 2^b for successive blocks) that is chosen beforehand. The ciphertext consists of what is obtained by XORing the encryption of the number with a b-bit block of plaintext. This mode is at least as secure as the other modes

I. CFB and OFB, are intended to use a block cipher as a stream cipher, the counter mode (CTR) retains the pure block structure relationship between the plaintext and ciphertext.
II. no part of the plaintext is directly exposed to the block encryption algorithm in the CTR mode. The encryption algorithm encrypts only a b-bit integer produced by the counter. What is transmitted is the XOR of the encryption of the integer and the b bits of the plaintext.
III. For the counter value, we start with some number for the first plaintext block and then increment this value modulo 2^b from block to block,
IV. only the forward encryption algorithm is used for both encryption and decryption. (This is of significance for block ciphers for which the encryption algorithm differs substantially from the decryption algorithm. AES is a case in point.) (This property of CTR is also true for CFB and OFB modes.)
V. advantages of the CTR mode for using a block cipher:
    i. Fast encryption and decryption: we can precompute the encryptions for as many counter values as needed. Then, at the transmit time, we only have to XOR the plaintext blocks with the pre-computed b-bit blocks. The same applies to fast decryption.
    ii. at least as secure as the other four modes for using block ciphers.
    iii. Because there is no block-to-block feedback, the algorithm is highly amenable to implementation on parallel machines. For the same reason, any block can be decrypted with random access.



CTR Encryption



CTR Decryption