Cryptography

Md. Shadmim Hasan Sifat

Lecturer in CSE,SUST

CSE 461 (Intro To Comp. Security and Forensics)

Jan - June 24

Cryptography

Practice and study of techniques for secure communication in the presence of adversarial behavior

Is about constructing and analyzing protocols that prevent third parties or the public from reading private messages

Hidden Writing : Hide the meaning of the message

Depends on secrecy of a short key, **not method**

Cryptography

- Cryptography (cryptographer)
  - Creating ciphers
- Cryptanalysis (cryptanalyst)
  - Breaking ciphers

- The history of cryptography is an arms race between cryptographers and cryptanalysts.

Goals of Cryptography

- The most fundamental problem cryptography addresses:
  - ensure security of communication over insecure medium

- What does secure communication mean?
  - confidentiality (privacy, secrecy) : only the intended recipient can see the communication
  - integrity (authenticity) : the communication is generated by the alleged sender
- Insecure medium
  - Passive attacker: the adversary can eavesdrop
  - Active attacker: the adversary has full control over the communication channel

Cryptography < Security

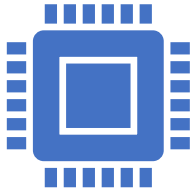
- Cryptography isn't the solution to security
  - Buffer overflows, worms, viruses etc. can still happen and cause damages
- Even when used, difficult to get right
  - Choice of encryption algorithms
  - Choice of parameters
  - Implementation
  - Hard to detect errors : vulnerability can only be recognized after exploitation
    - ..think of Caesar / substitution ciphers

Some Related Security Principles

Open Design Principle

Do not rely on secret designs, attacker ignorance, or security by obscurity.

Kerckhoffs' principle—*a system's security should not rely on the secrecy of its design details*.



Time-tested tools

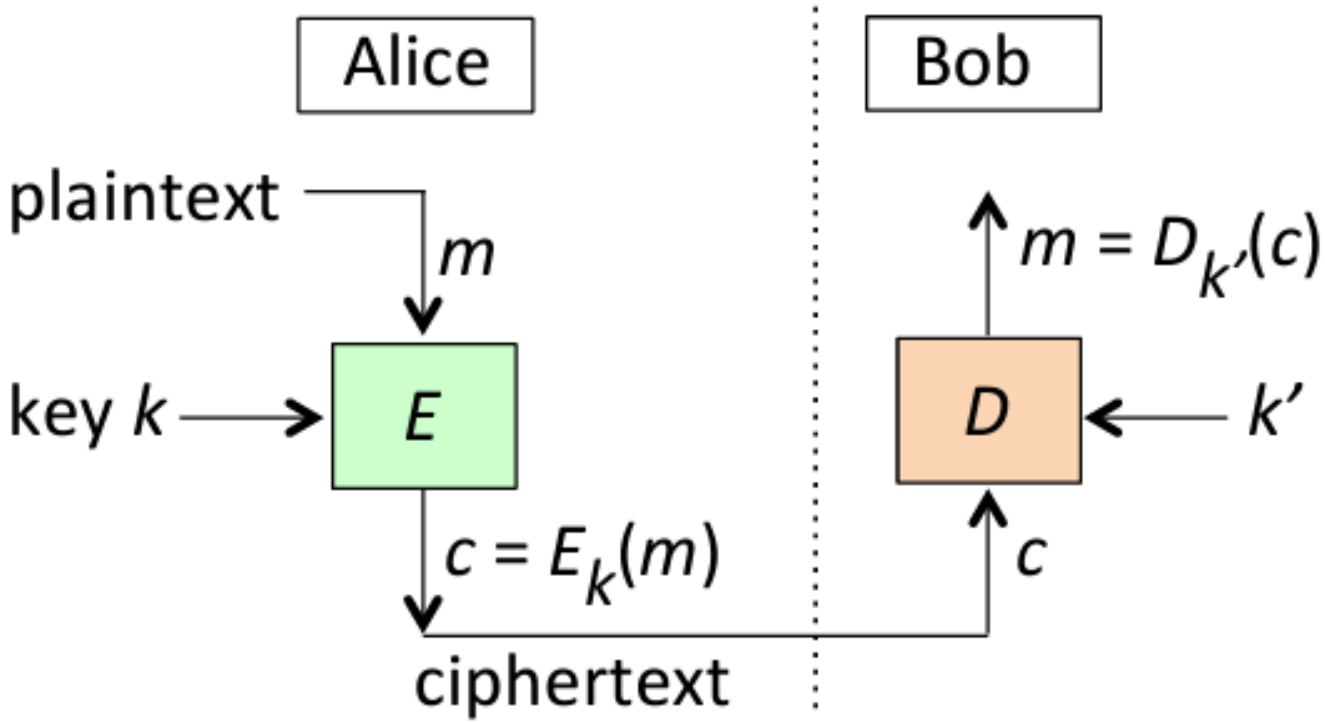
Rely wherever possible on time-tested, expert-built security tools including protocols, cryptographic primitives and toolkits, rather than designing and implementing your own.

History shows that security design and implementation is difficult to get right even for experts.

Sufficient work factor

tune the mechanism so that the cost to defeat the system (work factor) clearly exceeds the resources of anticipated classes of adversaries.

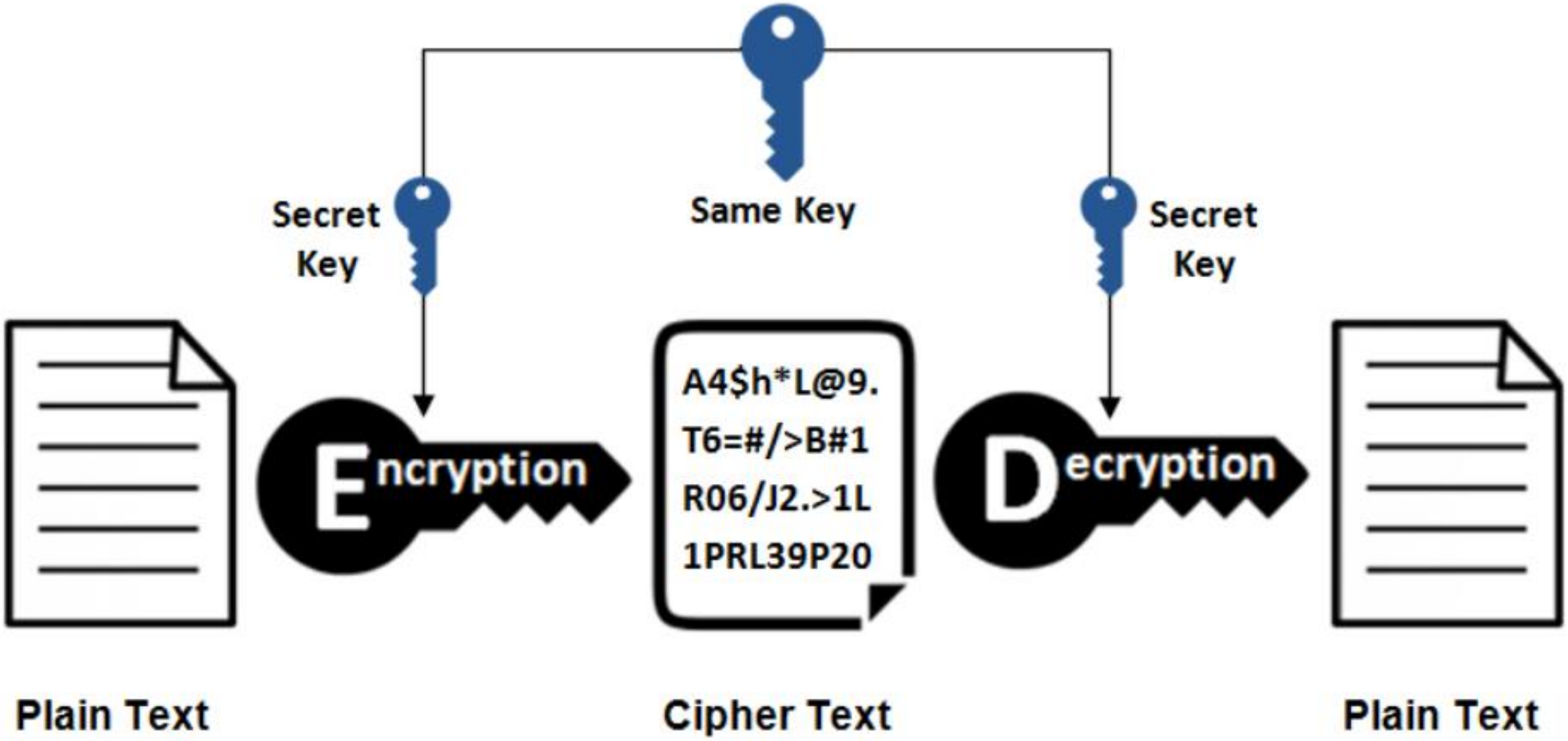
Generic Concepts



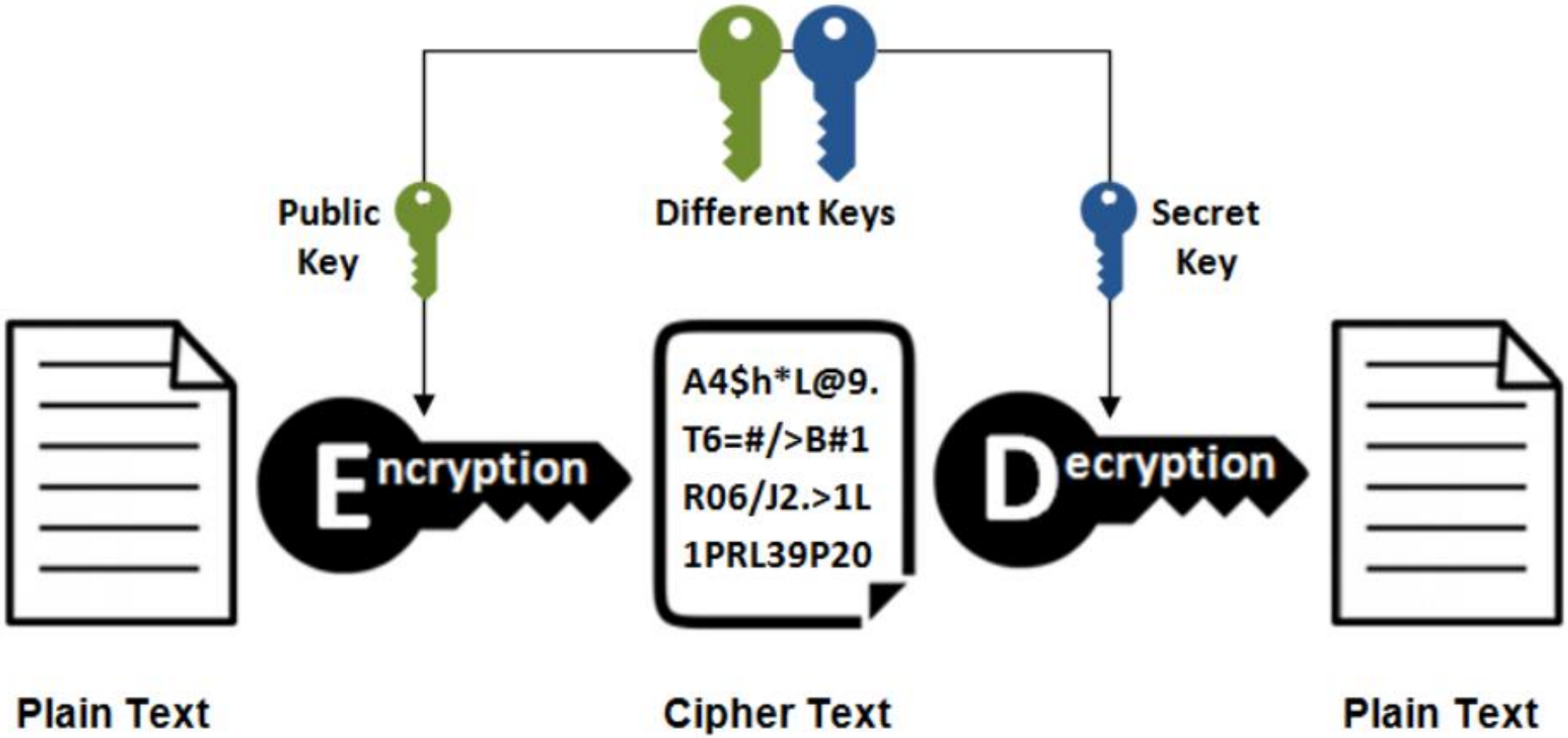
Types of Cryptography

- Symmetric key
  - Encryption and decryption keys are the same
- Asymmetric key
  - Encryption and decryption keys differ

Symmetric Encryption



Asymmetric Encryption



Caesar Cipher

- Every character is replaced with the character K slots to the right
  - S E C U R I T Y ::
    V H F X U L W B

$$E_n(x) = (x + n) mod\ 26$$

(Encryption Phase with shift n)

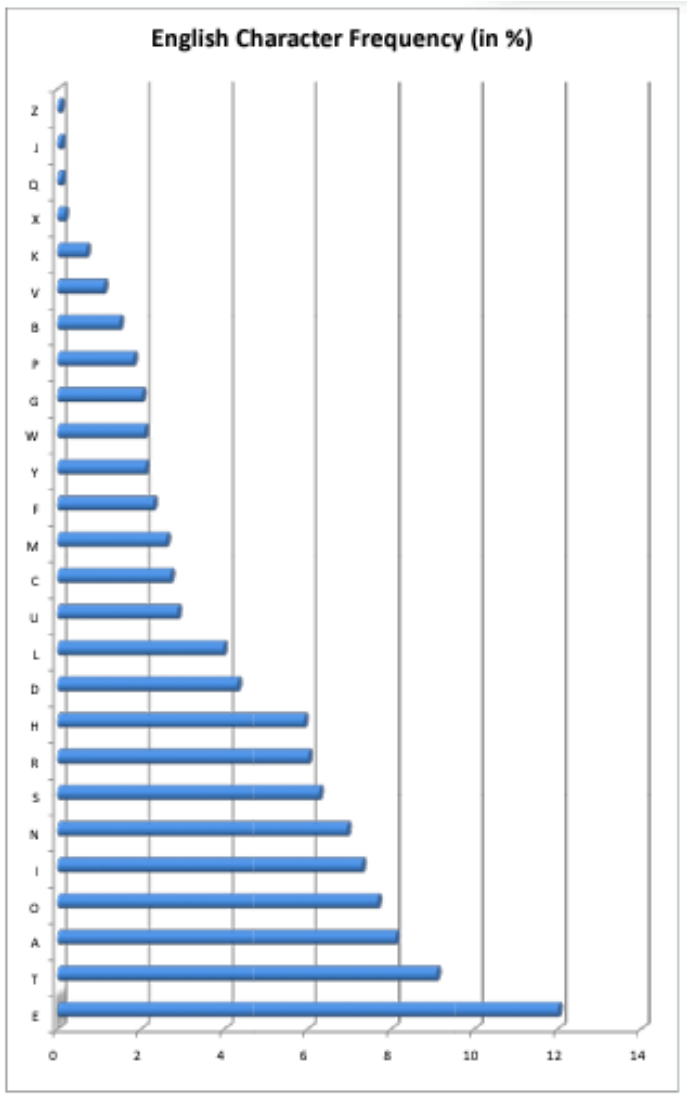
$$D_n(x) = (x - n) mod\ 26$$

(Decryption Phase with shift n)

Substitution Cipher

A	B	C	D	E	F	G	H	I	J	K	L	M
↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓
C	M	T	E	F	H	P	U	D	X	N	Z	L

N	O	P	Q	R	S	T	U	V	W	X	Y	Z
↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓
O	A	J	R	Y	I	G	W	V	B	S	Q	K



Cryptanalysis Hints

Order Of Frequency Of Single Letters	E T A O I N S H R D L U
Order Of Frequency Of Digraphs	th er on an re he in ed nd ha at en es of or nt ea ti to it st io le is ou ar as de rt ve
Order Of Frequency Of Trigraphs	the and tha ent ion tio for nde has nce edt tis oft sth men
Order Of Frequency Of Most Common Doubles	ss ee tt ff ll mm oo
Order Of Frequency Of Initial Letters	T O A W B C D S F M R H I Y E G L N P U J K
Order Of Frequency Of Final Letters	E S T D N R Y F L O G H A K M P U W
One-Letter Words	a, I.
Most Frequent Two-Letter Words	of, to, in, it, is, be, as, at, so, we, he, by, or, on, do, if, me, my, up, an, go, no, us, am
Most Frequent Three-Letter Words	the, and, for, are, but, not, you, all, any, can, had, her, was, one, our, out, day, get, has, him, his, how, man, new, now, old, see, two, way, who, boy, did, its, let, put, say, she, too, use
Most Frequent Four-Letter Words	that, with, have, this, will, your, from, they, know, want, been, good, much, some, time

Substitution Cipher : Example

- Vg gbbx n ybg bs oybbq, fjrng naq grnef gb trg gb jurer jr ner gbqnl, ohg jr unir whfg ortha. Gbqnl jr ortva va rnearfg gur jbex bs znxvat fher gung gur jbeyq jr yrnir bhe puvyqera vf whfg n yvggyr ovg orggre guna gur bar jr vaunovg gbqnl.

- It took a lot of blood, sweat and tears to get to where we are today, but we have just begun. Today we begin in earnest the work of making sure that the world we leave our children is just a little bit better than the one we inhabit today.

n:a	r:e	g:t	b:o	v:i	a:n	x:k	y:l	s:f	u:h	t:g
i:v	f:s	o:b	e:r	p:c	q:d	j:w	h:u	w:j	l:y	z:m

Hill Cipher

$$\vec{x} = \begin{bmatrix} 2 \\ 0 \\ 19 \end{bmatrix}.$$

$$\vec{c} = K \cdot \vec{x}, \qquad \vec{x} = K^{-1} \cdot \vec{c},$$

$$K^{-1} \cdot \vec{c} = K^{-1} \cdot (K \cdot \vec{x}) = (K^{-1} \cdot K) \cdot \vec{x} = \vec{1} \cdot \vec{x} = \vec{x}.$$

Hill Cipher

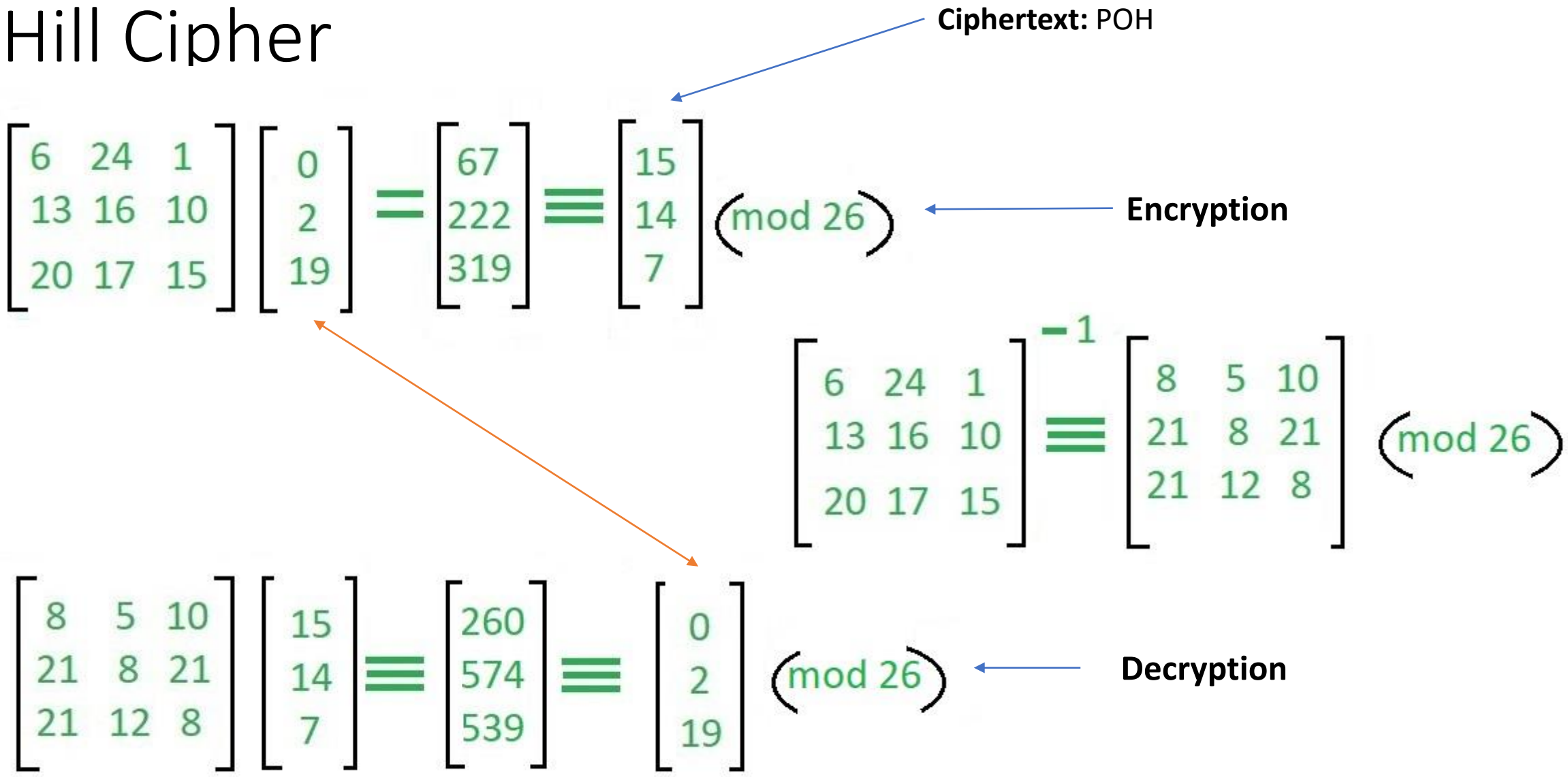
Plaintext : ACT

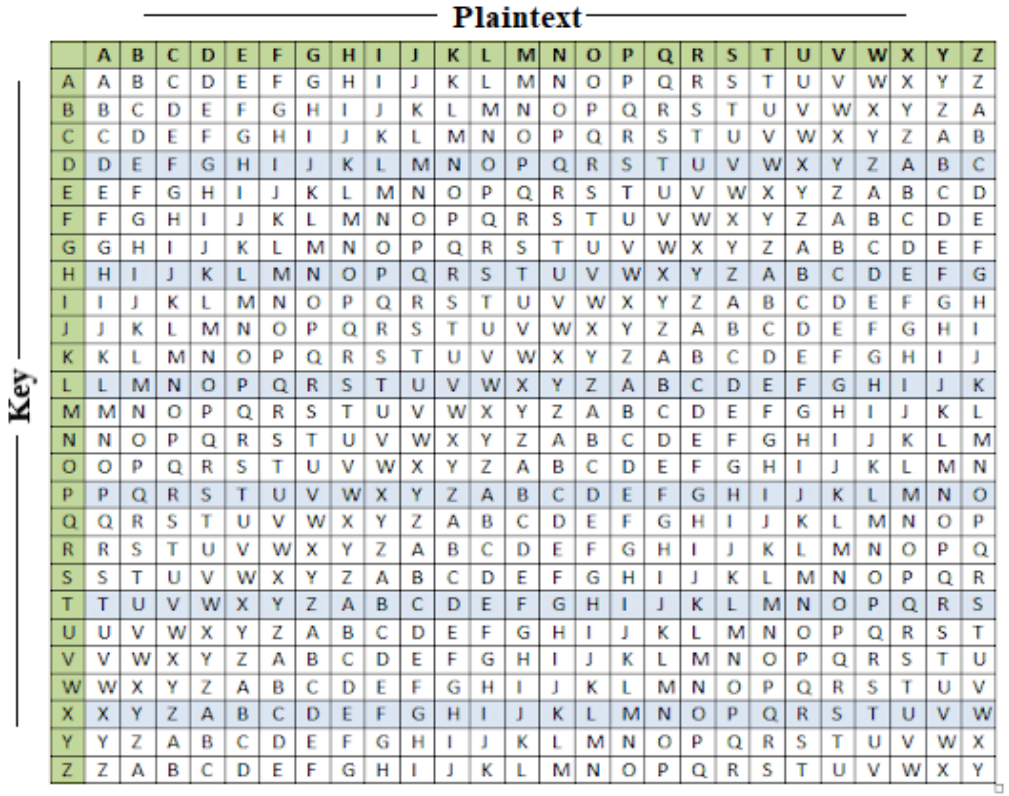
$$\begin{bmatrix} 0 \\ 2 \\ 19 \end{bmatrix}$$

KEY : GYBNQKURP >>> GYB-NQK-URP

$$\begin{bmatrix} 6 & 24 & 1 \\ 13 & 16 & 10 \\ 20 & 17 & 15 \end{bmatrix}$$

Hill Cipher


Ciphertext: POH
$\begin{bmatrix} 6 & 24 & 1 \\ 13 & 16 & 10 \\ 20 & 17 & 15 \end{bmatrix} \begin{bmatrix} 0 \\ 2 \\ 19 \end{bmatrix} = \begin{bmatrix} 67 \\ 222 \\ 319 \end{bmatrix} \equiv \begin{bmatrix} 15 \\ 14 \\ 7 \end{bmatrix} \pmod{26}$
Encryption
$\begin{bmatrix} 6 & 24 & 1 \\ 13 & 16 & 10 \\ 20 & 17 & 15 \end{bmatrix}^{-1} \equiv \begin{bmatrix} 8 & 5 & 10 \\ 21 & 8 & 21 \\ 21 & 12 & 8 \end{bmatrix} \pmod{26}$
$\begin{bmatrix} 8 & 5 & 10 \\ 21 & 8 & 21 \\ 21 & 12 & 8 \end{bmatrix} \begin{bmatrix} 15 \\ 14 \\ 7 \end{bmatrix} \equiv \begin{bmatrix} 260 \\ 574 \\ 539 \end{bmatrix} \equiv \begin{bmatrix} 0 \\ 2 \\ 19 \end{bmatrix} \pmod{26}$
Decryption



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

Plaintext:	attackatdawn
KEY:	LEMONLEMONLE
Ciphertext:	LXFOPVEFRNHR

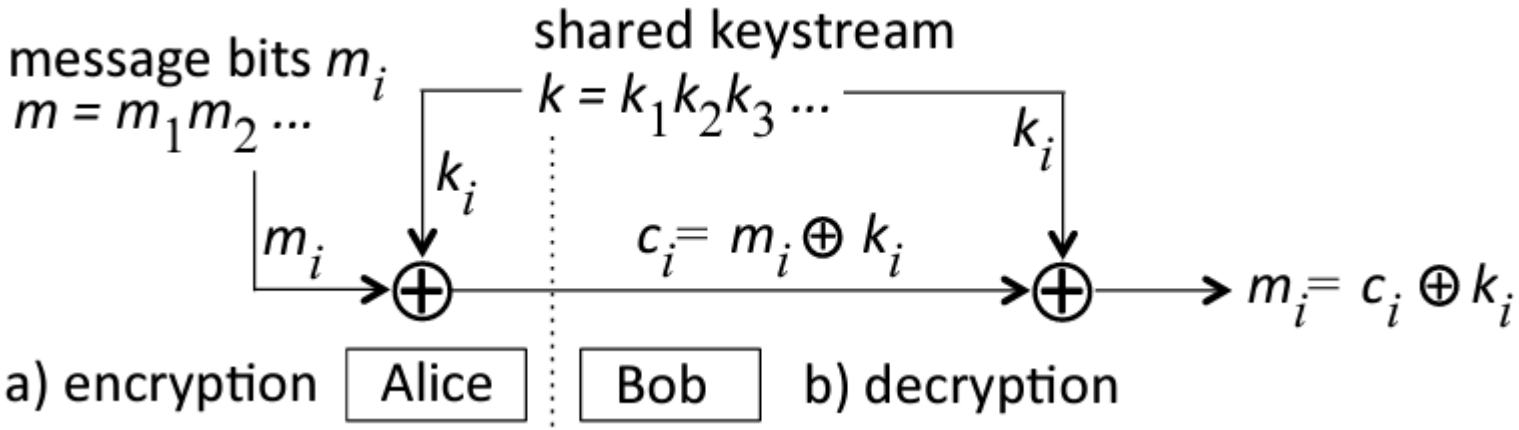
Vigenère cipher

- **Encryption**
- $E_i = (P_i + K_i) \bmod 26$

- **Decryption**
- $D_i = (E_i - K_i) \bmod 26$

Vernam Cipher

every subsequent message gets a different key



One Time Pads (OTP)

The key must be at least as long as the plaintext.

The key must be random (uniformly distributed in the set of all possible keys and independent of the plaintext)

The key must **never be reused** in whole or in part.

The key must be kept completely secret by the communicating parties.

One Time Pads

- One-time pads are ***information-theoretically secure***
  - The encrypted message (i.e., the ciphertext) provides *no information* about the original message to a cryptanalyst.
  - A cryptosystem is considered to have ***information-theoretic security*** if the system is secure against adversaries even with unlimited computing resources and time.
  - A system which depends on the computational cost of cryptanalysis to be secure (and thus can be broken by an attack with unlimited computation) is called ***computationally or conditionally secure***.

One Time Pads : Problems

- Bindings of Key to be as long as message and also has to be secret.

- Difficult to generate truly random key.
  - True random number generators exist but are typically slower and more specialized.

- Very limited practical usage
  - A common use of the one-time pad in quantum cryptography is being used in association with Quantum Key Distribution (QKD).

One Time Pads : Problems

- Vulnerable to ***Reused Key Attacks***
  - Keys must never be used twice!!!

$$E(A) = A \text{ xor } C$$

$$E(B) = B \text{ xor } C$$

$$E(A) \text{ xor } E(B) = (A \text{ xor } C) \text{ xor } (B \text{ xor } C) = A \text{ xor } B \text{ xor } C \text{ xor } C = A \text{ xor } B$$

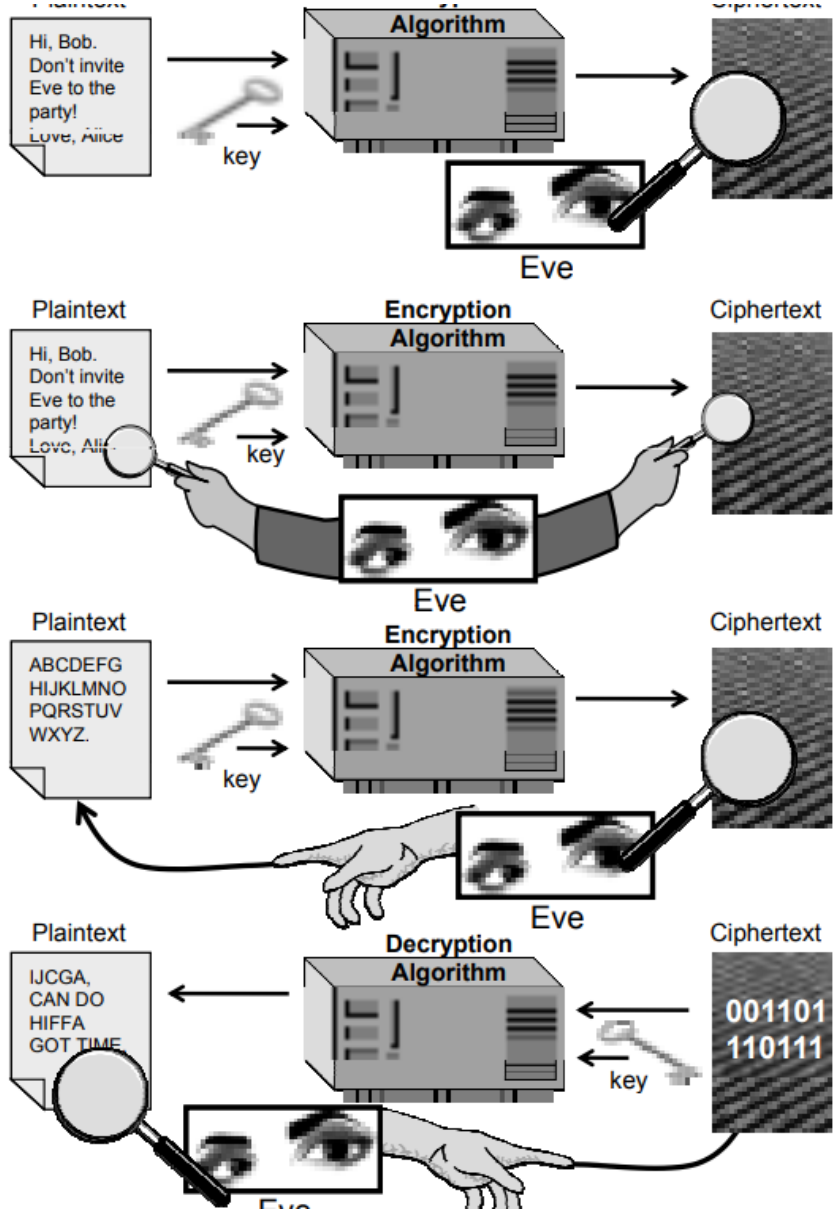
E(A) xor E(B) reveals the plaintext

- Even if neither message is known, as long as both messages are in a natural language, such a cipher can often be broken by paper-and-pencil methods

Types of Attacks

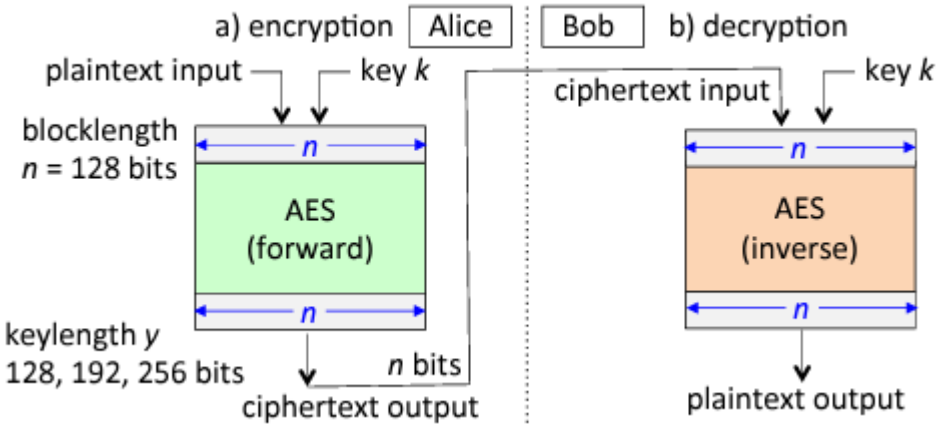
Attacker may have collections of

- ciphertexts (**ciphertext only attack**)
- plaintext/ciphertext pairs (**known plaintext attack**)
- plaintext/ciphertext pairs for plaintexts selected by the attacker (**chosen plaintext attack**)
- plaintext/ciphertext pairs for ciphertexts selected by the attacker (**chosen ciphertext attack**)



Block Ciphers

- Processes plaintext in fixed-length *chunks* or *blocks*
- Each block, perhaps a group of ASCII-encoded characters, is encrypted with a fixed transformation dependent on the key
- *Main properties of algorithm :* block-length and key-length
- If the last plaintext block has fewer bits than the block-length, it is padded with "filler" characters.

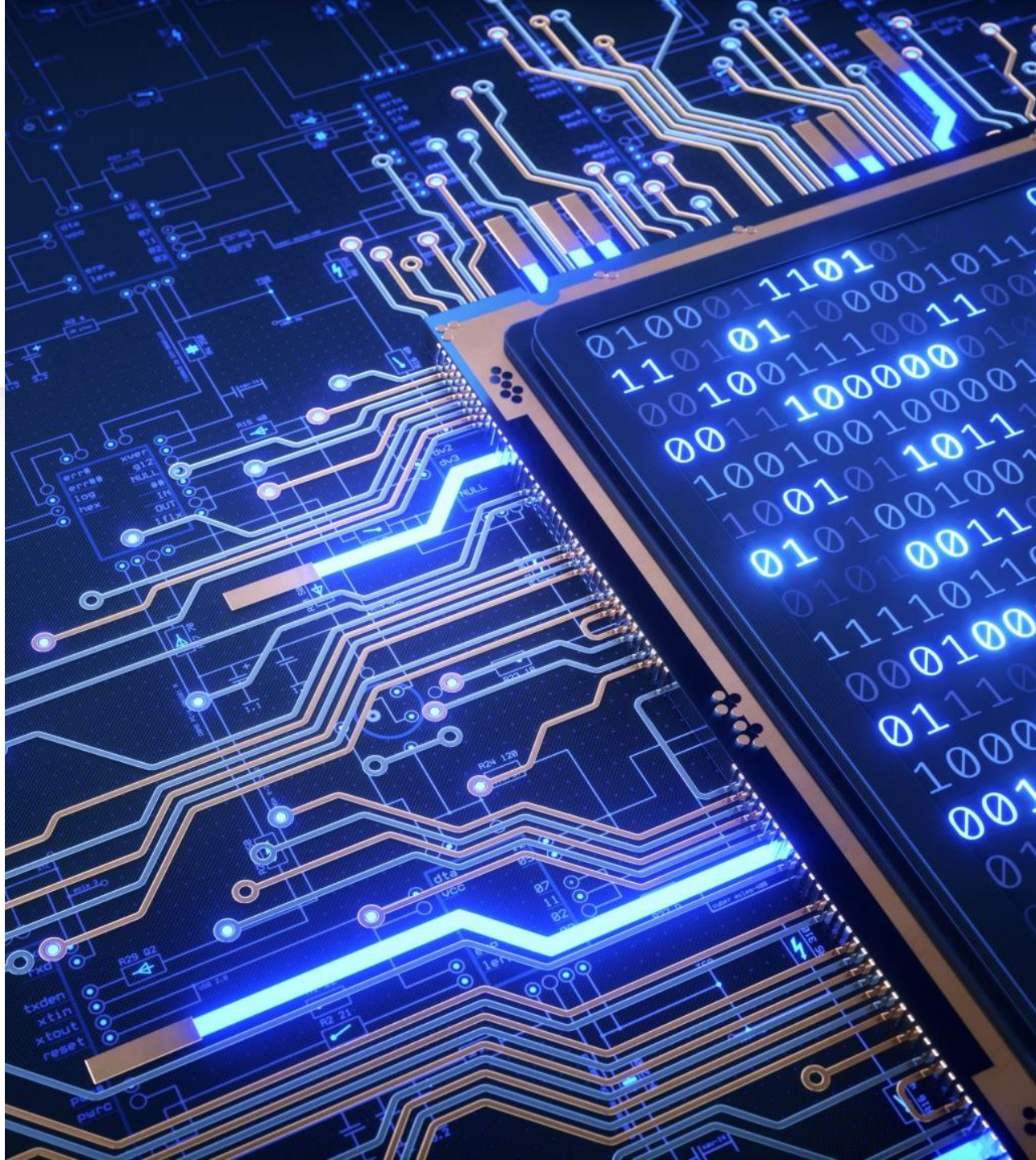


Block Ciphers : Modes of Operation

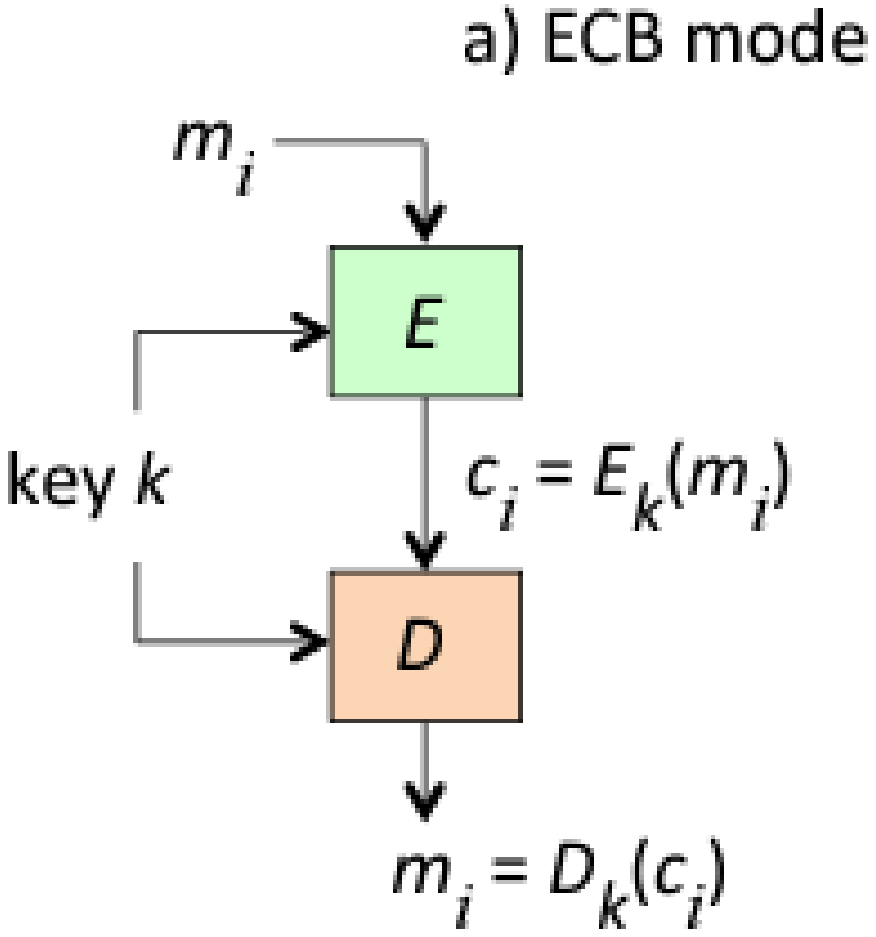
Various methods called *modes of operation* combine successive *n*-bit block operations such that the encryption of one block may or may not depends on other blocks.

For Example:

- ECB (Electronic Code Book)
- CBC (Cipher Block Chaining)
- CTR (Counter)



ECB MODE: Modes of Operation

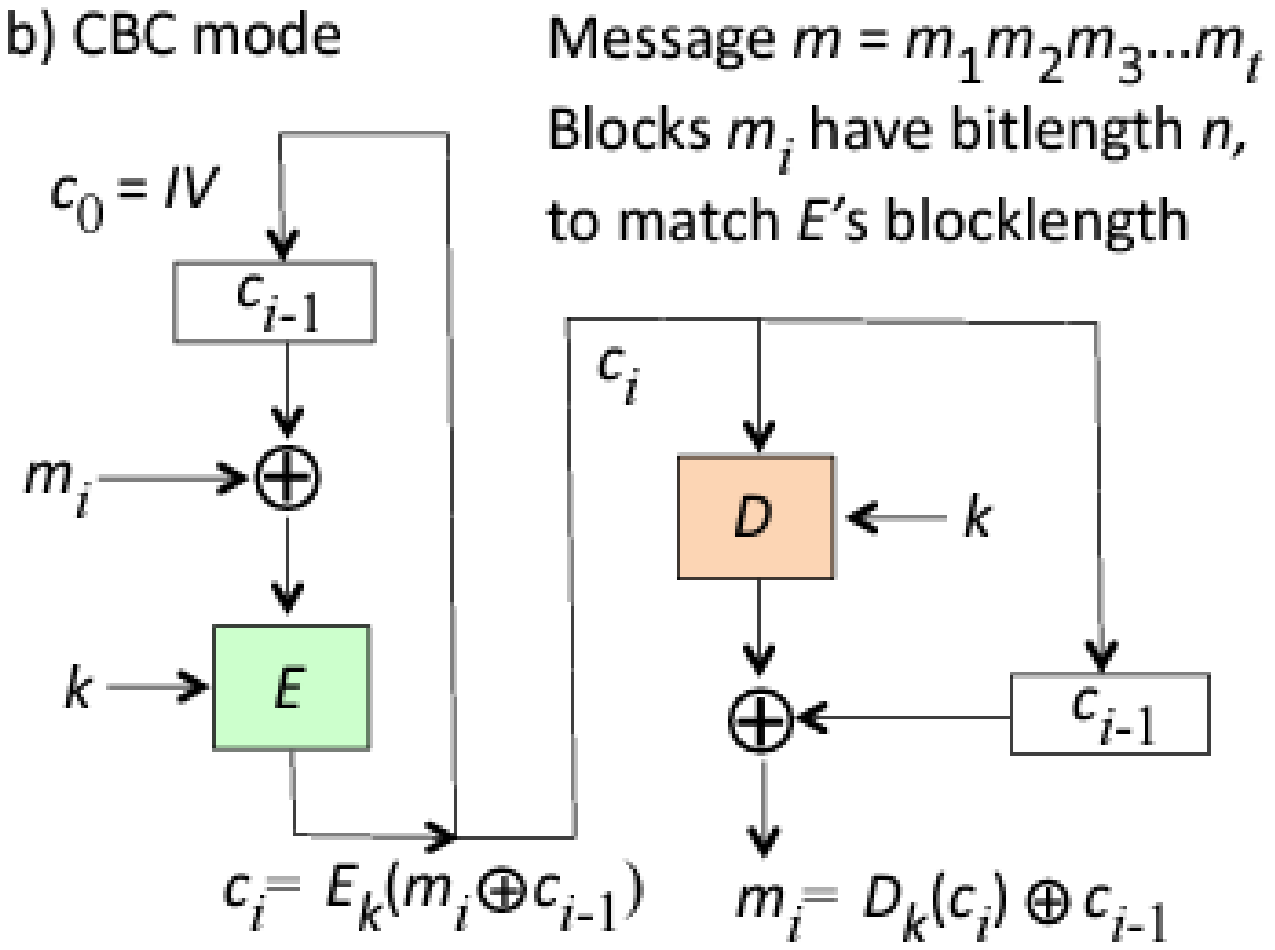


ECB (Electronic Code Book):

Each encryption block operation is independent of adjacent blocks

N.B: If a given key *k* is used to encrypt several identical plaintext blocks m_i then *identical* ciphertext blocks c_i result

CBC MODE: Modes of Operation

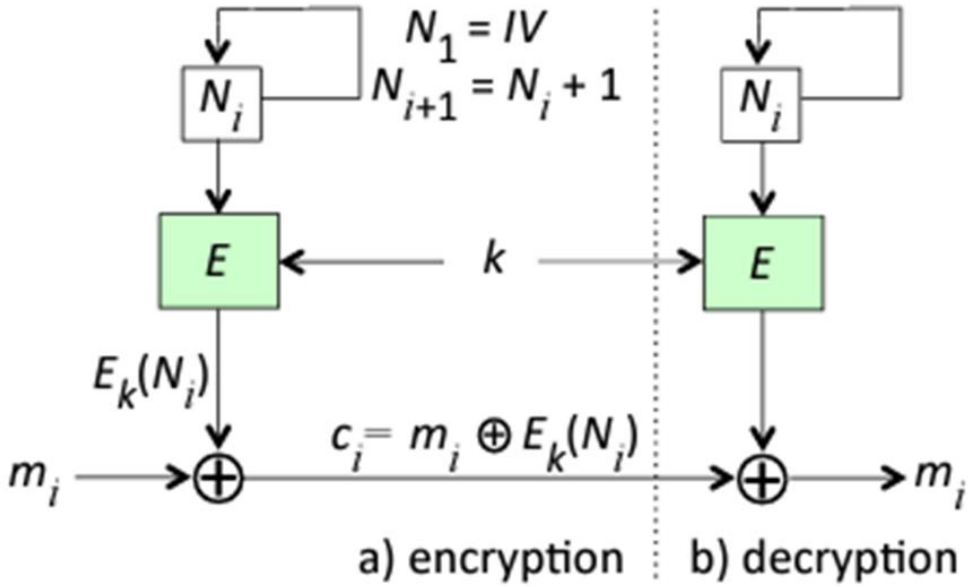


CBC (Cipher Block Chaining):

Each encryption block operation is dependent of adjacent blocks

N.B: If a given key *k* is used to encrypt several identical plaintext blocks m_i then *different* ciphertext blocks c_i result

CTR MODE: Modes of Operation



Counter (CTR) mode

Message $m = m_1 m_2 \ldots m_t$ is encrypted to yield ciphertext $c = c_1 c_2 \ldots c_t$

Blocks m_i , c_i are n bits

CTR (Counter):

Each encryption block operation is *partially* dependent of adjacent blocks

N.B: If a given key *k* is used to encrypt several identical plaintext blocks m_i then *different* ciphertext blocks c_i result

Data Encryption Standard (DES)

- **Why AES, not DES ???**
  - In 1990's the cracking of DES algorithm became possible. Around 50 hours of brute-forcing allowed to crack the message.
  - Requirements from National Institute of Standards and Technology (NIST) was that it had to be efficient both in software and hardware implementations. DES was originally practical only in hardware implementations.
  - DES is insecure due to the relatively short 56 bits key size where AES allows to choose the option for various key lengths like 128-bit, 192-bit or 256-bit key, making it exponentially stronger than the 56-bit key of DES.
- NIST started searching for new feasible algorithm and proposed its requirement in 1997 and put out a public call for a ***replacement to DES***.

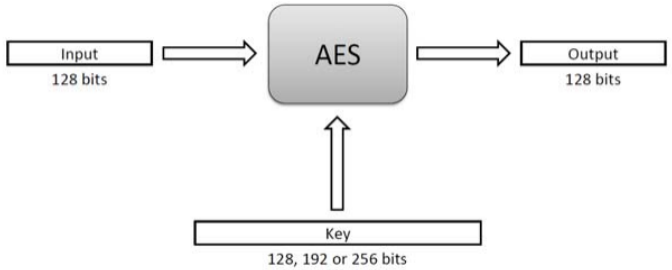
Why AES , not DES ???

- NIST arranged a competition and narrowed down the list of submissions to five finalists.
- In 2001 Rijndael algorithm designed by Rijment and Daemon of Belgium was declared as the winner of the competition.
- As It met all Security, Cost and Implementation criteria.
- NIST ultimately chose the Rijndael algorithm that is now known as the ***Advanced Encryption Standard (AES).***

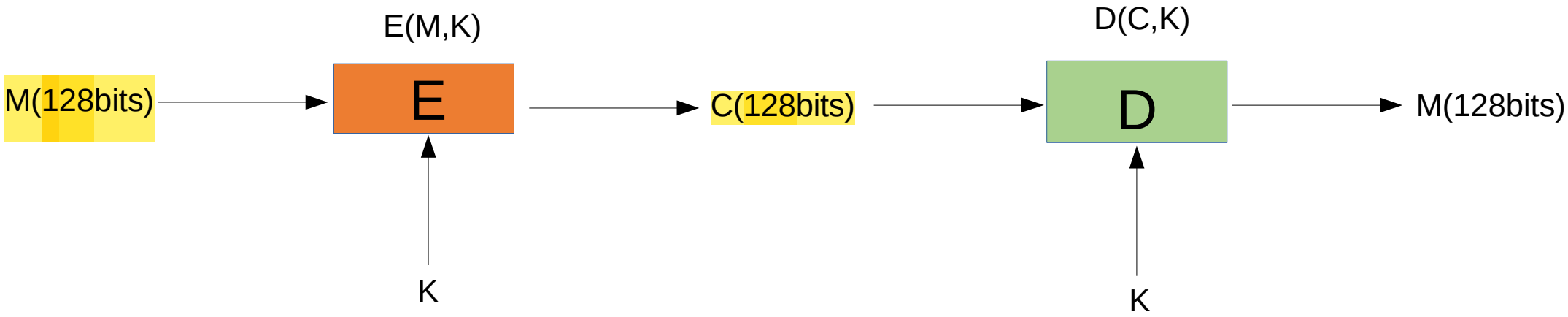
	DES	AES
Developed	1977	2000
Key Length	56 bits	128, 192, or 256 bits
Cipher Type	Symmetric block cipher	Symmetric block cipher
Block Size	64 bits	128 bits
Security	Proven inadequate	Considered secure

Advanced Encryption Standard (AES)

- AES is an encryption standard chosen by the National Institute of Standards and Technology(NIST), USA to protect classified information.
- It has been accepted world-wide as a desirable algorithm to encrypt sensitive data.
- AES, also called Rijndael is a symmetric block cipher that operates on 128-bit blocks.
- It is designed to be used with keys that are 128, 192, or 256 bits long, yielding ciphers known as AES-128, AES-192, and AES-256.



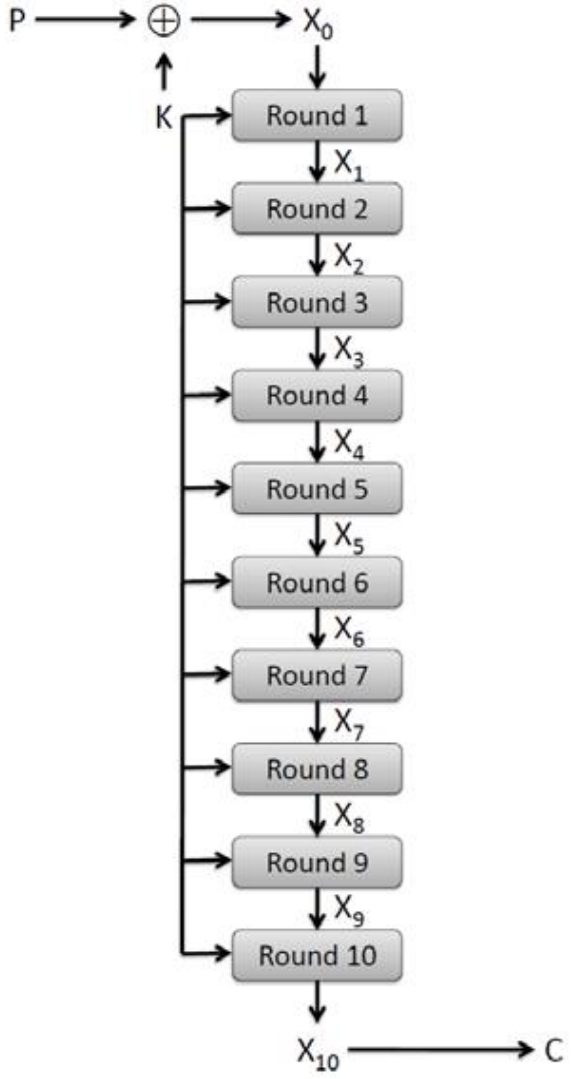
How Does it work?



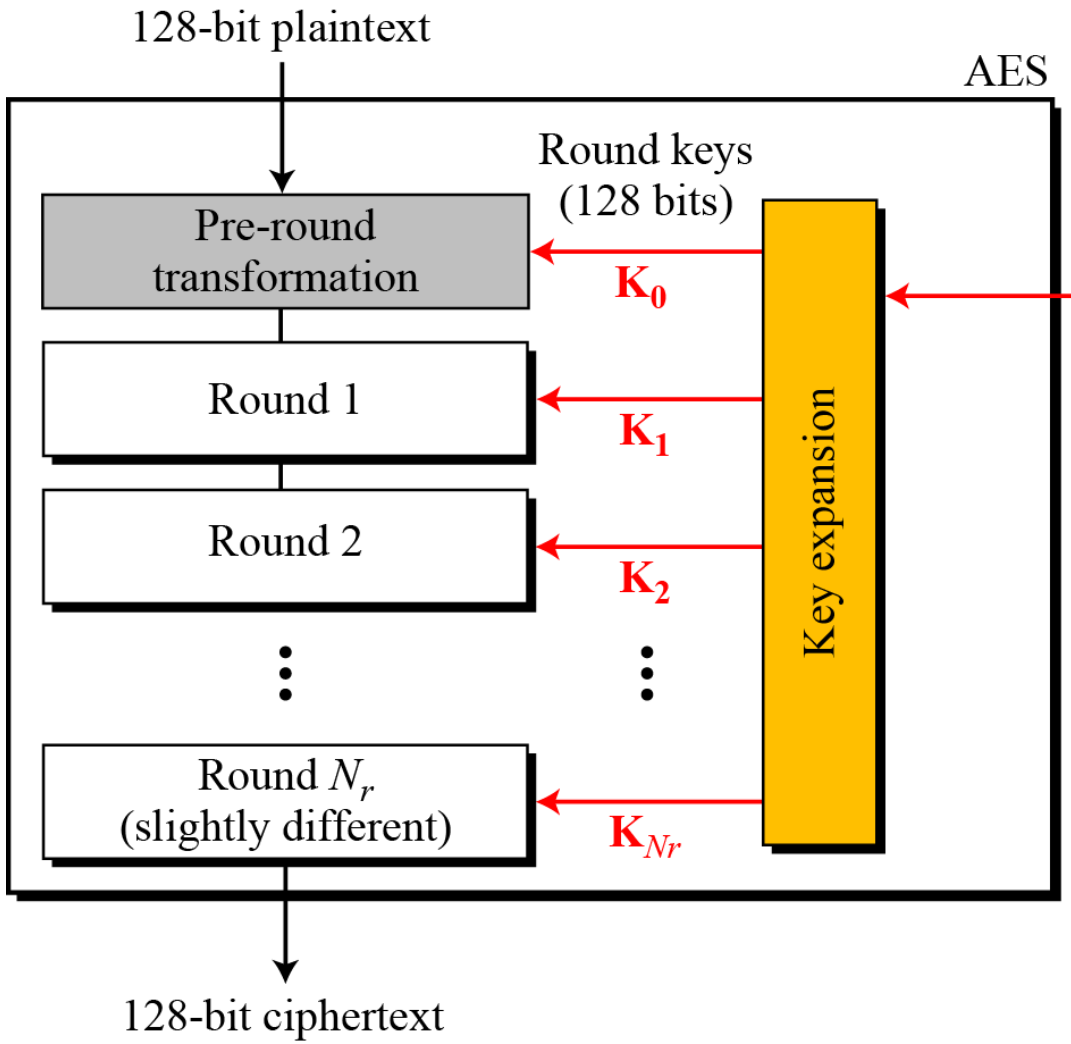
E = Encryption function for a symmetric block cipher
M = plaintext message of size 128 bits
C = ciphertext
K = Key of size 128 bits which is same for both encryption and decryption
D = Decryption function for symmetric block cipher

How Does it work?

- The 128-bits version of the AES encryption algorithm proceeds in 10 rounds.
- Each round performs an *invertible* transformation on a 128-bit array, called **state**.
- The initial state X_0 is the XOR of the plaintext P with the key K: X_0 = P XOR K.
- Round i (i = 1,2, …, 10) receives state X_{i-1} as input and produces state X_i.
- The ciphertext C is the output of the final round: C = X_{10}.



How Does it work?



Cipher key
(128, 192, or 256 bits)

Nr	Key size
10	128
12	192
14	256

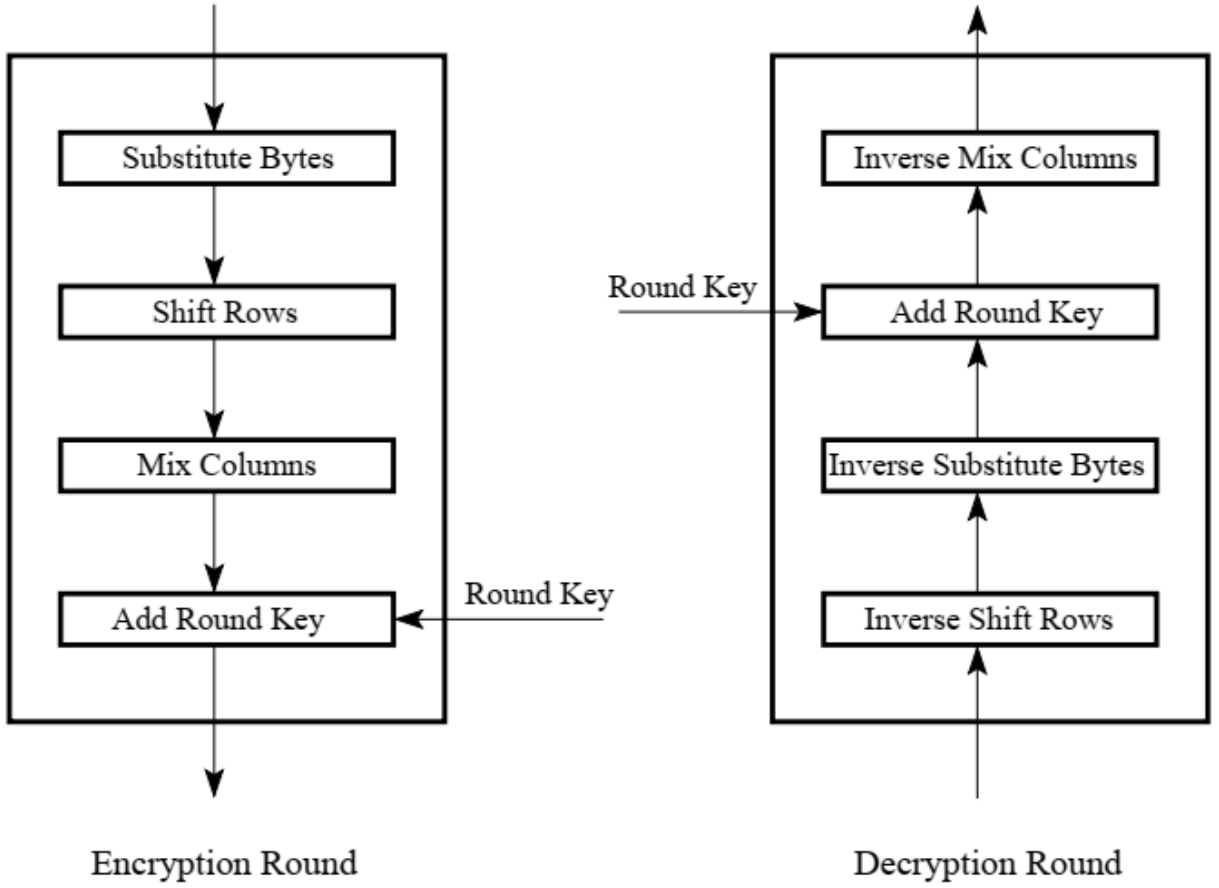
Relationship between number of rounds and cipher key size

How Does it work?

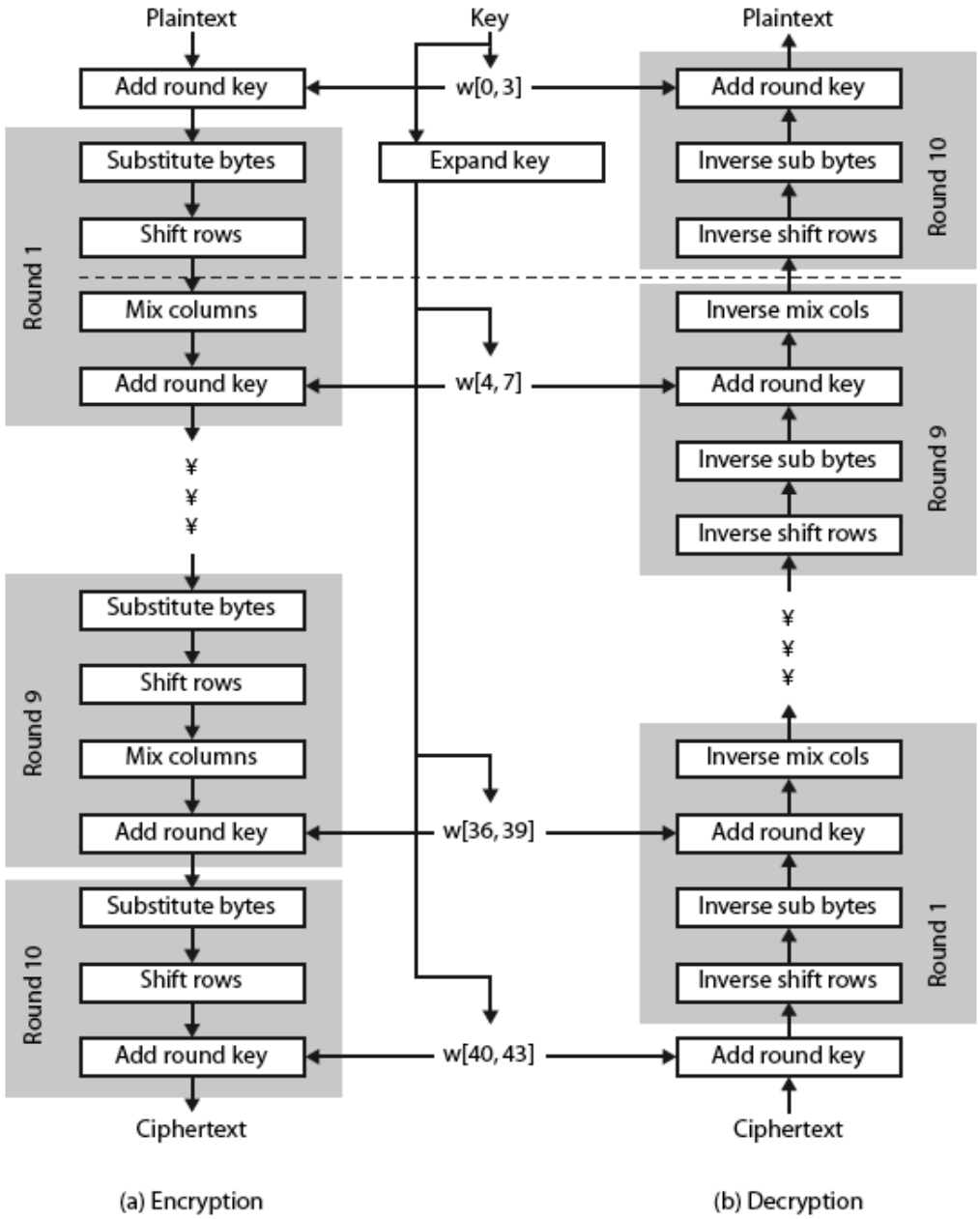
- AES basically repeats ***4 major functions per round*** ***(mostly)*** to encrypt data.
- It takes 128 bits block of data and a key and gives a ciphertext as output.

- The functions are:
  - **Substitute Bytes()**
  - **Shift Rows()**
  - **Mix Columns()**
  - **Add Round Key()**

How Does it work?



Block Diagram of AES

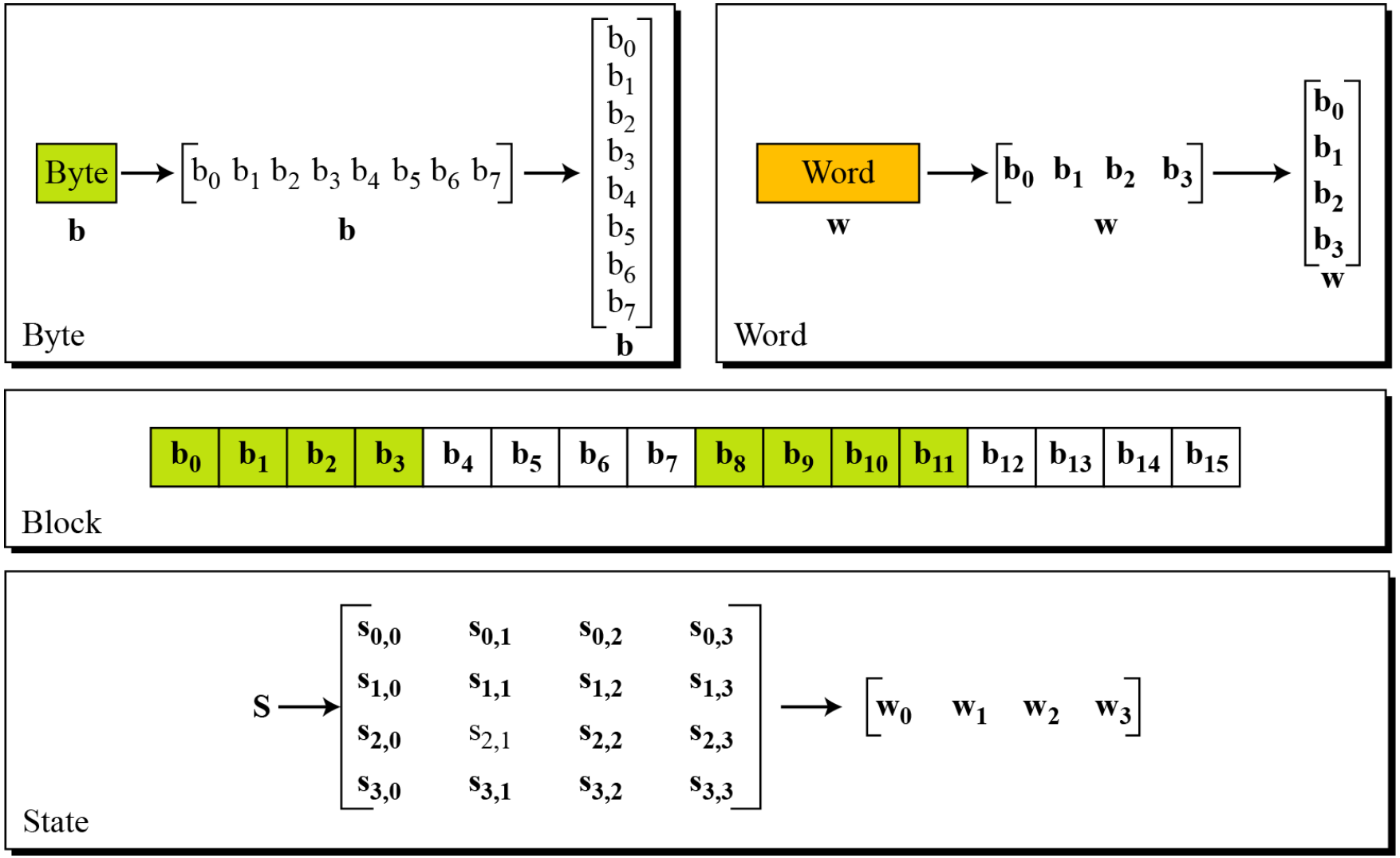


AES 128 bits Data Block

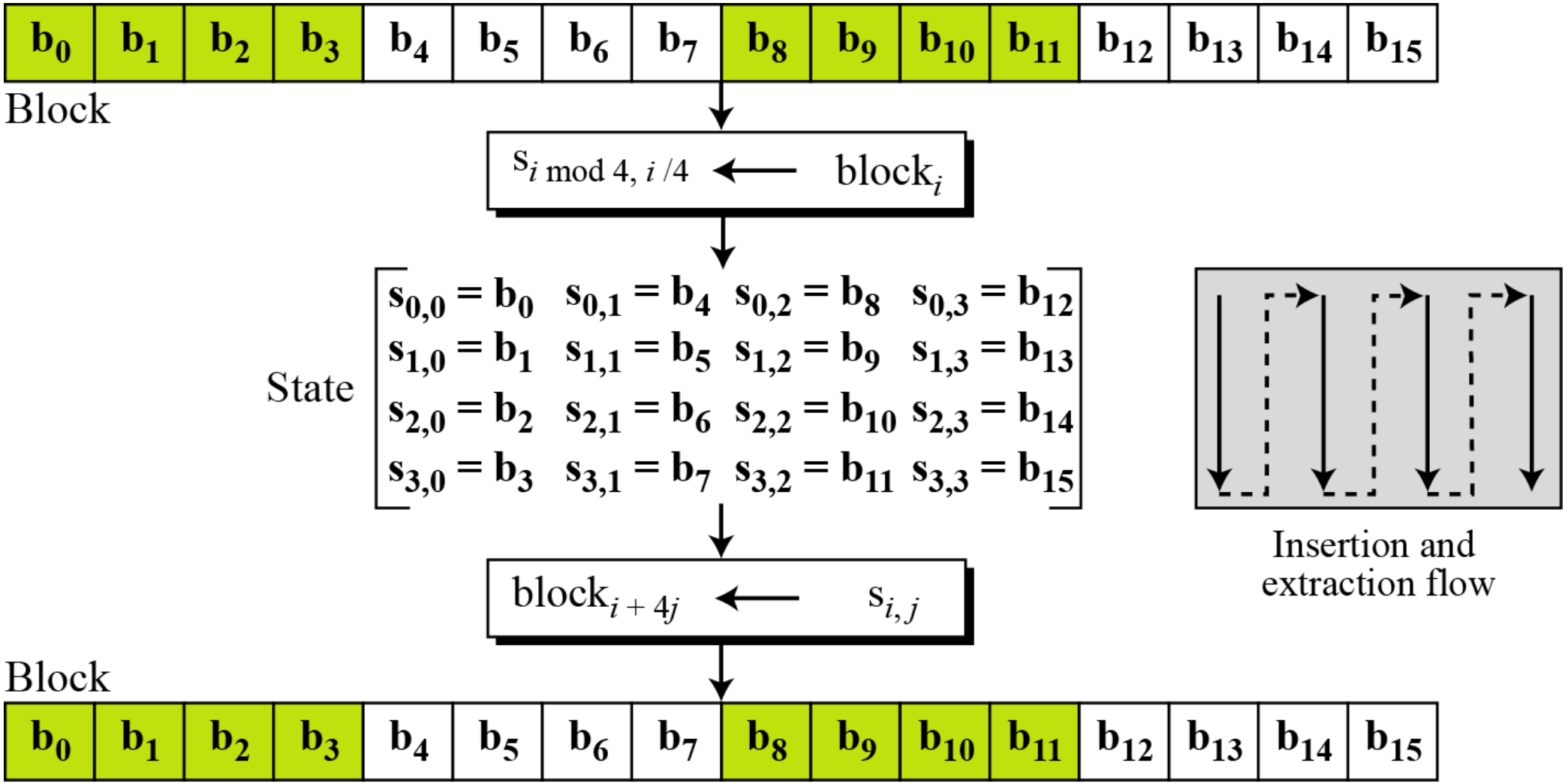
- AES operates on a 4 × 4 matrix of 4 column-wise order array of bytes, called the *state*.
- For instance, if there are 16 bytes, these bytes are represented as this two-dimensional array:

$$\begin{bmatrix} b_0 & b_4 & b_8 & b_{12} \\ b_1 & b_5 & b_9 & b_{13} \\ b_2 & b_6 & b_{10} & b_{14} \\ b_3 & b_7 & b_{11} & b_{15} \end{bmatrix}$$

Data units used in AES



Block-to-State and State-to-Block transformation



Insertion and extraction flow

Changing plaintext to state

Text	A	E	S	U	S	E	S	A	M	A	T	R	I	X	Z	Z
Hexadecimal	00	04	12	14	12	04	12	00	0C	00	13	11	08	23	19	19

$$\begin{bmatrix} 00 & 12 & 0C & 08 \\ 04 & 04 & 00 & 23 \\ 12 & 12 & 13 & 19 \\ 14 & 00 & 11 & 19 \end{bmatrix}$$ State

Analysis of Steps

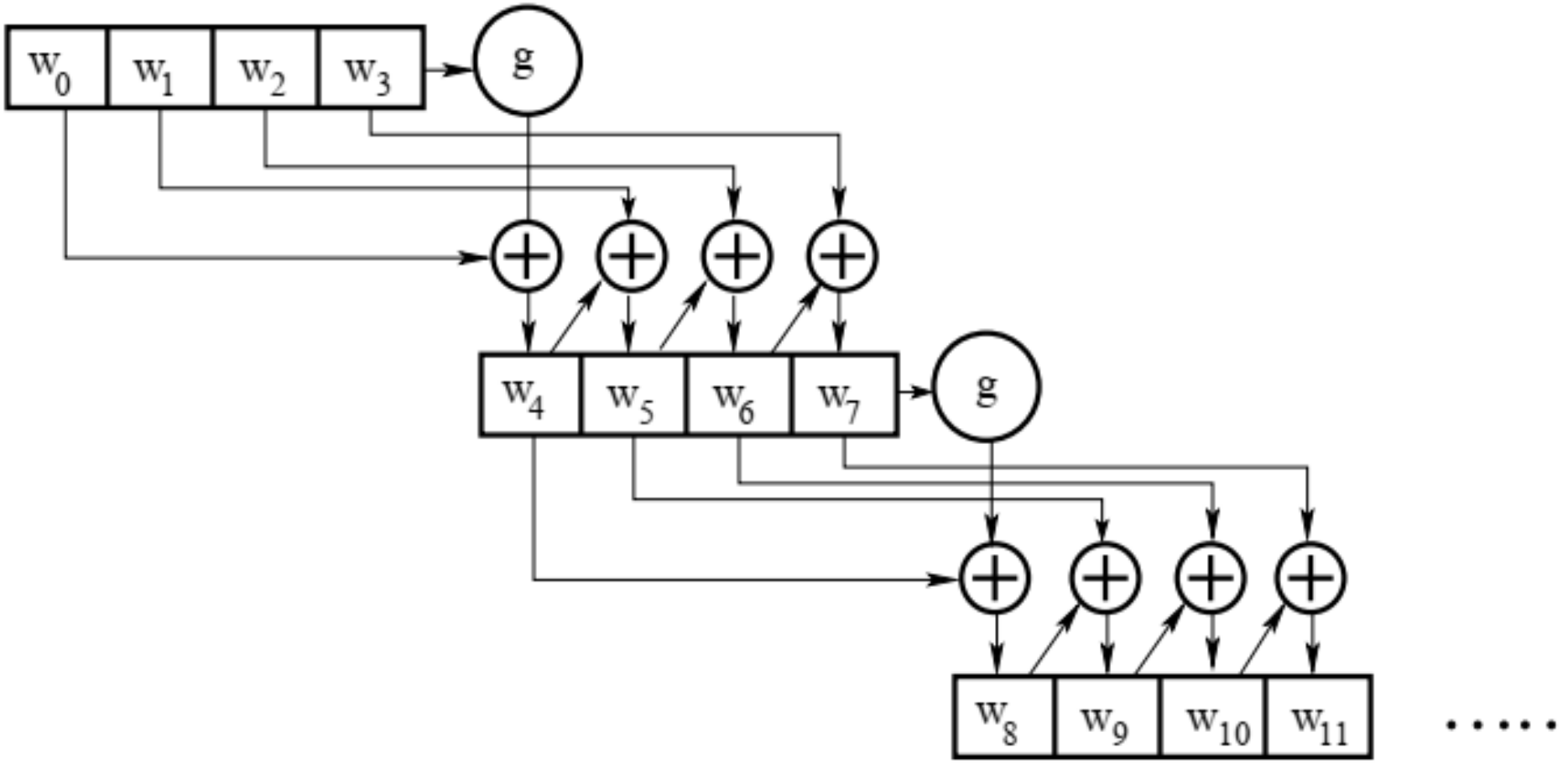
Key Expansions:

- In the key Expansion process the given 128 bits cipher key is stored in [4]x[4]= 16 bytes or 4 words matrix (16*8 = 128 bits) and then
- the four column words of the key matrix is expanded into a schedule of 44 words (44*4 = 176 bytes) resulting in 11 round keys (176/11 = 16 bytes or 128 bits per round).
- Number of round keys = $N_r + 1$.
  - Where N_r is the number of rounds (which is 10 in case of 128 bits key size)
- So here the round keys = 11.

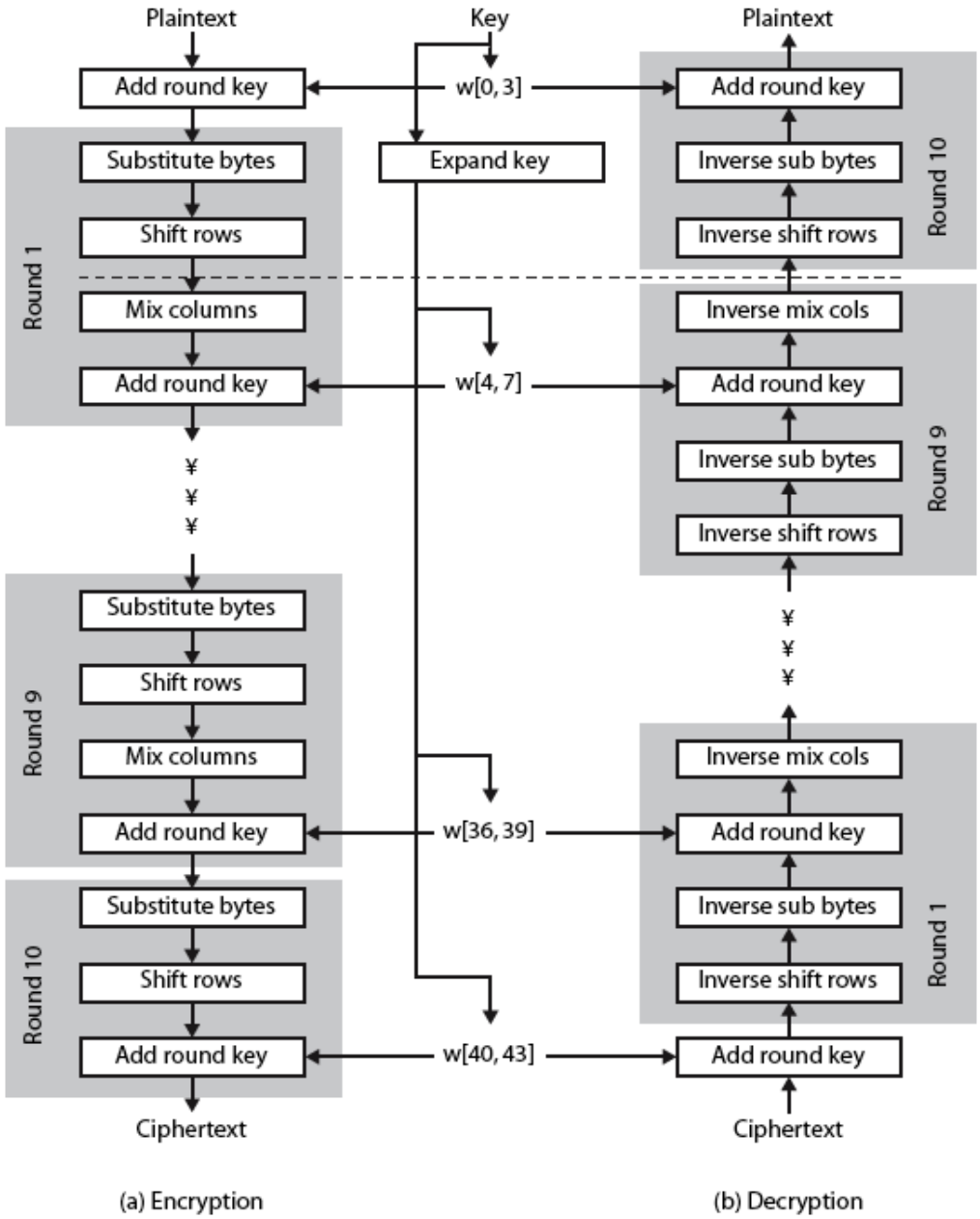
16 = 4x4

Analysis of Steps

Key Expansions:



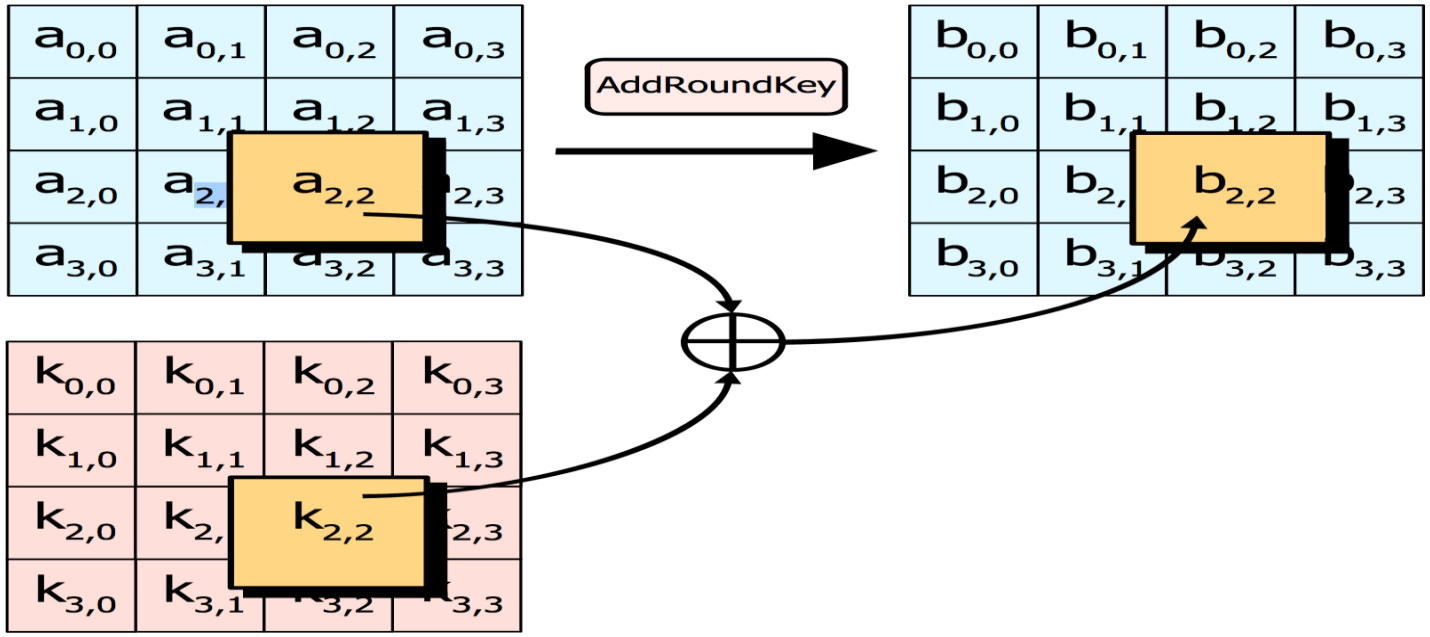
Block Diagram of AES



Analysis of Steps

Add round key:

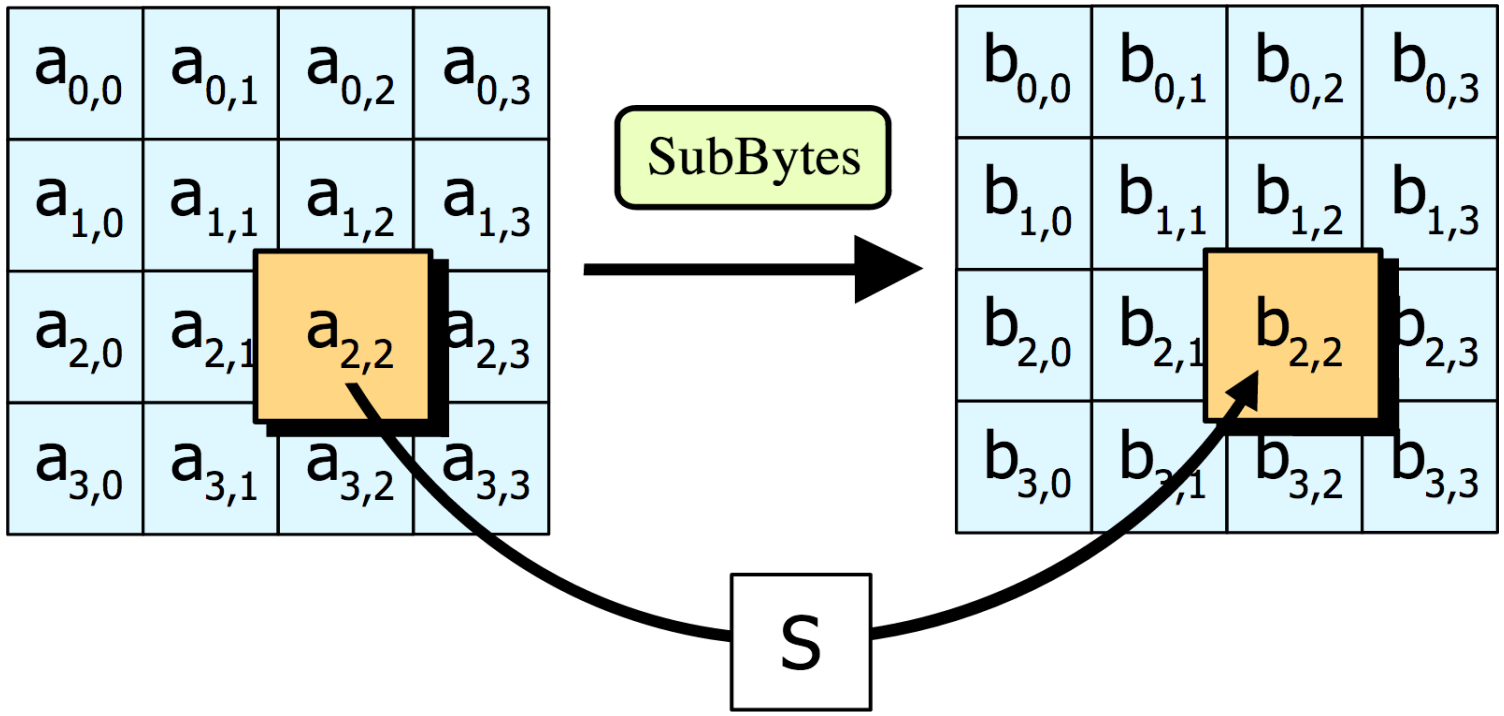
This is identical to the decryption side add round key transformation, because the XOR operation is its own inverse.



Analysis of Steps

Substitute Bytes:

Each element of the matrix is replaced by an element of **s-box** matrix.



Analysis of Steps

Substitute Bytes:

For an element {d1} corresponding value is {3e}

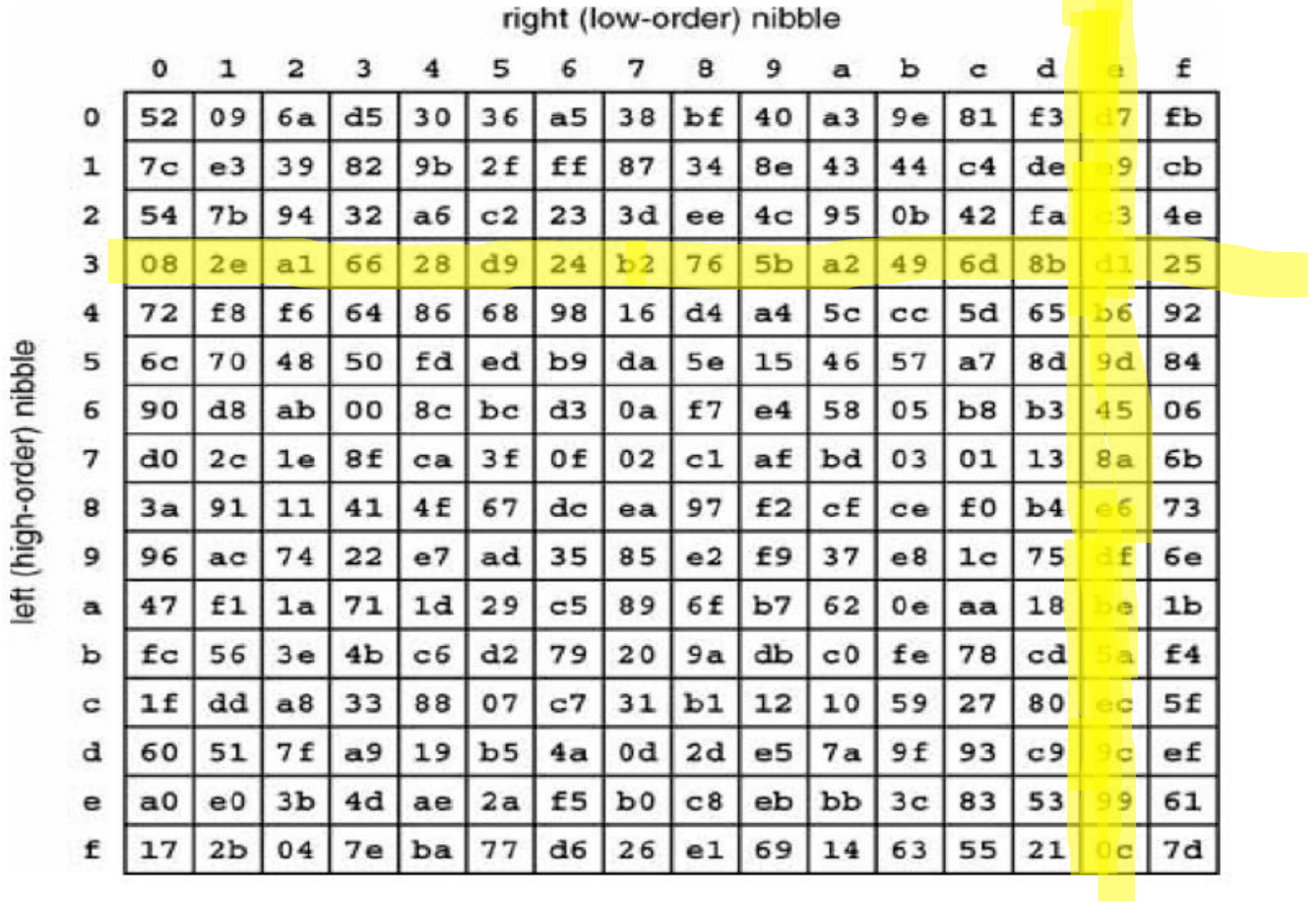
	x0	x1	x2	x3	x4	x5	x6	x7	x8	x9	xa	xb	xc	xd	xe	xf
0x	63	7c	77	7b	f2	6b	6f	c5	30	01	67	2b	fe	d7	ab	76
1x	ca	82	c9	7d	fa	59	47	f0	ad	d4	a2	af	9c	a4	72	c0
2x	b7	fd	93	26	36	3f	f7	cc	34	a5	e5	f1	71	d8	31	15
3x	04	c7	23	c3	18	96	05	9a	07	12	80	e2	eb	27	b2	75
4x	09	83	2c	1a	1b	6e	5a	a0	52	3b	d6	b3	29	e3	2f	84
5x	53	d1	00	ed	20	fc	b1	5b	6a	cb	be	39	4a	4c	58	cf
6x	d0	ef	aa	fb	43	4d	33	85	45	f9	02	7f	50	3c	9f	a8
7x	51	a3	40	8f	92	9d	38	f5	bc	b6	da	21	10	ff	f3	d2
8x	cd	0c	13	ec	5f	97	44	17	c4	a7	7e	3d	64	5d	19	73
9x	60	81	4f	dc	22	2a	90	88	46	ee	b8	14	de	5e	0b	db
ax	e0	32	3a	0a	49	06	24	5c	c2	d3	ac	62	91	95	e4	79
bx	e7	c8	37	6d	8d	d5	4e	a9	6c	56	f4	ea	65	7a	ae	08
cx	ba	78	25	2e	1c	a6	b4	c6	e8	dd	74	1f	4b	bd	8b	8a
dx	70	3e	b5	66	48	03	f6	0e	61	35	57	b9	86	c1	1d	9e
ex	e1	f8	98	11	69	d9	8e	94	9b	1e	87	e9	ce	55	28	df
fx	8c	a1	89	0d	bf	e6	42	68	41	99	2d	0f	b0	54	bb	16

Rijndael S-box

Analysis of Steps

Inverse Substitute Bytes:

For an element {3e} corresponding value is {d1}



left (high-order) nibble \ right (low-order) nibble	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
0	52	09	6a	d5	30	36	a5	38	bf	40	a3	9e	81	f3	d7	fb
1	7c	e3	39	82	9b	2f	ff	87	34	8e	43	44	c4	de	e9	cb
2	54	7b	94	32	a6	c2	23	3d	ee	4c	95	0b	42	fa	c3	4e
3	08	2e	a1	66	28	d9	24	b2	76	5b	a2	49	6d	8b	d1	25
4	72	f8	f6	64	86	68	98	16	d4	a4	5c	cc	5d	65	b6	92
5	6c	70	48	50	fd	ed	b9	da	5e	15	46	57	a7	8d	9d	84
6	90	d8	ab	00	8c	bc	d3	0a	f7	e4	58	05	b8	b3	45	06
7	d0	2c	1e	8f	ca	3f	0f	02	c1	af	bd	03	01	13	8a	6b
8	3a	91	11	41	4f	67	dc	ea	97	f2	cf	ce	f0	b4	e6	73
9	96	ac	74	22	e7	ad	35	85	e2	f9	37	e8	1c	75	df	6e
a	47	f1	1a	71	1d	29	c5	89	6f	b7	62	0e	aa	18	be	1b
b	fc	56	3e	4b	c6	d2	79	20	9a	db	c0	fe	78	cd	5a	f4
c	1f	dd	a8	33	88	07	c7	31	b1	12	10	59	27	80	ec	5f
d	60	51	7f	a9	19	b5	4a	0d	2d	e5	7a	9f	93	c9	9c	ef
e	a0	e0	3b	4d	ae	2a	f5	b0	c8	eb	bb	3c	83	53	99	61
f	17	2b	04	7e	ba	77	d6	26	e1	69	14	63	55	21	0c	7d

Analysis of Steps

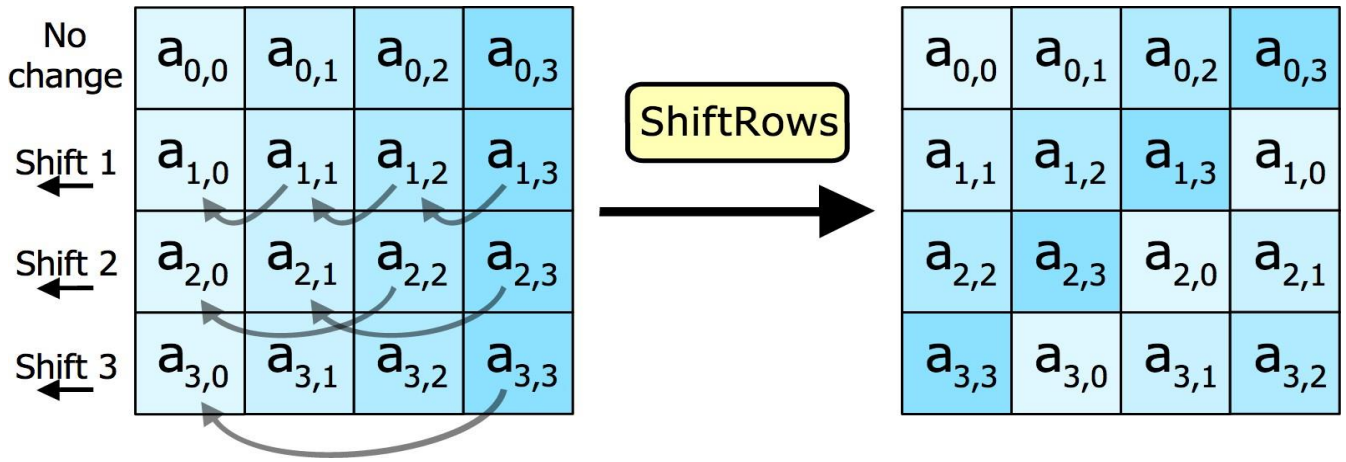
Substitute Bytes:

- The S-box is a special lookup table which is constructed by ***Galois fields***.
- The Generating function used in this algorithm is GF(2^8)
  - i.e. 256 values are possible.
- The elements of the sbox are written in hexadecimal system.

Analysis of Steps

Shift Rows:

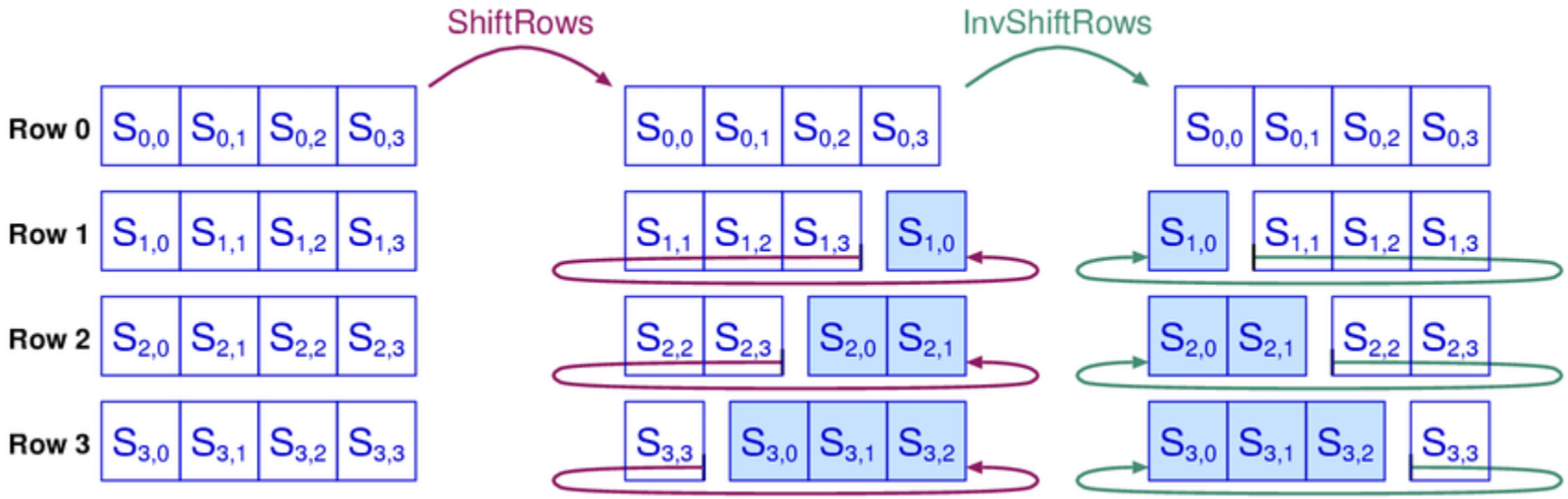
- In this step rows, of the block are cylindrically shifted in left direction.
- The first row is untouched, the second by one shift, third by two and fourth by three.



Analysis of Steps

Inverse Shift Rows:

It performs the circular shifts in the opposite direction (right) for each of the last three rows, with (first row is untouched), the second by one-byte circular right shift, third by two and fourth by three.



Analysis of Steps

Mix Columns:

- This is the most important part of the algorithm.
- It causes the flip of bits to spread all over the block.
- In this step the block is multiplied with a *fixed matrix*.
- The multiplication is *field multiplication* in ***Galois field***.
- For each row there are 16 multiplication, 12 XORs and a 4 bytes output.

Analysis of Steps

Mix Columns:

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

$$\begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix} \begin{bmatrix} s_{0,0} & s_{0,1} & s_{0,2} & s_{0,3} \\ s_{1,0} & s_{1,1} & s_{1,2} & s_{1,3} \\ s_{2,0} & s_{2,1} & s_{2,2} & s_{2,3} \\ s_{3,0} & s_{3,1} & s_{3,2} & s_{3,3} \end{bmatrix} = \begin{bmatrix} s'_{0,0} & s'_{0,1} & s'_{0,2} & s'_{0,3} \\ s'_{1,0} & s'_{1,1} & s'_{1,2} & s'_{1,3} \\ s'_{2,0} & s'_{2,1} & s'_{2,2} & s'_{2,3} \\ s'_{3,0} & s'_{3,1} & s'_{3,2} & s'_{3,3} \end{bmatrix} \Rightarrow$$

$$s'_{0,j} = (2 \bullet s_{0,j}) \oplus (3 \bullet s_{1,j}) \oplus s_{2,j} \oplus s_{3,j}$$
$$s'_{1,j} = s_{0,j} \oplus (2 \bullet s_{1,j}) \oplus (3 \bullet s_{2,j}) \oplus s_{3,j}$$
$$s'_{2,j} = s_{0,j} \oplus s_{1,j} \oplus (2 \bullet s_{2,j}) \oplus (3 \bullet s_{3,j})$$
$$s'_{3,j} = (3 \bullet s_{0,j}) \oplus s_{1,j} \oplus s_{2,j} \oplus (2 \bullet s_{3,j})$$

Predefine Matrix **State Array** **New State Array**

2	3	1	1
1	2	3	1
1	1	2	3
3	1	1	2

*

87	F2	4D	97
6E	4C	90	EC
46	E7	4A	C3
A6	8C	D8	95

$\Rightarrow$

$$\begin{aligned}
&(\{02\} \cdot \{87\}) \oplus (\{03\} \cdot \{6E\}) \oplus \{46\} \oplus \{A6\} = \{47\} \\
&\{87\} \oplus (\{02\} \cdot \{6E\}) \oplus (\{03\} \cdot \{46\}) \oplus \{A6\} = \{37\} \\
&\{87\} \oplus \{6E\} \oplus (\{02\} \cdot \{46\}) \oplus (\{03\} \cdot \{A6\}) = \{94\} \\
&(\{03\} \cdot \{87\}) \oplus \{6E\} \oplus \{46\} \oplus (\{02\} \cdot \{A6\}) = \{ED\}
\end{aligned}$$

$\Rightarrow$

47	40	A3	4C
37	D4	70	9F
94	E4	3A	42
ED	A5	A6	BC

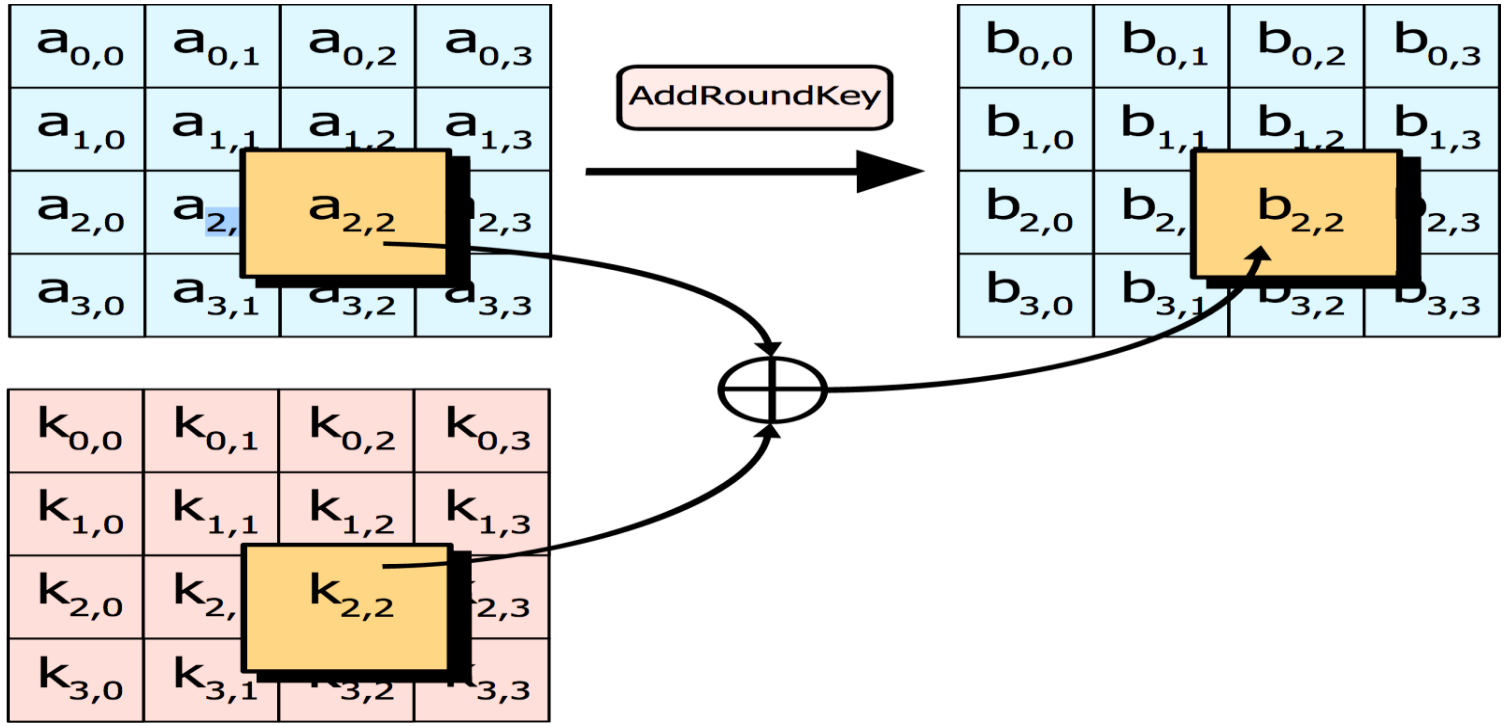
Analysis of Steps

Inverse Mix Columns:

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

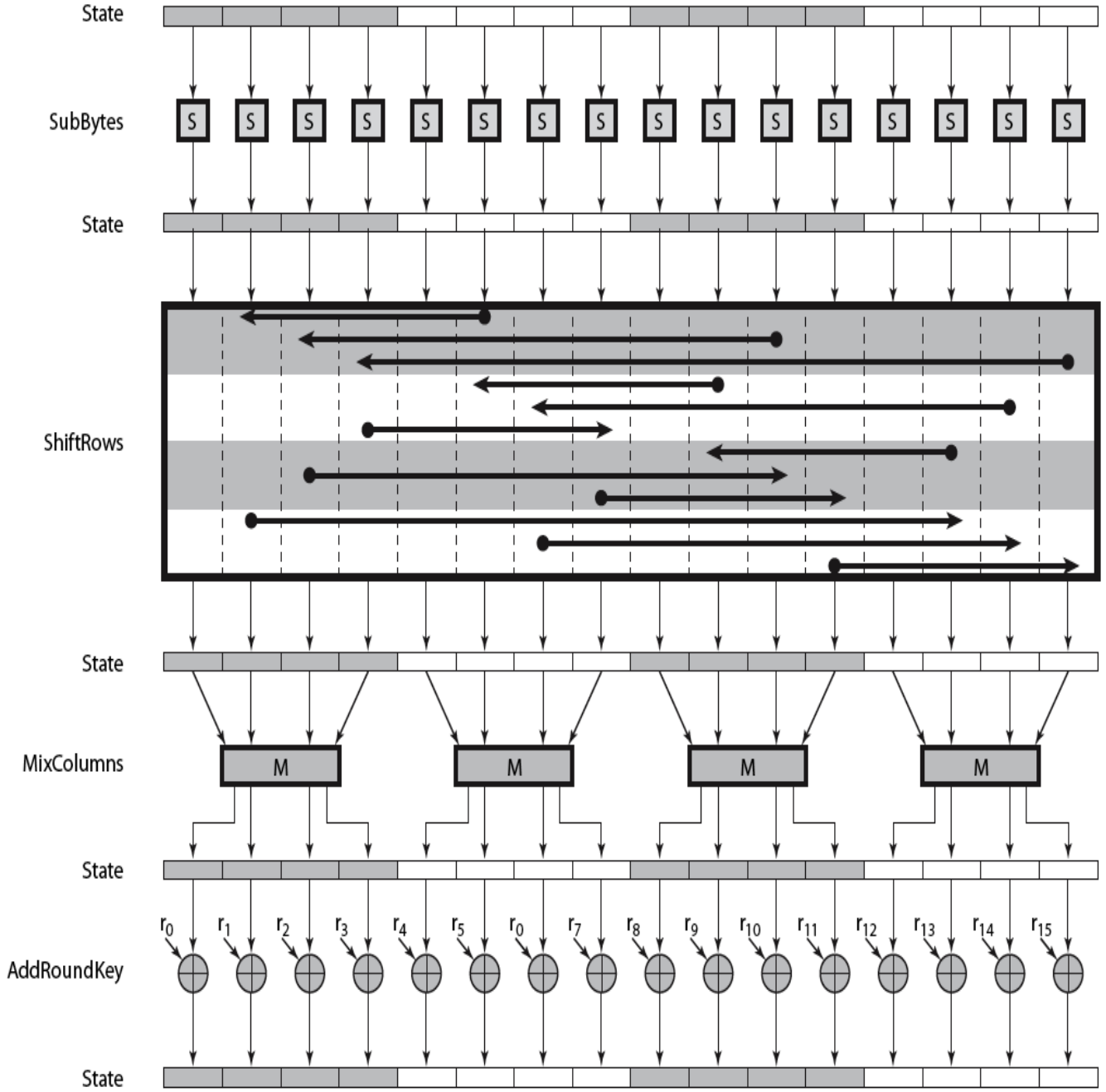
Analysis of Steps

Add round key:



AES Round

View of all the internal details of the AES round, showing how each byte of the state is manipulated.



AES Encryption Simulation

AES Example - Input (128 bit key and message)

Key in English: Thats my Kung Fu (16 ASCII characters, 1 byte each)

Translation into Hex:

T	h	a	t	s		m	y		K	u	n	g		F	u
54	68	61	74	73	20	6D	79	20	4B	75	6E	67	20	46	75

Key in Hex (128 bits): 54 68 61 74 73 20 6D 79 20 4B 75 6E 67 20 46 75

Plaintext in English: Two One Nine Two (16 ASCII characters, 1 byte each)

Translation into Hex:

T	w	o		O	n	e		N	i	n	e		T	w	o
54	77	6F	20	4F	6E	65	20	4E	69	6E	65	20	54	77	6F

Plaintext in Hex (128 bits): 54 77 6F 20 4F 6E 65 20 4E 69 6E 65 20 54 77 6F

AES Example - The first Roundkey

- Key in Hex (128 bits): 54 68 61 74 73 20 6D 79 20 4B 75 6E 67 20 46 75
- $w[0] = (54, 68, 61, 74), w[1] = (73, 20, 6D, 79), w[2] = (20, 4B, 75, 6E), w[3] = (67, 20, 46, 75)$
- $g(w[3])$:
  - circular byte left shift of $w[3]$: $(20, 46, 75, 67)$
  - Byte Substitution (S-Box): $(B7, 5A, 9D, 85)$
  - Adding round constant $(01, 00, 00, 00)$ gives: $g(w[3]) = (B6, 5A, 9D, 85)$
- $w[4] = w[0] \oplus g(w[3]) = (E2, 32, FC, F1)$:

0101 0100 1011 0110	0110 1000 0101 1010	0110 0001 1001 1101	0111 0100 1000 0101
1110 0010 E2	0011 0010 32	1111 1100 FC	1111 0001 F1

Table 7.4 *RCon constants*

Round	*Constant (RCon)*	*Round*	*Constant (RCon)*
1	$(\underline{\mathbf{01}}\ 00\ 00\ 00)_{16}$	6	$(\underline{\mathbf{20}}\ 00\ 00\ 00)_{16}$
2	$(\underline{\mathbf{02}}\ 00\ 00\ 00)_{16}$	7	$(\underline{\mathbf{40}}\ 00\ 00\ 00)_{16}$
3	$(\underline{\mathbf{04}}\ 00\ 00\ 00)_{16}$	8	$(\underline{\mathbf{80}}\ 00\ 00\ 00)_{16}$
4	$(\underline{\mathbf{08}}\ 00\ 00\ 00)_{16}$	9	$(\underline{\mathbf{1B}}\ 00\ 00\ 00)_{16}$
5	$(\underline{\mathbf{10}}\ 00\ 00\ 00)_{16}$	10	$(\underline{\mathbf{36}}\ 00\ 00\ 00)_{16}$

- $w[5] = w[4] \oplus w[1] = (91, 12, 91, 88)$, $w[6] = w[5] \oplus w[2] = (B1, 59, E4, E6)$,
  $w[7] = w[6] \oplus w[3] = (D6, 79, A2, 93)$
- first roundkey: E2 32 FC F1 91 12 91 88 B1 59 E4 E6 D6 79 A2 93

AES Example - All RoundKeys

- Round 0: 54 68 61 74 73 20 6D 79 20 4B 75 6E 67 20 46 75
- Round 1: E2 32 FC F1 91 12 91 88 B1 59 E4 E6 D6 79 A2 93
- Round 2: 56 08 20 07 C7 1A B1 8F 76 43 55 69 A0 3A F7 FA
- Round 3: D2 60 0D E7 15 7A BC 68 63 39 E9 01 C3 03 1E FB
- Round 4: A1 12 02 C9 B4 68 BE A1 D7 51 57 A0 14 52 49 5B
- Round 5: B1 29 3B 33 05 41 85 92 D2 10 D2 32 C6 42 9B 69
- Round 6: BD 3D C2 B7 B8 7C 47 15 6A 6C 95 27 AC 2E 0E 4E
- Round 7: CC 96 ED 16 74 EA AA 03 1E 86 3F 24 B2 A8 31 6A
- Round 8: 8E 51 EF 21 FA BB 45 22 E4 3D 7A 06 56 95 4B 6C
- Round 9: BF E2 BF 90 45 59 FA B2 A1 64 80 B4 F7 F1 CB D8
- Round 10: 28 FD DE F8 6D A4 24 4A CC C0 A4 FE 3B 31 6F 26

AES Example - Add Roundkey, Round 0

- State Matrix and Roundkey No.0 Matrix:

$$\begin{pmatrix} 54 & 4F & 4E & 20 \\ 77 & 6E & 69 & 54 \\ 6F & 65 & 6E & 77 \\ 20 & 20 & 65 & 6F \end{pmatrix} \qquad \begin{pmatrix} 54 & 73 & 20 & 67 \\ 68 & 20 & 4B & 20 \\ 61 & 6D & 75 & 46 \\ 74 & 79 & 6E & 75 \end{pmatrix}$$

- XOR the corresponding entries, e.g., $69 \oplus 4B = 22$

$$\begin{array}{r} 0110\ 1001 \\ 0100\ 1011 \\ \hline 0010\ 0010 \end{array}$$

- the new State Matrix is

$$\begin{pmatrix} 00 & 3C & 6E & 47 \\ 1F & 4E & 22 & 74 \\ 0E & 08 & 1B & 31 \\ 54 & 59 & 0B & 1A \end{pmatrix}$$

AES Example - Round 1, Substitution Bytes

- current State Matrix is

$$\begin{pmatrix} 00 & 3C & 6E & 47 \\ 1F & 4E & 22 & 74 \\ 0E & 08 & 1B & 31 \\ 54 & 59 & 0B & 1A \end{pmatrix}$$

- substitute each entry (byte) of current state matrix by corresponding entry in AES S-Box
- for instance: byte 6E is substituted by entry of S-Box in row 6 and column E, i.e., by 9F
- this leads to new State Matrix

$$\begin{pmatrix} 63 & EB & 9F & A0 \\ C0 & 2F & 93 & 92 \\ AB & 30 & AF & C7 \\ 20 & CB & 2B & A2 \end{pmatrix}$$

- this non-linear layer is for resistance to differential and linear cryptanalysis attacks

AES Example - Round 1, Shift Row

- the current State Matrix is

$$\begin{pmatrix} 63 & EB & 9F & A0 \\ C0 & 2F & 93 & 92 \\ AB & 30 & AF & C7 \\ 20 & CB & 2B & A2 \end{pmatrix}$$

- four rows are shifted cyclically to the left by offsets of 0,1,2, and 3
- the new State Matrix is

$$\begin{pmatrix} 63 & EB & 9F & A0 \\ 2F & 93 & 92 & C0 \\ AF & C7 & AB & 30 \\ A2 & 20 & CB & 2B \end{pmatrix}$$

- this linear mixing step causes diffusion of the bits over multiple rounds

AES Example - Round 1, Mix Column

- Mix Column multiplies fixed matrix against current State Matrix:

$$\begin{pmatrix} 02\,03\,01\,01 \\ 01\,02\,03\,01 \\ 01\,01\,02\,03 \\ 03\,01\,01\,02 \end{pmatrix} \begin{pmatrix} 63 & EB & 9F & A0 \\ 2F & 93 & 92 & C0 \\ AF & C7 & AB & 30 \\ A2 & 20 & CB & 2B \end{pmatrix} = \begin{pmatrix} BA & 84 & E8 & 1B \\ 75 & A4 & 8D & 40 \\ F4 & 8D & 06 & 7D \\ 7A & 32 & 0E & 5D \end{pmatrix}$$

- entry BA is result of $(02 \bullet 63) \oplus (03 \bullet 2F) \oplus (01 \bullet AF) \oplus (01 \bullet A2)$:
  - $02 \bullet 63 = 00000010 \bullet 01100011 = 11000110$
  - $03 \bullet 2F = (02 \bullet 2F) \oplus 2F = (00000010 \bullet 00101111) \oplus 00101111 = 01110001$
  - $01 \bullet AF = AF = 10101111$ and $01 \bullet A2 = A2 = 10100010$
  - hence

$$\begin{array}{r} 11000110 \\ 01110001 \\ 10101111 \\ 10100010 \\ \hline 10111010 \end{array}$$

AES Example - Add Roundkey, Round 1

- State Matrix and Roundkey No.1 Matrix:

$$\begin{pmatrix} BA & 84 & E8 & 1B \\ 75 & A4 & 8D & 40 \\ F4 & 8D & 06 & 7D \\ 7A & 32 & 0E & 5D \end{pmatrix} \qquad \begin{pmatrix} E2 & 91 & B1 & D6 \\ 32 & 12 & 59 & 79 \\ FC & 91 & E4 & A2 \\ F1 & 88 & E6 & 93 \end{pmatrix}$$

- XOR yields new State Matrix

$$\begin{pmatrix} 58 & 15 & 59 & CD \\ 47 & B6 & D4 & 39 \\ 08 & 1C & E2 & DF \\ 8B & BA & E8 & CE \end{pmatrix}$$

- AES output after Round 1: 58 47 08 8B 15 B6 1C BA 59 D4 E2 E8 CD 39 DF CE

AES Example - Round 2

- after Substitute Byte and after Shift Rows:

$$\begin{pmatrix} 6A & 59 & CB & BD \\ A0 & 4E & 48 & 12 \\ 30 & 9C & 98 & 9E \\ 3D & F4 & 9B & 8B \end{pmatrix} \qquad \begin{pmatrix} 6A & 59 & CB & BD \\ 4E & 48 & 12 & A0 \\ 98 & 9E & 30 & 9B \\ 8B & 3D & F4 & 9B \end{pmatrix}$$

- after Mixcolumns and after Roundkey:

$$\begin{pmatrix} 15 & C9 & 7F & 9D \\ CE & 4D & 4B & C2 \\ 89 & 71 & BE & 88 \\ 65 & 47 & 97 & CD \end{pmatrix} \qquad \begin{pmatrix} 43 & 0E & 09 & 3D \\ C6 & 57 & 08 & F8 \\ A9 & C0 & EB & 7F \\ 62 & C8 & FE & 37 \end{pmatrix}$$

AES Example - Round 3

- after Substitute Byte and after Shift Rows:

$$\begin{pmatrix} 1A & AB & 01 & 27 \\ B4 & 5B & 30 & 41 \\ D3 & BA & E9 & D2 \\ AA & E8 & BB & 9A \end{pmatrix} \qquad \begin{pmatrix} 1A & AB & 01 & 27 \\ 5B & 30 & 41 & B4 \\ E9 & D2 & D3 & BA \\ A9 & AA & E8 & BB \end{pmatrix}$$

- after Mixcolumns and after Roundkey:

$$\begin{pmatrix} AA & 65 & FA & 88 \\ 16 & 0C & 05 & 3A \\ 3D & C1 & DE & 2A \\ B3 & 4B & 5A & 0A \end{pmatrix} \qquad \begin{pmatrix} 78 & 70 & 99 & 4B \\ 76 & 76 & 3C & 39 \\ 30 & 7D & 37 & 34 \\ 54 & 23 & 5B & F1 \end{pmatrix}$$

AES Example - Round 4

- after Substitute Byte and after Shift Rows:

$$\begin{pmatrix} BC & 51 & EE & B3 \\ 38 & 38 & EB & 12 \\ 04 & FF & 9A & 18 \\ 20 & 26 & 39 & A1 \end{pmatrix} \qquad \begin{pmatrix} BC & 51 & EE & B3 \\ 38 & EB & 12 & 38 \\ 9A & 18 & 04 & FF \\ A1 & 20 & 26 & 39 \end{pmatrix}$$

- after Mixcolumns and after Roundkey:

$$\begin{pmatrix} 10 & BC & D3 & F3 \\ D8 & 94 & E0 & E0 \\ 53 & EA & 9E & 25 \\ 24 & 40 & 73 & 7B \end{pmatrix} \qquad \begin{pmatrix} B1 & 08 & 04 & E7 \\ CA & FC & B1 & B2 \\ 51 & 54 & C9 & 6C \\ ED & E1 & D3 & 20 \end{pmatrix}$$

AES Example - Round 5

- after Substitute Byte and after Shift Rows:

$$\begin{pmatrix} C8 & 30 & F2 & 94 \\ 74 & B0 & C8 & 37 \\ D1 & 20 & DD & 50 \\ 55 & F8 & 66 & B7 \end{pmatrix} \qquad \begin{pmatrix} C8 & 30 & F2 & 94 \\ B0 & C8 & 37 & 74 \\ DD & 50 & D1 & 20 \\ B7 & 55 & F8 & 66 \end{pmatrix}$$

- after Mixcolumns and after Roundkey:

$$\begin{pmatrix} 2A & 26 & 8F & E9 \\ 78 & 1E & 0C & 7A \\ 1B & A7 & 6F & 0A \\ 5B & 62 & 00 & 3F \end{pmatrix} \qquad \begin{pmatrix} 9B & 23 & 5D & 2F \\ 51 & 5F & 1C & 38 \\ 20 & 22 & BD & 91 \\ 68 & F0 & 32 & 56 \end{pmatrix}$$

AES Example - Round 6

- after Substitute Byte and after Shift Rows:

$$\begin{pmatrix} 14 & 26 & 4C & 15 \\ D1 & CF & 9C & 07 \\ B7 & 93 & 7A & 81 \\ 45 & 8C & 23 & B1 \end{pmatrix} \qquad \begin{pmatrix} 14 & 26 & 4C & 15 \\ CF & 9C & 07 & D1 \\ 7A & 81 & B7 & 93 \\ B1 & 45 & 8C & 23 \end{pmatrix}$$

- after Mixcolumns and after Roundkey:

$$\begin{pmatrix} A9 & 37 & AA & F2 \\ AE & D8 & 0C & 21 \\ E7 & 6C & B1 & 9C \\ F0 & FD & 67 & 3B \end{pmatrix} \qquad \begin{pmatrix} 14 & 8F & C0 & 5E \\ 93 & A4 & 60 & 0F \\ 25 & 2B & 24 & 92 \\ 77 & E8 & 40 & 75 \end{pmatrix}$$

AES Example - Round 7

- after Substitute Byte and after Shift Rows:

$$\begin{pmatrix} FA & 73 & BA & 58 \\ DC & 49 & D0 & 76 \\ 3F & F1 & 36 & 4F \\ F5 & 9B & 09 & 9D \end{pmatrix} \qquad \begin{pmatrix} FA & 73 & BA & 58 \\ 49 & D0 & 76 & DC \\ 36 & 4F & 3F & F1 \\ 9D & F5 & 9B & 09 \end{pmatrix}$$

- after Mixcolumns and after Roundkey:

$$\begin{pmatrix} 9F & 37 & 51 & 37 \\ AF & EC & 8C & FA \\ 63 & 39 & 04 & 66 \\ 4B & FB & B1 & D7 \end{pmatrix} \qquad \begin{pmatrix} 53 & 43 & 4F & 85 \\ 39 & 06 & 0A & 52 \\ 8E & 93 & 3B & 57 \\ 5D & F8 & 95 & BD \end{pmatrix}$$

AES Example - Round 8

- after Substitute Byte and after Shift Rows:

$$\begin{pmatrix} ED & 1A & 84 & 97 \\ 12 & 6F & 67 & 00 \\ 19 & DC & E2 & 5B \\ 4C & 41 & 2A & 7A \end{pmatrix} \qquad \begin{pmatrix} ED & 1A & 84 & 97 \\ 6F & 67 & 00 & 12 \\ E2 & 5B & 19 & DC \\ 7A & 4C & 41 & 2A \end{pmatrix}$$

- after Mixcolumns and after Roundkey:

$$\begin{pmatrix} E8 & 8A & 4B & F5 \\ 74 & 75 & EE & E6 \\ D3 & 1F & 75 & 58 \\ 55 & 8A & 0C & 38 \end{pmatrix} \qquad \begin{pmatrix} 66 & 70 & AF & A3 \\ 25 & CE & D3 & 73 \\ 3C & 5A & 0F & 13 \\ 74 & A8 & 0A & 54 \end{pmatrix}$$

AES Example - Round 9

- after Substitute Byte and after Shift Rows:

$$\begin{pmatrix} 33 & 51 & 79 & 0A \\ 3F & 8B & 66 & 8F \\ EB & BE & 76 & 7D \\ 92 & C2 & 67 & 20 \end{pmatrix} \qquad \begin{pmatrix} 33 & 51 & 79 & 0A \\ 8B & 66 & 8F & 3F \\ 76 & 7D & EB & BE \\ 20 & 92 & C2 & 67 \end{pmatrix}$$

- after Mixcolumns and after Roundkey:

$$\begin{pmatrix} B6 & E7 & 51 & 8C \\ 84 & 88 & 98 & CA \\ 34 & 60 & 66 & FB \\ E8 & D7 & 70 & 51 \end{pmatrix} \qquad \begin{pmatrix} 09 & A2 & F0 & 7B \\ 66 & D1 & FC & 3B \\ 8B & 9A & E6 & 30 \\ 78 & 65 & C4 & 89 \end{pmatrix}$$

AES Example - Round 10

- after Substitute Byte and after Shift Rows:

$$\begin{pmatrix} 01 & 3A & 8C & 21 \\ 33 & 3E & B0 & E2 \\ 3D & B8 & 8E & 04 \\ BC & 4D & 1C & A7 \end{pmatrix} \qquad \begin{pmatrix} 01 & 3A & 8C & 21 \\ 3E & B0 & E2 & 33 \\ 8E & 04 & 3D & B8 \\ A7 & BC & 4D & 1C \end{pmatrix}$$

- after Roundkey (Attention: no Mix columns in last round):

$$\begin{pmatrix} 29 & 57 & 40 & 1A \\ C3 & 14 & 22 & 02 \\ 50 & 20 & 99 & D7 \\ 5F & F6 & B3 & 3A \end{pmatrix}$$

- ciphertext: 29 C3 50 5F 57 14 20 F6 40 22 99 B3 1A 02 D7 3A

Self Study (**Not Included in Syllabus****)

- Rationale behind the steps.
- Mathematics Behind Galois Fields.
- S-box and Inverse S-box table generation

AES

- Based on a design principle known as a ***substitution–permutation network*** and is efficient in both software and hardware.
- Takes a block of the plaintext and the key as inputs, and applies several alternating rounds or layers of substitution boxes (**S-boxes**) and permutation boxes (**P-boxes**) to produce the ciphertext block
- The key is introduced in each round, usually in the form of "round keys" derived from it.
- Decryption is done by simply reversing the process.

Substitution-Permutation Network (SP Net)

- An S-box substitutes a small block of bits (the input of the S-box) by another block of bits (the output of the S-box).
  - This substitution should be **one-to-one**, to ensure invertibility (hence decryption).
- A P-box is a permutation of all the bits
  - It takes the outputs of all the S-boxes of one round, permutes the bits, and feeds them into the S-boxes of the next round.

- A good P-box has the property that the output bits of any S-box are distributed to as many S-box inputs as possible.
- The round key (obtained from the key with some simple operations, for instance, using S-boxes and P-boxes) is combined using some group operation, typically XOR.

SP Net : Target

- A single typical S-box or a single P-box alone does not have much cryptographic strength
  - An S-box could be thought of as a substitution cipher, while a P-box could be thought of as a transposition cipher.

- A well-designed SP network with several alternating rounds of S-boxes and P-boxes already satisfies ***Shannon's confusion and diffusion properties***
  - These properties, when present, work to restrict the application of statistics and other methods of cryptanalysis.

Shannon's **Confusion** and **Diffusion** Properties

- ***Confusion*** means that each binary digit (bit) of the ciphertext should depend on several parts of the key, obscuring the connections between the two.

- ***Diffusion*** means that if we change a single bit of the plaintext, then *about half of the bits* in the ciphertext should change, and similarly, if we change one bit of the ciphertext, then about half of the plaintext bits should change.

AES (Advanced Encryption Standard)

- Based on the substitution–permutation network design principle
- Is efficient in both software and hardware
- Steps: (Encryption Side
  - Key expansion
  - Initial round key addition
  - 9/11/13 rounds of subBytes, shiftRow and mixColumn and addRoundKey
  - Final row of shiftRow, subBytes and addRoundKey
    - N.B: mixColumn sub-step is not present here!!!

AES (Advanced Encryption Standard)

- **subBytes Step:** *Uses* S-Box
  - Avoid attacks based on simple algebraic properties
  - The **Rijndael S-box** was specifically designed to be resistant to linear and differential cryptanalysis.
    - Minimizing the correlation between linear transformations of input/output bits.
    - Minimizing the difference propagation probability.
  - Obscures the relationship between the key and the ciphertext, thus ensuring Shannon's property of ***confusion***.

AES (Advanced Encryption Standard)

shiftRows and shiftColumns Step *Implements* P-Box combinedly.

- Shift Rows:
  - Avoids the columns being encrypted independently, otherwise AES would degenerate into four independent block ciphers
- Mix Columns:
  - Together with ShiftRows, MixColumns provides ***diffusion*** in the cipher.

Attacks on Block Ciphers (like DES,3DES,AES)

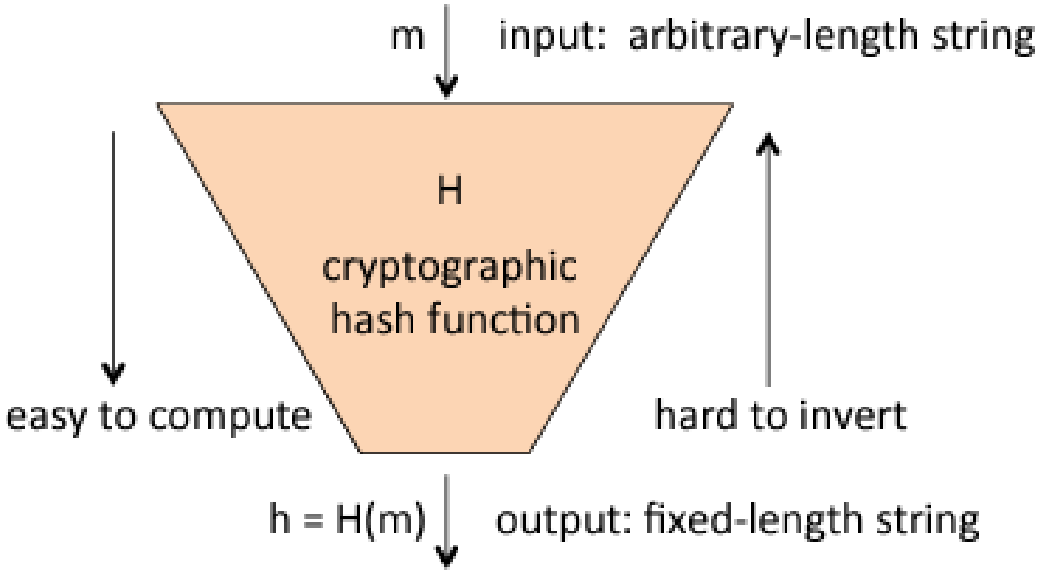
- **Linear Cryptanalysis:**
  - General form of cryptanalysis based on finding affine approximations to the action of a cipher
- Steps:
  - Construct linear equations relating plaintext, ciphertext and key bits that have a high bias; that is, whose probabilities of holding are as close as possible to 0 or 1
  - Use these linear equations in conjunction with ***known plaintext-ciphertext*** pairs to derive key bits.

Attacks on Block Ciphers

- **Differential Cryptanalysis:**
  - It is the study of how differences in information input can affect the resultant difference at the output
  - The attacker then computes the differences of two ciphertexts of two known plaintexts, hoping to detect statistical patterns in their distribution
- Statistical properties depend upon the ***nature of the S-boxes*** used for encryption

Cryptographic Hash Function

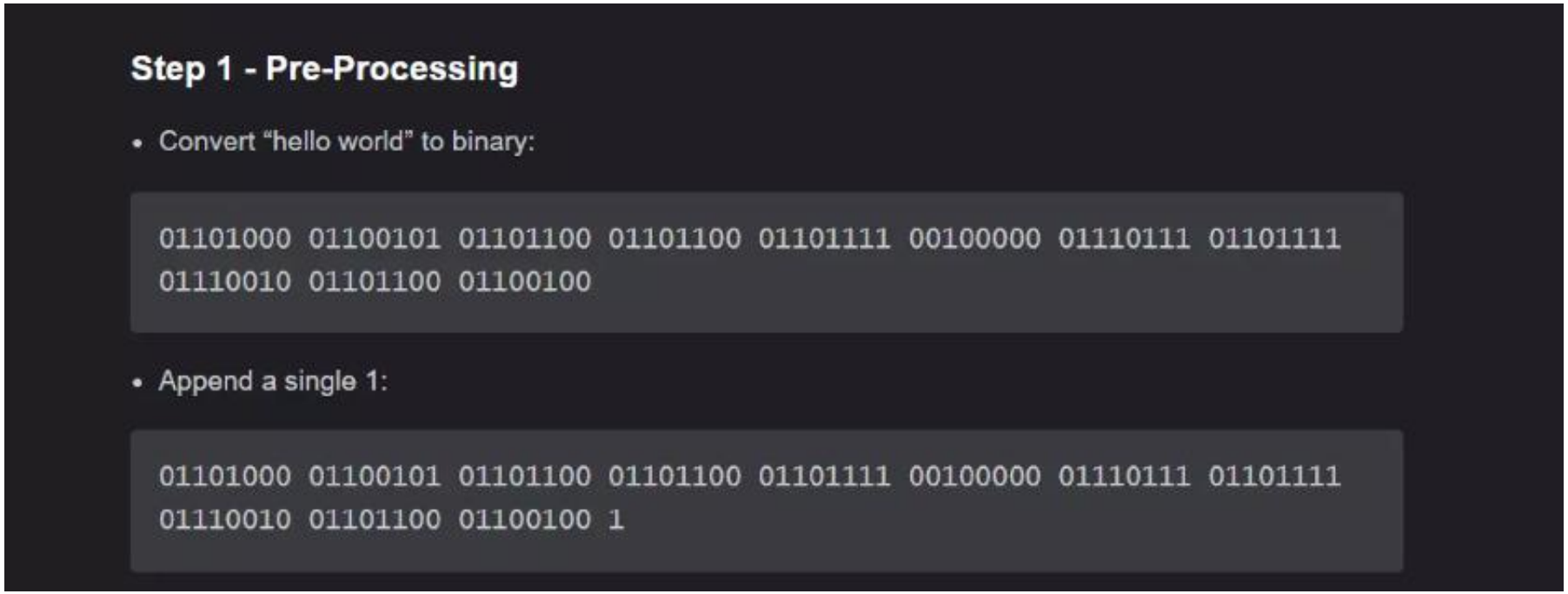
- A cryptographic hash function (CHF) is a mathematical algorithm that maps data of an arbitrary to a bit array of a fixed size



SHA-256

- SHA stands for **Secure Hash Algorithm** is one of a number of cryptographic hash functions.
- SHA is a family of cryptographic hash functions published by the National Institute of Standards and Technology (NIST).
- SHA-0 is the first version of SHA-Algorithm in 1993 and then SHA-1 in 2004.
- SHA-2 is a widely used in standard modern cryptographic applications.
- Bitcoin uses the SHA-256 variants as a hashing algorithm to solve Proof-of-Work(POW : the process of validating transactions on a blockchain to confirm transactions, close a block, and open a new one).

How does SHA-256 Algorithm Works?



- Pad with 0's until data is a multiple of 512, less 64 bits (448 bits in our case):

```
01101000 01100101 01101100 01101100 01101111 00100000 01110111 01101111
01110010 01101100 01100100 10000000 00000000 00000000 00000000 00000000
00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
```

- Append 64 bits to the end, where the 64 bits are a big-endian integer representing the length of the original input in binary. In our case, 88, or in binary, "1011000".

```
01101000 01100101 01101100 01101100 01101111 00100000 01110111 01101111
01110010 01101100 01100100 10000000 00000000 00000000 00000000 00000000
00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
00000000 00000000 00000000 00000000 00000000 00000000 00000000 01011000
```

Now we have our input, which will always be evenly divisible by 512.

Step 2 - Initialize Hash Values (h)

Now we create 8 hash values. These are hard-coded constants that represent the first 32 bits of the fractional parts of the square roots of the first 8 primes: 2, 3, 5, 7, 11, 13, 17, 19

```
h0 := 0x6a09e667
h1 := 0xbb67ae85
h2 := 0x3c6ef372
h3 := 0xa54ff53a
h4 := 0x510e527f
h5 := 0x9b05688c
h6 := 0x1f83d9ab
h7 := 0x5be0cd19
```

Step 3 - Initialize Round Constants (k) 🔗

Similar to step 2, we are creating some constants (*Learn more about constants and when to use them here*). This time, there are 64 of them. Each value (0-63) is the first 32 bits of the fractional parts of the cube roots of the first 64 primes (2 - 311).

Step 3 - Initialize Round Constants (k)

Similar to step 2, we are creating some constants (*Learn more about constants and when to use them here*). This time, there are 64 of them. Each value (0-63) is the first 32 bits of the fractional parts of the cube roots of the first 64 primes (2 - 311).

```
0x428a2f98 0x71374491 0xb5c0fbcf 0xe9b5dba5 0x3956c25b 0x59f111f1 0x923f82a
0xd807aa98 0x12835b01 0x243185be 0x550c7dc3 0x72be5d74 0x80deb1fe 0x9bdc06a
0xe49b69c1 0xefbe4786 0x0fc19dc6 0x240ca1cc 0x2de92c6f 0x4a7484aa 0x5cb0a9c
0x983e5152 0xa831c66d 0xb00327c8 0xbf597fc7 0xc6e00bf3 0xd5a79147 0x06ca635
0x27b70a85 0x2e1b2138 0x4d2c6dfc 0x53380d13 0x650a7354 0x766a0abb 0x81c2c92
0xa2bfe8a1 0xa81a664b 0xc24b8b70 0xc76c51a3 0xd192e819 0xd6990624 0xf40e358
0x19a4c116 0x1e376c08 0x2748774c 0x34b0bcb5 0x391c0cb3 0x4ed8aa4a 0x5b9cca4
0x748f82ee 0x78a5636f 0x84c87814 0x8cc70208 0x90befffa 0xa4506ceb 0xbef9a3f
```

Step 4 - Chunk Loop

The following steps will happen for each 512-bit “chunk” of data from our input. In our case, because “hello world” is so short, we only have one chunk. At each iteration of the loop, we will be mutating the hash values h0-h7, which will be the final output.

Step 5 - Create Message Schedule (w)

- Copy the input data from step 1 into a new array where each entry is a 32-bit word:

```
01101000011001010110110001101100 01101111001000000111011101101111
01110010011011000110010010000000 00000000000000000000000000000000
00000000000000000000000000000000 00000000000000000000000000000000
00000000000000000000000000000000 00000000000000000000000000000000
00000000000000000000000000000000 00000000000000000000000000000000
00000000000000000000000000000000 00000000000000000000000000000000
00000000000000000000000000000000 00000000000000000000000000000000
00000000000000000000000000000000 00000000000000000000000001011000
```

- Add 48 more words initialized to zero, such that we have an array **w[0…63]**

```
01101000011001010110110001101100 01101111001000000111011101101111
01110010011011000110010010000000 00000000000000000000000000000000
00000000000000000000000000000000 00000000000000000000000000000000
00000000000000000000000000000000 00000000000000000000000000000000
00000000000000000000000000000000 00000000000000000000000000000000
00000000000000000000000000000000 00000000000000000000000000000000
00000000000000000000000000000000 00000000000000000000000000000000
00000000000000000000000000000000 00000000000000000000000001011000
00000000000000000000000000000000 00000000000000000000000000000000
00000000000000000000000000000000 00000000000000000000000000000000
00000000000000000000000000000000 00000000000000000000000000000000
00000000000000000000000000000000 00000000000000000000000000000000
```

- Modify the zero-ed indexes at the end of the array using the following algorithm:
- For **i** from w[16…63]:
  - s0 = (w[i-15] rightrotate 7) xor (w[i-15] rightrotate 18) xor (w[i-15] rightshift 3)
  - s1 = (w[i- 2] rightrotate 17) xor (w[i- 2] rightrotate 19) xor (w[i- 2] rightshift 10)
  - w[i] = w[i-16] + s0 + w[i-7] + s1

Let's do w[16] so we can see how it works:

```
w[1] rightrotate 7:
  01101111001000000111011101101111 -> 11011110110111100100000011101110
w[1] rightrotate 18:
  01101111001000000111011101101111 -> 00011101110110111101101111001000
w[1] rightshift 3:
  01101111001000000111011101101111 -> 00001101111001000000111011101101

s0 = 11011110110111100100000011101110 XOR 00011101110110111101101111001000

s0 = 11001110111000011001010111001011

w[14] rightrotate 17:
  00000000000000000000000000000000 -> 00000000000000000000000000000000
w[14] rightrotate19:
  00000000000000000000000000000000 -> 00000000000000000000000000000000
w[14] rightshift 10:
  00000000000000000000000000000000 -> 00000000000000000000000000000000

s1 = 00000000000000000000000000000000 XOR 00000000000000000000000000000000
```

```
s1 = 00000000000000000000000000000000

w[16] = w[0] + s0 + w[9] + s1

w[16] = 01101000011001010110110001101100 + 11001110111000011001010111001011

// addition is calculated modulo 2^32

w[16] = 00110111010001110000001000110111
```

This leaves us with 64 words in our message schedule (w):


01101000011001010110110001101100 01101111001000001110111011011111 01110010011011000110010010000000
00000000000000000000000000000000 00000000000000000000000000000000 00000000000000000000000000000000
00000000000000000000000000000000 00000000000000000000000000000000 00000000000000000000000000000000
00000000000000000000000000000000 00000000000000000000000000000000 00000000000000000000000000000000
00000000000000000000000000000000 00000000000000000000000000000000 00000000000000000000000000000000
00000000000000000000000001011000 00110111010001110000010001101111 10000110110100001100000000110001
11010011101111010001000100001011 01111000001111110100011110000010 00101010100100000111110011101101
01001011001011110111111001100100 10001001001101100100100101100100
01111111011110100000011011011010 11000001011110011010100100111010 10111011111010001111011001010101
00001100000110101110001111100110 10110000111111100000110101111101 01011111011011100101010110010011
00000000100010011001101101010010 00000111111100011100101010010100 00111011010111111110010111010110
01101000011001010110001011100110 11001000010011100000101010011110 00000110101011111001101100100101
10010010111011110110010011010111 01100011111110010101111001011010 11100011000101100110011111010111
10000100001110111101111000010110 11101110111011001010100001011011 10100000010011111111001000100001
11111001000110001010110110111000 00010100101010001001001000011001 00010000100001000101001100011101
01100000100100111110000011001101 10000011000000110101111111101001 11010101101011100111100100111000
00111001001111110000010110101101 11111011010010110001101111101111 11101011011101011111111100101001
01101010001101101001010100110100 00100010111111001001110011011000 10101001011101000000110100101011
01100000110011110011100010000101 11000100101011001001100000111010 00010001010000101111110110101101
10110000101100000001110111011001 10011000111100001100001101101111 01110010000101111011100000011110
10100010110101000110011110011010 00000001000011111001100101111011 11111100000101110100111100001010
11000010110000101110101100010110

Step 6 - Compression

- Initialize variables **a, b, c, d, e, f, g, h** and set them equal to the current hash values respectively. **h0, h1, h2, h3, h4, h5, h6, h7**
- Run the compression loop. The compression loop will mutate the values of **a...h**. The compression loop is as follows:
- for i from 0 to 63
  - S1 = (e rightrotate 6) xor (e rightrotate 11) xor (e rightrotate 25)
  - ch = (e and f) xor ((not e) and g)
  - temp1 = h + S1 + ch + k[i] + w[i]
  - S0 = (a rightrotate 2) xor (a rightrotate 13) xor (a rightrotate 22)
  - maj = (a and b) xor (a and c) xor (b and c)
  - temp2 := S0 + maj
  - h = g
  - g = f
  - f = e
  - e = d + temp1
  - d = c
  - c = b
  - b = a
  - a = temp1 + temp2

Let's go through the first iteration, all addition is calculated modulo 2^32:

```
a = 0x6a09e667 = 01101010000010011110011001100111
b = 0xbb67ae85 = 10111011011001111010111010000101
c = 0x3c6ef372 = 00111100011011011111001101110010
d = 0xa54ff53a = 10100101010011111111010100111010
e = 0x510e527f = 01010001000011100101001001111111
f = 0x9b05688c = 10011011000001010110100010001100
g = 0x1f83d9ab = 00011111100000111101100110101011
h = 0x5be0cd19 = 01011011111000001100110100011001

e rightrotate 6:
  01010001000011100101001001111111 -> 11111101010001000011100101001001
e rightrotate 11:
  01010001000011100101001001111111 -> 01001111111010100010000111001010
e rightrotate 25:
  01010001000011100101001001111111 -> 10000111001010010011111110101000
S1 = 11111101010001000011100101001001 XOR 01001111111010100010000111001010
S1 = 00110101100001110010011100101011

e and f:
    01010001000011100101001001111111
  & 10011011000001010110100010001100 =
    00010001000001000100000000001100
not e:
  01010001000011100101001001111111 -> 10101110111100011010110110000000
(not e) and g:
    10101110111100011010110110000000
```

```
(not e) and g:
    10101110111100011010110110000000
  & 00011111100000111101100110101011 =
    00001110100000011000100110000000
ch = (e and f) xor ((not e) and g)
   = 00010001000001000100000000001100 xor 00001110100000011000100110000000
   = 00011111100001011100100110001100

// k[i] is the round constant
// w[i] is the batch
temp1 = h + S1 + ch + k[i] + w[i]
temp1 = 01011011111000001100110100011001 + 00110101100001110010011100101011
temp1 = 01011011110111010101100111010100

a rightrotate 2:
  01101010000010011110011001100111 -> 11011010100000100111100110011001
a rightrotate 13:
  01101010000010011110011001100111 -> 00110011001110110101000001001111
a rightrotate 22:
  01101010000010011110011001100111 -> 00100111100110011001110110101000
S0 = 11011010100000100111100110011001 XOR 00110011001110110101000001001111
S0 = 11001110001000001011010001111110

a and b:
    01101010000010011110011001100111
  & 10111011011001111010111010000101 =
    00101010000000011010011000000101
```

That entire calculation is done 63 more times, modifying the variables a-h throughout. We won't do it by hand but we would have ender with:

```
h0 = 6A09E667 = 01101010000010011110011001100111
h1 = BB67AE85 = 10111011011001111010111010000101
h2 = 3C6EF372 = 00111100011011101111001101110010
h3 = A54FF53A = 10100101010011111111010100111010
h4 = 510E527F = 01010001000011100101001001111111
h5 = 9B05688C = 10011011000001010110100010001100
h6 = 1F83D9AB = 00011111100000111101100110101011
h7 = 5BE0CD19 = 01011011111000001100110100011001

a = 4F434152 = 01001111010000110100000101010010
b = D7E58F83 = 11010111111001011000111110000011
c = 68BF5F65 = 01101000101111110101111101100101
d = 352DB6C0 = 00110101001011011011011011000000
e = 73769D64 = 01110011011101101001110101100100
f = DF4E1862 = 11011111010011100001100001100010
g = 71051E01 = 01110001000001010001111000000001
h = 870F00D0 = 10000111000011110000000011010000
```

Step 7 - Modify Final Values

After the compression loop, but still, within the *chunk* loop, we modify the hash values by adding their respective variables to them, a-h. As usual, all addition is modulo 2^32.

```
h0 = h0 + a = 10111001010011010010011110111001
h1 = h1 + b = 10010011010011010011111000001000
h2 = h2 + c = 10100101001011100101001011010111
h3 = h3 + d = 11011010011111011010101111111010
h4 = h4 + e = 11000100100001001110111111100011
h5 = h5 + f = 01111010010100111000000011101110
h6 = h6 + g = 10010000100010001111011110101100
h7 = h7 + h = 11100010111011111100110111101001
```

Cryptographic Hash Function : **Properties**

Pre-image resistance

- for essentially all possible hash values *h*, given *h* it should be infeasible to find any *m* such that *H(m) = h*.
- Also known as **One-way** property

Second pre-image resistance

- Given any first input m1, it should be infeasible to find any distinct second input m2 such that H(m1) = H(m2)
- Also known as ***weak*** *collision resistance*

Collision resistance

- It should be infeasible to find any pair of distinct inputs m1, m2 such that H(m1) = H(m2)
- Also known as ***strong*** *collision resistance*

Collision Resistant Hash Functions

- Second-preimage resistance **fails** to guarantee collision resistance
  - Fixing one string (say m1 in H) makes collision-finding significantly more costly
  - Reason : **Birthday Paradox !!!**

Let's assume, there are "m" number of days in a year and we have k persons for whom we want to check whether any two of them has the same day as birthdate.

Let's use the complement of Probability technique.

For 1st person,
probability that he has a unique date as his birthday.
is $\frac{m}{m} = 1$

For 2nd person,
the probability $= \frac{m-1}{m} = \left(1 - \frac{1}{m}\right)$

For 3rd person,
the probability $= \frac{m-2}{m} = \left(1 - \frac{2}{m}\right)$

⋮ ⋮

For kth person,
the probability, $= \frac{m-(k-1)}{m} = 1 - \frac{k-1}{m}$

∴ Probability that each of them has a unique birthdate is

$$= 1 \times \left(1 - \frac{1}{m}\right) \times \left(1 - \frac{2}{m}\right) \times \left(1 - \frac{3}{m}\right) \cdots \times \left(1 - \frac{k-1}{m}\right)$$

$$= p(k, m) \quad \text{[Let's assume]}$$

Let's take logarithm both Side, we get,

$$\ln p = \ln\left(1 - \frac{1}{m}\right) + \ln\left(1 - \frac{2}{m}\right) + \cdots + \ln\left(1 - \frac{k-1}{m}\right)$$

$$= \left(-\frac{1}{m} - \frac{1}{m^2} - \frac{1}{m^3} - \cdots\right) +$$

$$\left(-\frac{2}{m} - \frac{2^2}{m^2} - \frac{2^3}{m^3} - \cdots\right) +$$

$$\cdots\cdots\cdots\cdots + \left\{-\frac{k-1}{m} - \frac{(k-1)^2}{m^2} - \cdots\right\}$$

Only considering simple terms (neglecting higher order terms) we get an approx. value for the probability is

$$\ln p \cong \left(-\frac{1}{m} - \frac{2}{m} - \frac{3}{m} - \cdots - \frac{k-1}{m}\right)$$

$$\Rightarrow p \cong e^{-\left(\frac{1}{m} + \frac{2}{m} + \cdots + \frac{k-1}{m}\right)}$$

$$\cong e^{-\frac{1}{m} \times \frac{(k-1)\cdot k}{2}} \quad \left[\because (1+2+3+\cdots n) = \frac{n(n+1)}{2}\right]$$

$$\cong e^{-\frac{k^2}{2m}} \quad \text{[applying more of approximations]}$$

So, probability that each of them (~~k~~ people) has a unique birthday is $e^{-\frac{k^2}{2m}}$ (approx).

∴ Probability that **at least** two of them has same birthdate is $1 - P(k,m)$

$$\cong 1 - e^{-\frac{k^2}{2m}} = P'(k,m) \quad \text{[Let]}$$

if

∴ So, probability that at least two of them has same birthdate is 50%, then we get

$$p'(k,m) \cong 1 - e^{-\frac{k^2}{2m}}$$

$$\Rightarrow 50\% \cong 1 - e^{-\frac{k^2}{2\times 365}} \quad [m = 365, \text{fun \#days in a year}]$$

$$\Rightarrow e^{-\frac{k^2}{730}} = 1/2$$

$$\Rightarrow k^2 \cong -730 \ln(0.5)$$

$$\therefore k \cong 22.49 \cong 23 \text{ (approx)}$$

So, Only 23 people are present in a room, there is 50% chance that at least two of them has same birthdate.

The Birthday Paradox

- ***What number n of people are needed in a room before a shared birthday is expected among them (i.e., with probability p = 0.5)?***
  - Ans. Only 23.
- Related Question :

  Given *n* people in a room, what is the probability that two of them have the same birthday?
  - P = 0.71 for n = 30
  - P = 0.97 for n = 50
- In security, attackers can often solve problems more efficiently than expected !
- *The "collision" here is not for one pre-specified day (e.g., your birthday); any matching pair will do*

Probability that each of them has a unique birth date is $P(k,m)$

$$= 1 \times \left(1-\frac{1}{m}\right) \times \left(1-\frac{2}{m}\right) \times \cdots \times \left(1-\frac{k-1}{m}\right)$$

$$= \frac{m-1}{m} \times \frac{m-2}{m} \times \frac{m-3}{m} \times \cdots \times \frac{m-(k-1)}{m}$$

$$= \frac{(m-1)\cdot(m-2)\cdot(m-3)\cdots\{m-(k-1)\}}{m^{(k-1)}}$$

$$= \frac{m\cdot(m-1)\cdot(m-2)\cdots\{m-(k-1)\}}{m^{k}}$$

$$= \frac{k! \times \frac{m(m-1)(m-2)\cdots\{m-(k-1)\}}{k!}}{m^{k}}$$

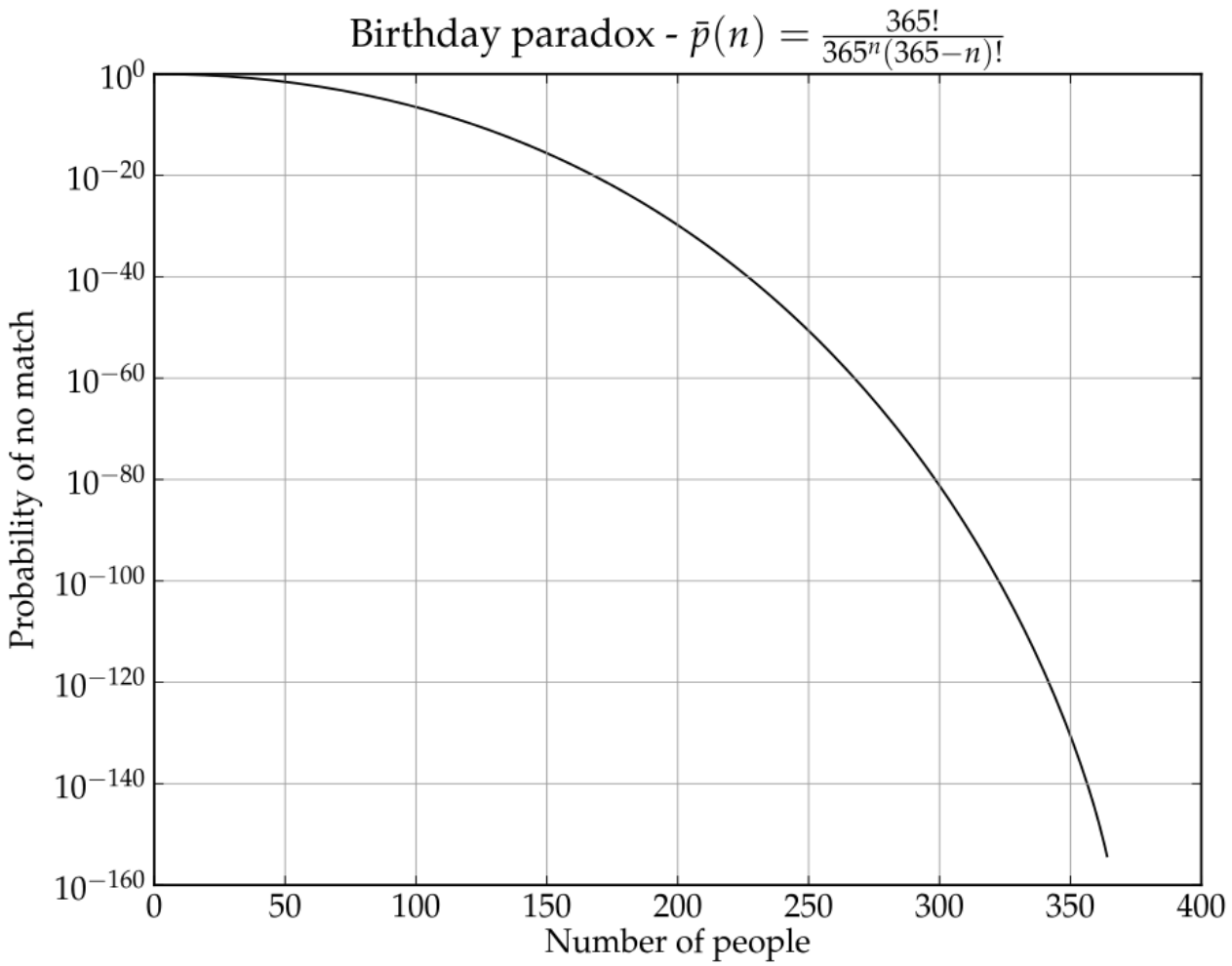
$$= \frac{k!\cdot\left({}^{m}C_{k}\right)}{m^{k}}$$

$$= \frac{k! \cdot m!}{k!(m-k)! \times m^k}$$

$$= \frac{m!}{(m-k)!} \times \frac{1}{m^k}$$

So, if $m = 365$, then

$$= \frac{365!}{(365-k)! \times 365^k}$$

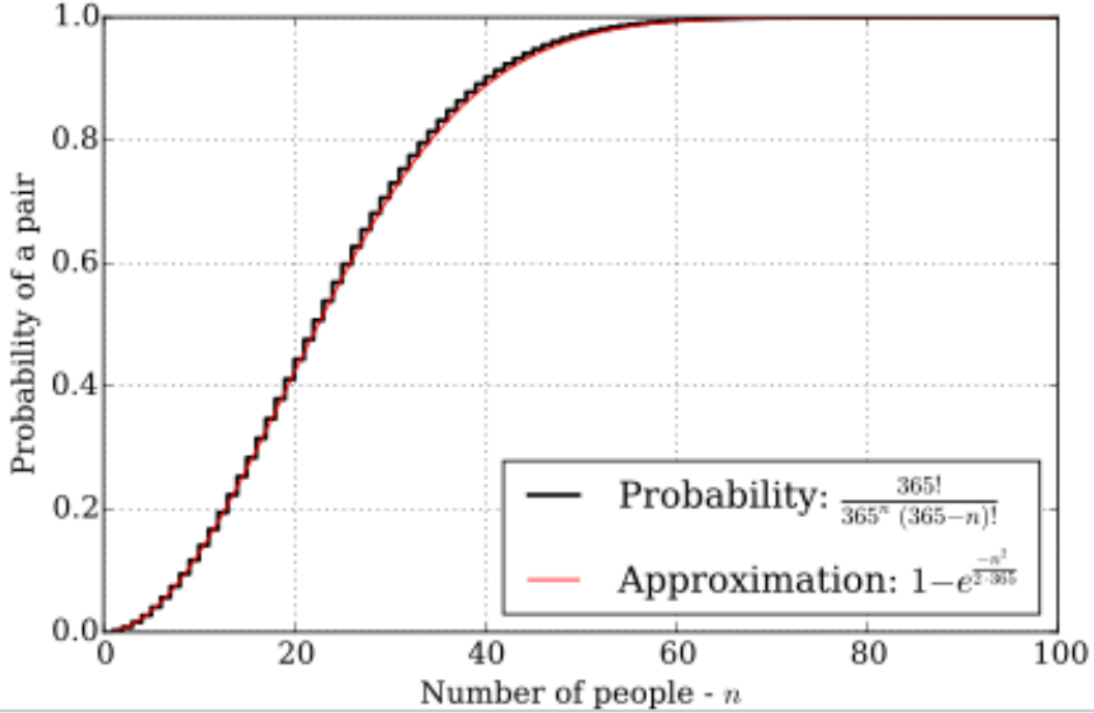


Probability that no two people share a birthday in a group of *n* people

Note that the vertical scale is logarithmic (each step down is 10^{20} times less likely)

The Birthday Paradox

n	$p(n)$
1	0.0%
5	2.7%
10	11.7%
20	41.1%
23	50.7%
30	70.6%
40	89.1%
50	97.0%
60	99.4%
70	99.9%
75	99.97%
100	99.999 97%
200	99.999 999 999 999 999 999 999 999 999 9998%
300	$(100 - 6 \times 10^{-80})\%$
350	$(100 - 3 \times 10^{-129})\%$
365	$(100 - 1.45 \times 10^{-155})\%$
≥ 366	100%



A graph showing the accuracy of the approximation $1 - e^{-n^2/730}$ (red)

The Birthday Paradox Vs Hash Value Collisions

Birthday Collisions

m = 365 (1 year = 365 days)

k = # of persons considered

Hash Collisions

m = # of hash values available for messages

k = # of messages considered for mapping into those available hash values.

Hash Function Design : The Merkle-Damgard Construction

- The Merkle Damgard construction is a process of making a cryptographic hash function using a **one-way compression function**.
- This construction is based on the rule that if the compression function is collision resistance, the hash function will also be collision resistance.
- Many popular hash functions like **MD5**, **SHA-1**, and **SHA-2** have been designed using Merkle Damgard construction.

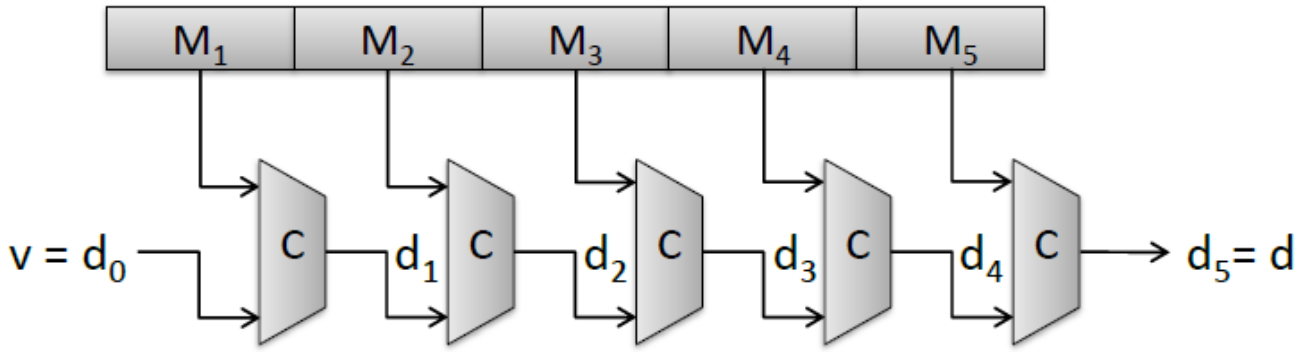


Figure 12: The Merkle-Damgård construction.

The Merkle-Damgard Construction : **How it works?**

- The process starts with expanding the input message to a length that is multiple of some fixed number of bits.
- It is necessary because compression function only works on the fixed-length inputs.

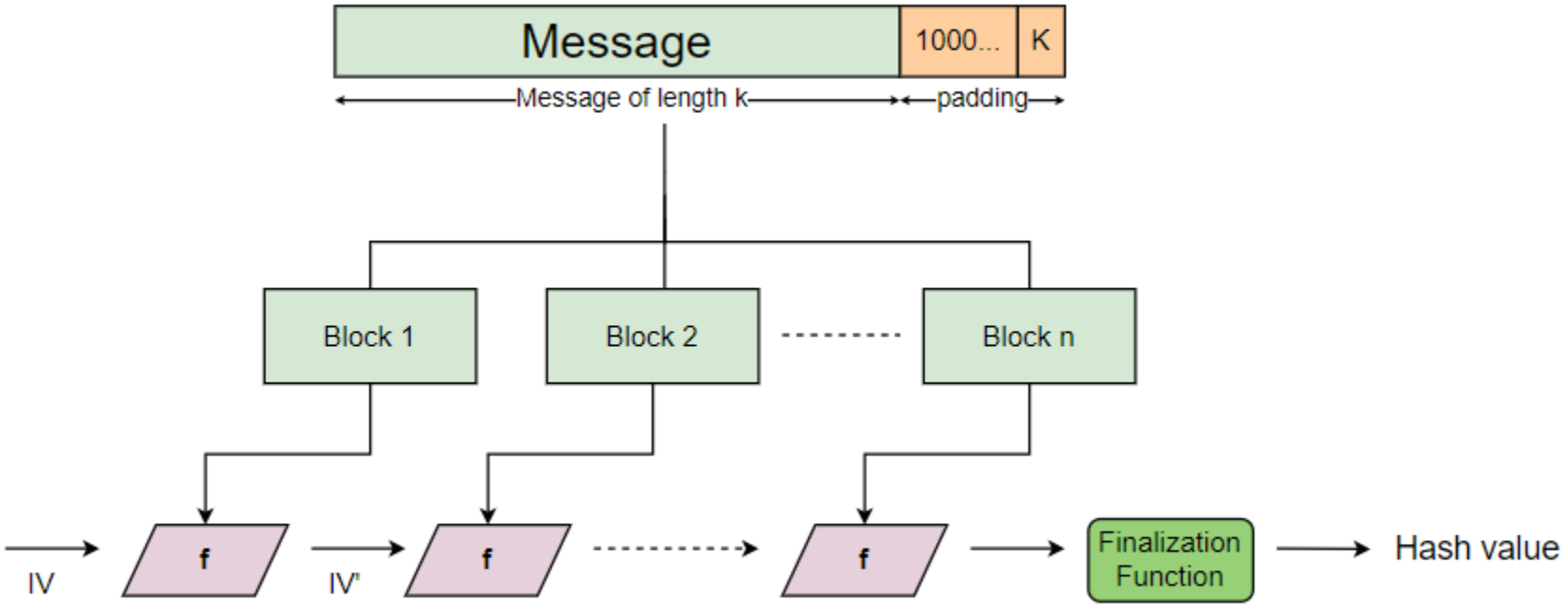
Input message with padding

MD-compliant padding scheme

The Merkle-Damgard Construction : **How it works?**

- Why padding is important?
  - It is important to carefully select the padding scheme for message length expansion because weak padding can introduce security vulnerability to the function.
  - Padding should be **MD-compliant** which means it should satisfy following conditions:
    - M is a prefix of $Pad(M)$
    - If $|M_1| = |M_2|$ then $|Pad(M_1)| = |Pad(M_2)|$
    - If $|M_1| \neq |M_2|$, then the last block of $|Pad(M_1)|$ is different from the last block of $|Pad(M_2)|$

The Merkle-Damgard Construction : **How it works?**



Merkle-Damgard construction

RSA

RSA is one of the oldest **asymmetric** encryption algorithm

The acronym "RSA" comes from the surnames of **R**on Rivest, Adi **S**hamir and Leonard **A**dleman

The security of RSA relies on the practical difficulty of factoring the product of two large prime numbers, the "factoring problem".

RSA algorithm steps

1. Key-Generation

Step-1: Select two large prime numbers p and q where $p \neq q$.

Step-2: Calculate $n = p * q$.

Step-3: Calculate $\Phi(n) = (p-1) * (q-1)$.

Step-4: Select e such that, e is relatively prime to $\Phi(n)$, i.e. $\gcd(e, \Phi(n)) = 1$ and $1 < e < \Phi(n)$

Step-5: Calculate $d = e^{-1} \bmod \Phi(n)$ or $ed = 1 \bmod \Phi(n)$.

Step-6: Public key = {e, n}, Private key = {d, n}.

RSA algorithm : Example

Step – 1: Select two prime numbers p and q where $p \neq q$.
Example, Two prime numbers p = 13, q = 11.

Step – 2: Calculate n = p * q.
Example, n = p * q = 13 * 11 = 143.

Step – 3: Calculate $\Phi(n) = (p-1) * (q-1)$.
Example, $\Phi(n) = (13 - 1) * (11 - 1) = 12 * 10 = 120$.

Step – 4: Select e such that, e is relatively prime to $\Phi(n)$,
i.e. $\gcd(e, \Phi(n)) = 1$ and $1 < e < \Phi(n)$.
Example, Select e = 13, gcd (13, 120) = 1.
See Euler's Totient

RSA algorithm : Example

- **Step – 5:** Calculate $d = e^{-1} \bmod \Phi(n)$ or $e * d = 1 \bmod \Phi(n)$
- **Example,** Finding d: $e * d \bmod \Phi(n) = 1$
- $13 * d \bmod 120 = 1$
- (How to find: $d * e = 1 \bmod \Phi(n)$
- $d = ((\Phi(n) * i) + 1) / e$
- $d = (120 + 1) / 13 = 9.30$ ($\because i = 1$)
- $d = (240 + 1) / 13 = 18.53$ ($\because i = 2$)
- $d = (360 + 1) / 13 = 27.76$ ($\because i = 3$)
- $d = (480 + 1) / 13 = 37$ ($\because i = 4$))

Step – 6: Public key = {e, n}, private key = {d, n}.
Example,
Public key = {13, 143} and
Private key = {37, 143}.

2. Encryption

- Find out *cipher text* using the formula,
  $C = P^e \bmod n$ where, $P < n$.

- **Example,** Plain text P = 13. (Where, $P < n$)
  $C = P^e \bmod n = 13^{13} \bmod 143 = 52$.

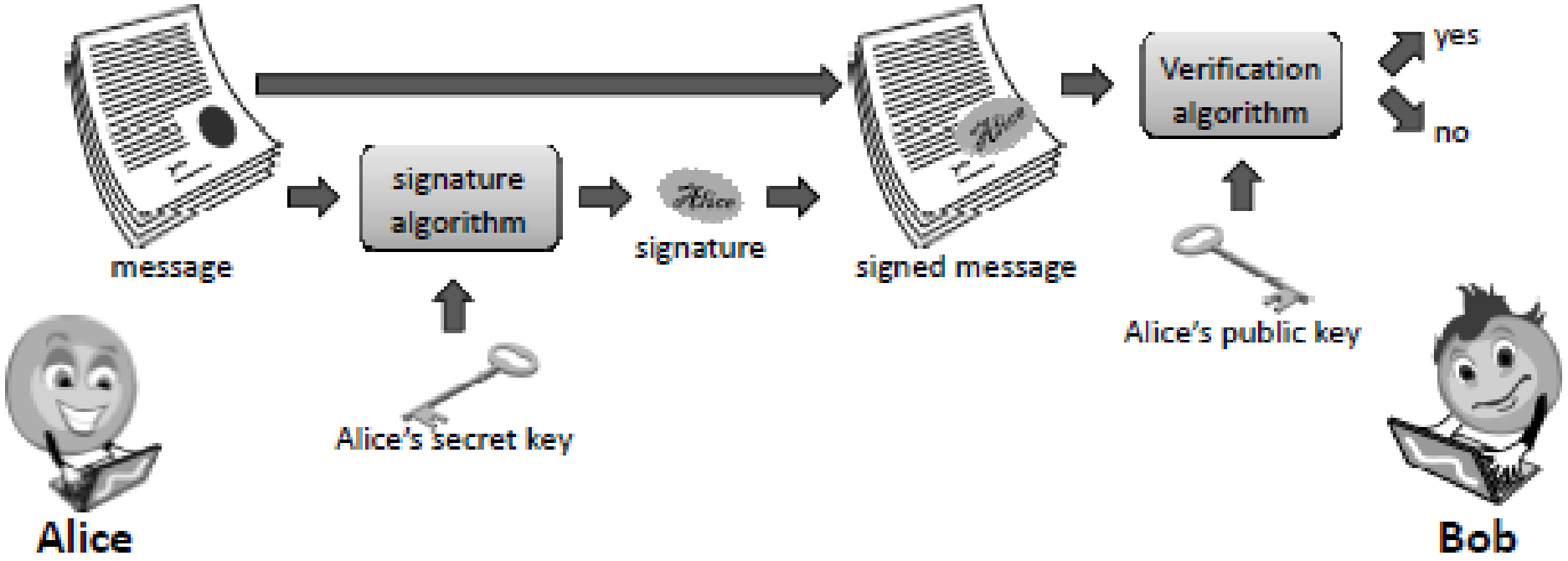
3. Decryption

- $P = C^d \bmod n$. Plain text P can be obtained using the given formula.

- **Example,** Cipher text C = 52

  $P = C^d \bmod n = 52^{37} \bmod 143 = 13.$

Digital Signature



RSA Signature Scheme

- Almost Similar (actually reverse) to the RSA Algorithm

$$S = M^d \bmod n.$$

$$\text{Is it true that } M = S^e \bmod n?$$

Given That, $: de \bmod \phi(n) = 1.$

$$S^e \bmod n = M^{de} \bmod n = M^{de \bmod \phi(n)} \bmod n = M^1 \bmod n = M.$$

RSA Digital Signature Scheme : Proof

Alice (Sender)

- Creating Message, M
- Generating Signature, S
  - $S = M^d \ mod \ n$
- Sending (M, S) pair through Communication Channel)

Bob (Receiver)

- Receiving the pair (M, S)
- Extracting M' from Signature
- $M' = S^e mod \ n$

$$= M^{de} \bmod n$$

$$= M^{de \bmod \phi(n)} \bmod n$$

$$= M^1 \bmod n = M.$$

Digital Signature

- Digital signature provides 3 important properties
  - ***Data origin authentication***: assurance of who originated (signed) a message or file
  - ***Data integrity:*** assurance that received content is the same as that originally signed.
  - ***Non-repudiation:*** strong evidence of unique origination, making it hard for a party to digitally sign data and later successfully deny having done so.

RSA Signature Scheme : **Existential Forgery**

Alice: (Sender)

- Creating Messages M_1, M_2 [In a gap of few seconds.]
- Generating Signatures
  $S_1 = M_1^d \bmod n$
  $S_2 = M_2^d \bmod n$
- Sending to Bob these (M_1, S_1), (M_2, S_2) pairs. through communication channel.

Eve: (Eavesdropper)

- Eavesdrop these pairs (M_1, S_1) & (M_2, S_2).
- Creating new message,
  $M = M_1 * M_2$
- Generating new signature
  $S = S_1 * S_2$
- Sending the Pair (M, S) to Bob.

RSA Signature Scheme : **Existential Forgery**

Bob: (Receiver)

- Receiving the Pair (M, S)
- Applying the Public Key $\{e, n\}$ On Signature
- $M' = S^e \bmod n$

$= (S_1 \cdot S_2)^e \bmod n$

$= \{(M_1^d \bmod n)^e \cdot (M_2^d \bmod n)^e\} \bmod n$

$= \{(M_1 de^{\bmod \phi(n)} \bmod n) \cdot (M_2 de^{\bmod \phi(n)} \bmod n)\} \bmod n$

$= \{(M_1^1 \bmod n) \cdot (M_2^1 \bmod n)\} \bmod n$

$= (M_1^1 * M_2^1) \bmod n$

$= M^1 \bmod n$

$= M \bmod n$

So, $M' = M$

∵ No Hints of violation of Data Integrity, Authentication, Non-Repudiation!!!

RSA Signature Scheme : Problem

- **Solution** :
  - Get cryptographic hash h of the message m
  - Sign on the *hash* of the message
  - Send the message other side
  - Receiver gets the hash h’ for that message
  - Extract the signature from message.
  - So.H(M) . H(N) very unlikely to match H(M.N)

RSA Signature Scheme : Existential Forgery Solution

Alice: (Sender)

- Creating Messages, M_1, M_2
- Generating Hashes, $H_1 = H(M_1)$, $H_2 = H(M_2)$
- Generating Signatures $S_1 = H_1^d \bmod n$, $S_2 = H_2^d \bmod n$
- Sending to Bob. - There (M_1, S_1) & (M_2, S_2) Pairs. through channel

Eve: (Eavesdropper)

- Eavesdrop there pairs (M_1, S_1) & (M_2, S_2).
- Creating new message, $M = M_1 * M_2$
- Generating new signature $S = S_1 * S_2$
- Sending the Pair (M, S) to Bob.

RSA Signature Scheme : Existential Forgery Solution

Bob: (Receiver)

- Receiving the pairs (M, S).
- Applying Hash on M.
  $H' = H(M)$
  $= H(M_1 \cdot M_2)$
- Applying Public key $\{e, n\}$ on Signature
- $H'' = S^e \bmod n$
  $= \{(H_1^d \bmod n)^e . (H_2^d \bmod n)^e\} \% n$
  $= \{(H_1^{de \bmod \phi(n)} \bmod n) . (H_2^{de \bmod \phi(n)} \bmod n)\} \% n$
  $= \{(H_1 \bmod n) . (H_2 \bmod n)\} \% n$
  $= (H_1 \cdot H_2) \bmod n$
  $= H_1 \cdot H_2$
  $= H(M_1) \cdot H(M_2)$

So, $H'' == H'$???

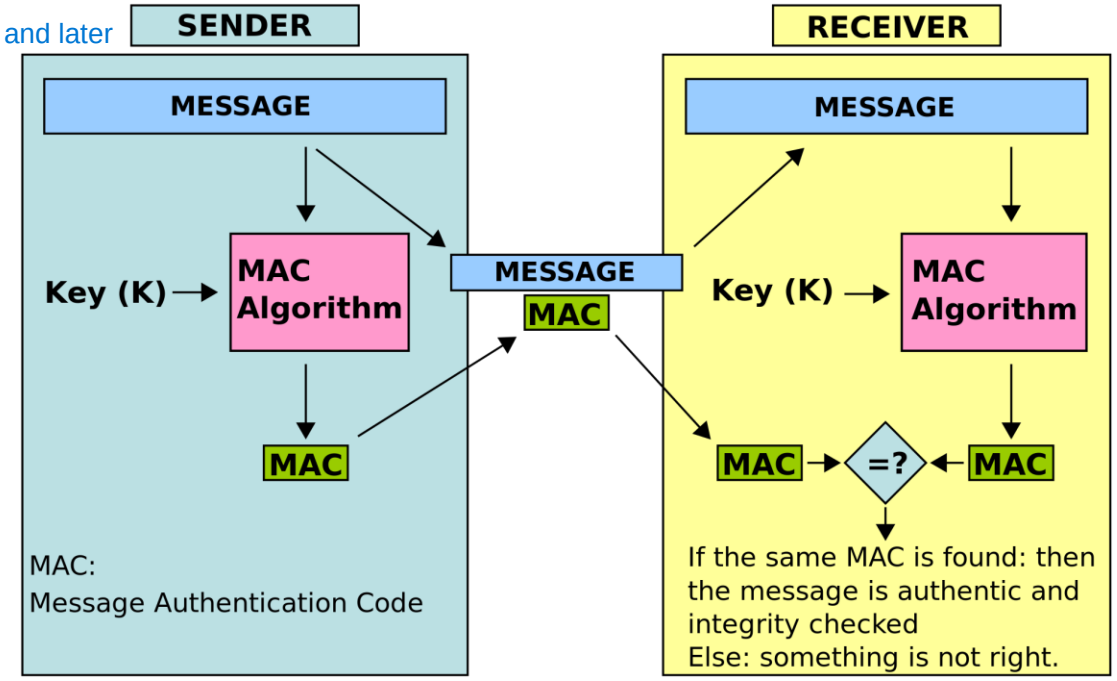
$H(M_1) \cdot H(M_2) == H(M_1 \cdot M_2)$.

So, Forgery Detected!!!

Message Authentication Code (MAC)

- Provides authentication and integrity
- But not Non-repudiation…
  - Because, even if a third party cannot compute the MAC, the sender could **dispute** sending the message and assert that the **recipient forged it** because it is impossible to tell which of the two parties computed the MAC.

making it hard for a party to digitally sign data and later successfully deny having done so



The Elgamal Cryptosystem

- The Elgamal cryptosystem, named after its inventor, Taher Elgamal, is a **public-key cryptosystem** that uses randomization, so that
- independent encryptions of the *same plaintext* are likely to produce *different ciphertexts.*

- It is based on viewing input blocks as numbers and applying arithmetic operations on these numbers to perform encryption and decryption.

The Elgamal Cryptosystem

- In the number system Z_p, all arithmetic is done modulo a prime number p
- A number, g in Z_p, is said to be a ***generator*** or primitive root modulo p if for each positive integer i in Z_p there is an integer k such that $i = g^k \ mod \ p$.
- There are $\phi(p-1)$ generators in Z_p

$Z_n^* = \{x \in Z_n \text{ such that } \text{GCD}(x,n) = 1\}.$

Let $\phi(n)$ be the number of elements of Z_n^*, that is,

$$\phi(n) = |Z_n^*|.$$

The Elgamal Cryptosystem

- For a prime number $\phi(p) = p - 1$
- Test different numbers until we find a generator
- Test for g : $g^{(p-1)/p_i} \bmod p \neq 1,$
  - For each prime factor p_i of $p - 1$
  - If any value is equal to 1, g is not a generator
  - Difficult to find all prime factors of p – 1 : choose p such that p is large and the factors of p-1 is known

The Elgamal Cryptosystem

- Once we have a generator g, we can efficiently compute $x = g^k \, mod \, p$ for any value k between 1 to $p - 2$

- Conversely, Conversely, given x, g, and p, the problem of determining k such that $x = g^k \, mod \, p$ is known as the ***discrete logarithm problem***.
  - Widely considered computationally difficult (similar to factoring problem)

The Elgamal Cryptosystem : **Method**

- As a part of the setup, Bob chooses a random large prime number, p, and finds a generator, g, for Z_p
- picks a random number, x, between 1 and $p - 2$, and computes

$$y = g^x \, mod \, p$$

  - x is the secret key for Bob
  - (p, g, y) triplet is the public key

The Elgamal Cryptosystem

- Alice wants to send message M to Bob
- First, she collects Bob's public key i.e. (p, g, y)
- Generates a random number k between 1 to $p - 2$
- she then uses modular ***multiplication and exponentiation*** to compute two numbers:

$$a = g^k \bmod p$$
$$b = My^k \bmod p.$$

- So, the ciphertext pair (a, b) is sent to Bob.
- Bob **decrypts** it by computing :

$$M = b(a^x)^{-1} \bmod p.$$

The Elgamal Cryptosystem

- Given that,

$$y = g^x \bmod p$$

$$\begin{aligned} a &= g^k \bmod p \\ b &= My^k \bmod p. \end{aligned}$$

Is it Valid ?

$$M = b(a^x)^{-1} \bmod p.$$

***Proof*:**

$$\begin{aligned} M' = b(a^x)^{-1} \bmod p &= My^k(g^{kx})^{-1} \bmod p \\ &= M(g^x)^k g^{-kx} \bmod p \\ &= Mg^{xk}g^{-kx} \bmod p \\ &= Mg^{kx}g^{-kx} \bmod p \\ &= M \bmod p \\ &= M. \end{aligned}$$

The Elgamal Cryptosystem : **Secure ???**

- The security of this scheme is based on the fact that, without knowing x, it would be very difficult for an eavesdropper to decrypt the ciphertext, (a, b).
- Since everyone knows $y = g^x \, mod \, p$, from Bob's public key, the security of this scheme is therefore related to the difficulty of solving the discrete logarithm problem.
  - Which is computationally difficult.
- Thus, the security of the Elgamal cryptosystem is based on a difficult problem from number theory.

The Elgamal Signature Scheme

- In the Elgamal signature scheme, document signatures are done through randomization, as in Elgamal encryption,
  - but the details for Elgamal signatures are quite different from Elgamal encryption

- Alice chooses a large random number, p, finds a generator for Z_P, picks a (secret) random number, x, computes $y = g^x \, mod \, p$, and publishes the pair (y, p, g) as her public key.
- To sign a message, M, Alice generates a fresh one-time-use random number, k, and computes the following two numbers:

$$a = g^k \bmod p$$
$$b = k^{-1}(M - xa) \bmod (p - 1).$$

The Elgamal Signature Scheme

$$\begin{aligned} a &= g^k \bmod p \\ b &= k^{-1}(M - xa) \bmod (p-1). \end{aligned}$$

The pair, (a, b), is Alice's signature on the message, M.

To verify the signature, (a, b), on M, Bob performs the following test:

$$\text{Is it true that } y^a a^b \bmod p \;=\; g^M \bmod p?$$

This is true because of the following:

$$\begin{aligned} y^a a^b \bmod p &= (g^x \bmod p)((g^k \bmod p)^{k^{-1}(M-xa) \bmod (p-1)} \bmod p) \\ &= g^{xa} g^{kk^{-1}(M-xa) \bmod (p-1)} \bmod p \\ &= g^{xa+M-xa} \bmod p \\ &= g^M \bmod p. \end{aligned}$$

The Elgamal Signature Scheme : Secure ???

- The security of this scheme is based on the fact that the computation of b depends on both the random number, k, and Alice's secret key, x.
- Also, because k is random, its inverse is also random; hence, it is impossible for an adversary to distinguish b from a random number, unless she can solve the discrete logarithm problem to determine the number k from a (which equals g k mod p).
- Thus, like Elgamal encryption, the security of the Elgamal signature scheme is based on the difficulty of computing discrete logarithms.

The Elgamal Signature Scheme : **Caution**

- Alice should ***never reuse*** the same k twice

$$b_1 = k^{-1}(M_1 - ax) \bmod (p-1) \quad \text{and} \quad b_2 = k^{-1}(M_2 - ax) \bmod (p-1),$$

with the same $a = g^k \bmod p$, for two different messages, M_1 and M_2. Then

$$(b_1 - b_2)k \bmod (p-1) = (M_1 - M_2) \bmod (p-1).$$

Key Exchange Protocol

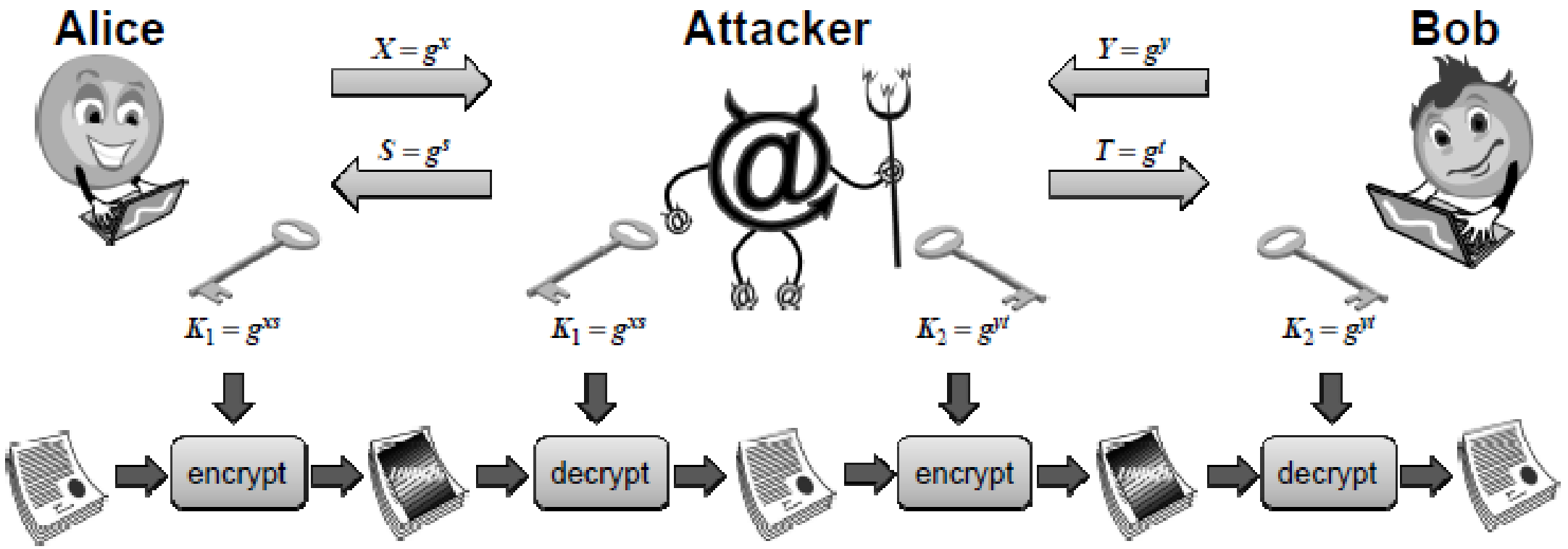
- A ***key exchange protocol*** is a cryptographic approach to establishing a shared secret key by communicating solely over an ***insecure*** channel, without any previous private communication.
  - **Shared** secret key for **symmetric** encryption
- **Assumption :** Attacker cannot actively modify message
  - it can be shown that no key exchange protocol exists if the adversary can actively modify messages sent over the insecure channel
  - Passive eavesdropping on channel

Diffie-Hellman Key Exchange Protocol (DH)

1. Alice picks a random positive number x in Z_p and uses it to compute $X = g^x \bmod p$. She sends X to Bob.
2. Bob picks a random positive number y in Z_p and uses it to compute $Y = g^y \bmod p$. He sends Y to Alice.
3. Alice computes the secret key as $K_1 = Y^x \bmod p$.
4. Bob computes the secret key as $K_2 = X^y \bmod p$.

$$K_1 = Y^x \bmod p = (g^y)^x \bmod p = (g^x)^y \bmod p = X^y \bmod p = K_2.$$

Man In the Middle Attack



Man In the Middle Attack

1. The attacker picks numbers s and t in Z_p.
2. When Alice sends the value $X = g^x \bmod p$ to Bob, the attacker reads it and replaces it with $T = g^t \bmod p$.
3. When Bob sends the value $Y = g^y \bmod p$ to Alice, the attacker reads it and replaces it with $S = g^s \bmod p$.
4. Alice and the attacker compute key $K_1 = g^{xs} \bmod p$.
5. Bob and the attacker compute key $K^2 = g^{yt} \bmod p$.
6. When Alice sends a message to Bob encrypted with the key K_1, the attacker decrypts it, reencrypts it with the key K_2 and sends it to Bob.
7. When Bob sends a message to Alice encrypted with the key K_2, the attacker decrypts it, reencrypts it with the key K_1 and sends it to Alice.

References

- https://en.wikipedia.org/wiki/Substitution%E2%80%93permutation_network
- https://en.wikipedia.org/wiki/Confusion_and_diffusion
- https://en.wikipedia.org/wiki/Advanced_Encryption_Standard
- https://en.wikipedia.org/wiki/Data_Encryption_Standard
- https://www.youtube.com/watch?v=orIgy2MjqrA

Resources

- https://people.scs.carleton.ca/~paulv/toolsjewels.html : Chapter 2
- Goodrich and Tamassia : Chapter 8
- https://www3.nd.edu/~busiforc/handouts/cryptography/cryptography%20hints.html#:~:text=The%20most%20common%20two%2Dletter,words%20are%20the%20and%20and.
- https://crypto.stackexchange.com/questions/13274/the-difference-between-these-4-breaking-cipher-techniques
- https://security.stackexchange.com/questions/261753/why-is-one-time-pad-informationally-secure