

# **COMP4127/COMP7850**

## **Information Security**

### **Symmetric and Asymmetric Encryption**

Some figures are adopted from the Textbook by William Stallings.

After this lecture you shall be able to:

- Identify the difference between symmetric and asymmetric key encryption;
- Name some symmetric and asymmetric key encryption algorithms;
- Identify the differences between different modes of operation;
- Have some hand-on experiences with symmetric and asymmetric encryptions;

# Symmetric vs Asymmetric Key

Most of the classic crypto systems are all symmetric key systems which the sender and the receiver shares the same ***secret key***.

- The sender encrypts a message with a key and the receiver decrypts it using the same key.

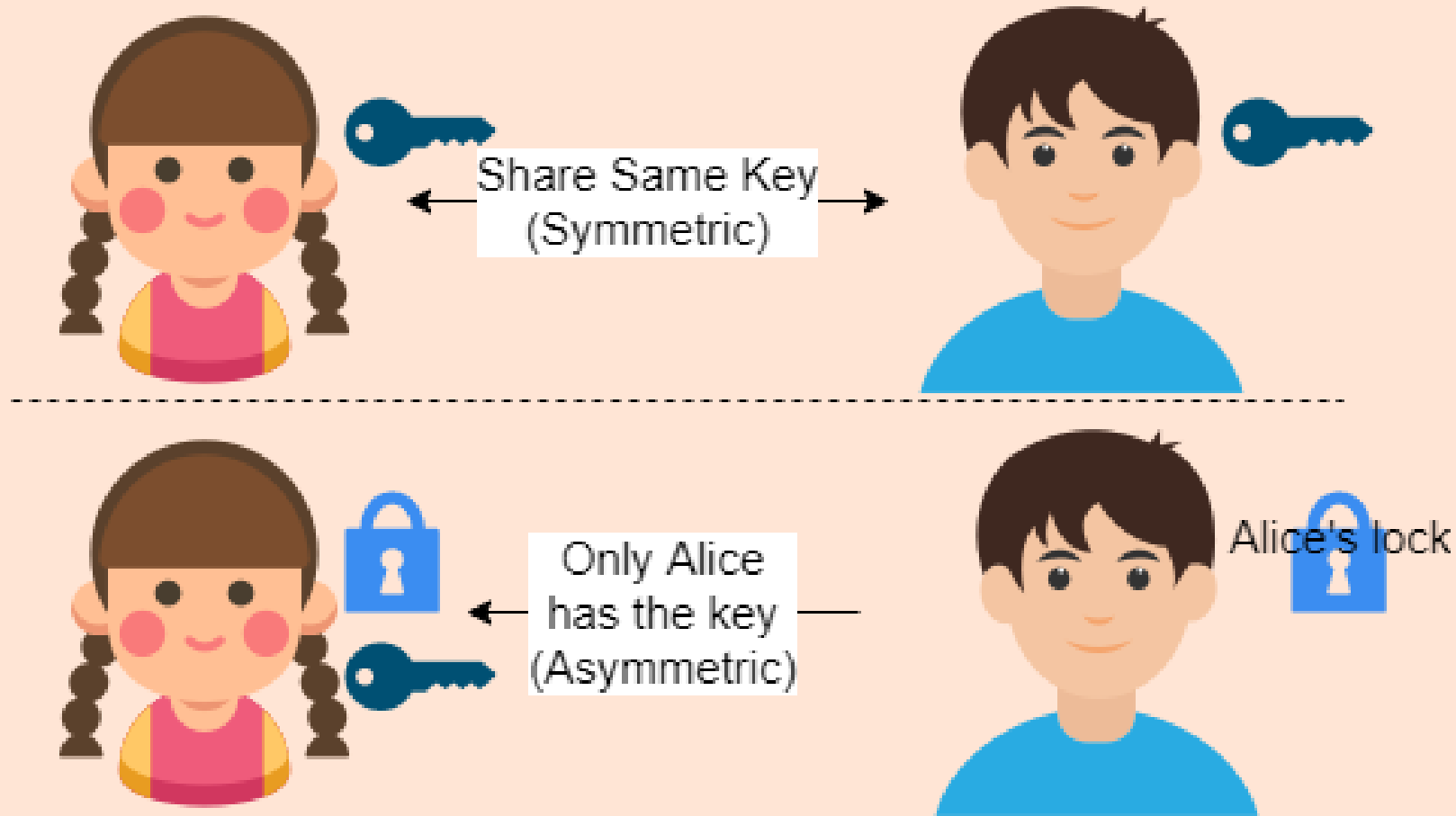
In 1970s, the concept of asymmetric key was proposed such that only the receiver has a ***private key***\* while every other can have her ***public key***.

- The sender encrypts a message using the receiver's public key and the receiver decrypts it using its private key.

Asymmetric key encryption is also called **Public Key Encryption**. Symmetric key encryption can be referred as **Secret Key Encryption**.

\* Don't mix up secret key and private key.

# Symmetric vs Asymmetric Key



# Discussion



Is it safer to disclose my encryption system?

- It is actually more secure if a symmetric encryption algorithm is opened and publicly review.
- Steel is hard because it has been hitted by many hammers.
- Many proprietary cryptosystem has been broken because there are no public review.

- Two types of symmetric encryption:
  - Stream Cipher: the smallest unit to encrypt is one-bit.
  - Block Cipher: the smallest unit to encrypt is a block (a typical block-size:64)
- Most stream ciphers are less secure than block ciphers.
- Most symmetric encryptions scramble the message by substitution and rearrangement for multiple-times.
  - There isn't any **hard problem** behind the algorithm.
- The security of a symmetric encryption is bounded by the key size. If a key has 64-bits, it can be broken with at most  $2^{64}$  operations.

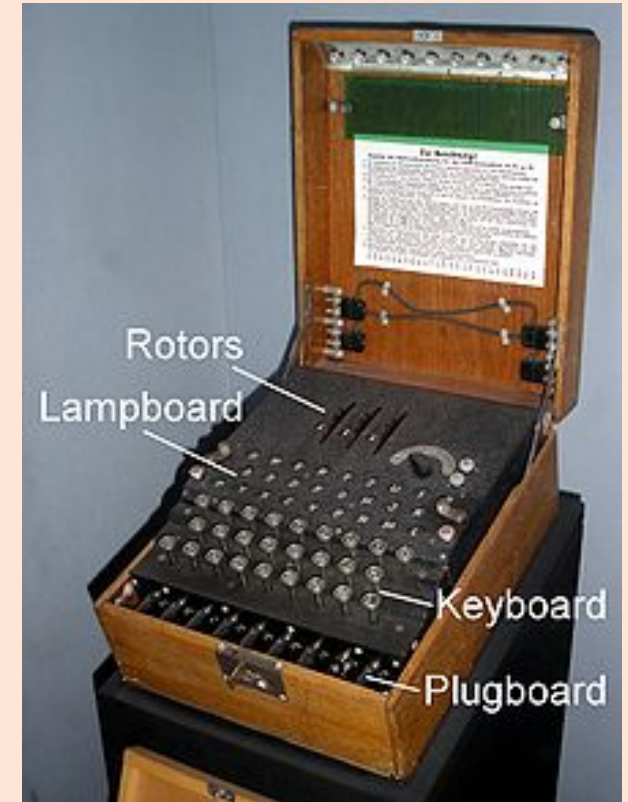
# Symmetric Encryption Evolution

Some serious crypto systems were known because of wars. During WWI to WWII, some famous systems like **Enigma** (Axis), **CCM**, **Typex** (Allied) were used.

These systems are Rotor-type design

See movies:

- U-571



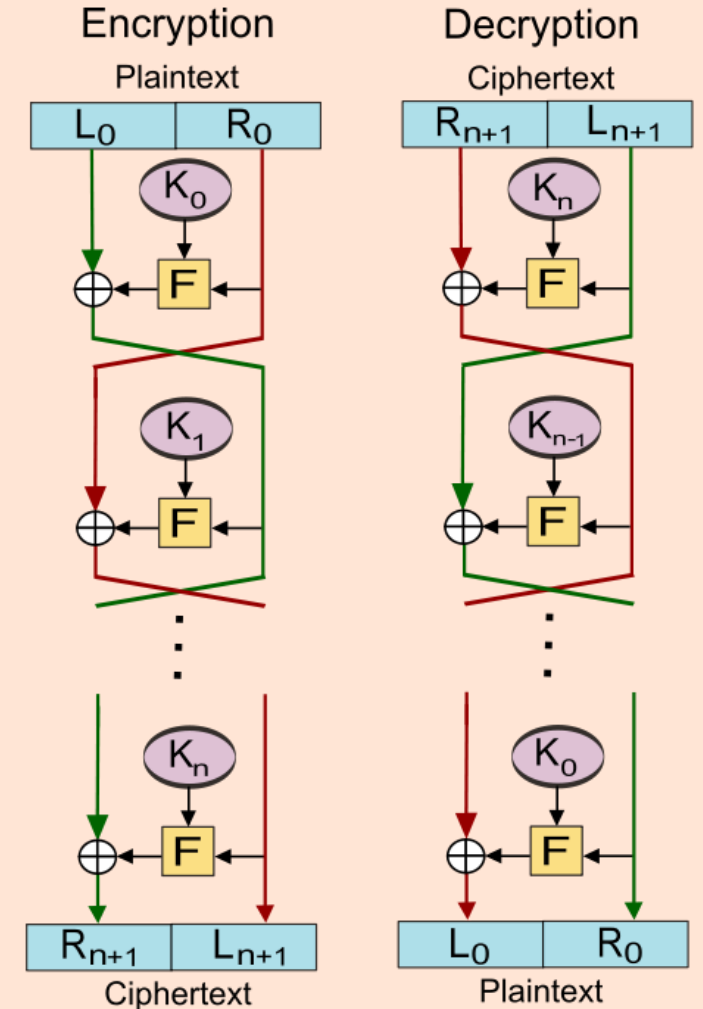


# Symmetric Encryption Evolution

1975: Data Encryption Standard (DES) by IBM, adopted by National Bureau of Standard NBS (later known as NIST).

- It is based on **Feistel's transform**
- The function  $F$  is a one-way, non-invertible function
- Block-size = 64 bits (Encrypt 64 bits at a time).
- Key is just 56-bits long.

1999: DES was cracked in 22 hours.



# How Feistel's transform work?

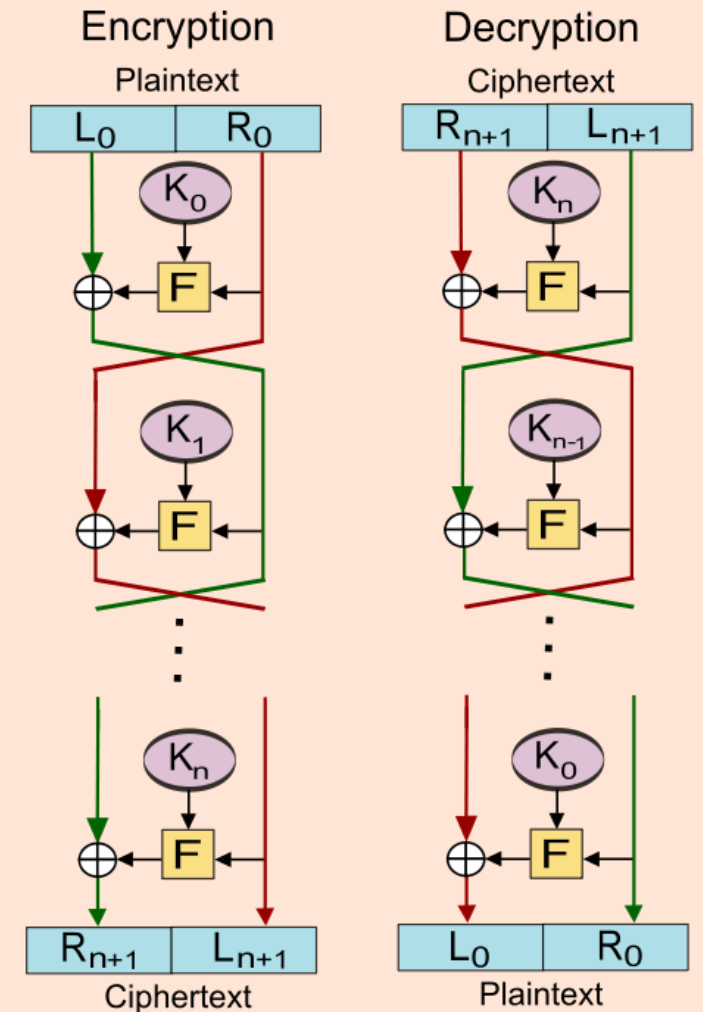
- Given the fact that  $A \oplus B \oplus B = A$ 
  - e.g.  $1001 \oplus 1100 \oplus 1100 = 0101 \oplus 1100 = 1001$
- $X \oplus Y = Z \iff X = Z \oplus Y$

We know

- $L_{j+1} = R_j$
- $R_{j+1} = L_j \oplus F_{K_j}(R_j)$

Thus, when we decrypt

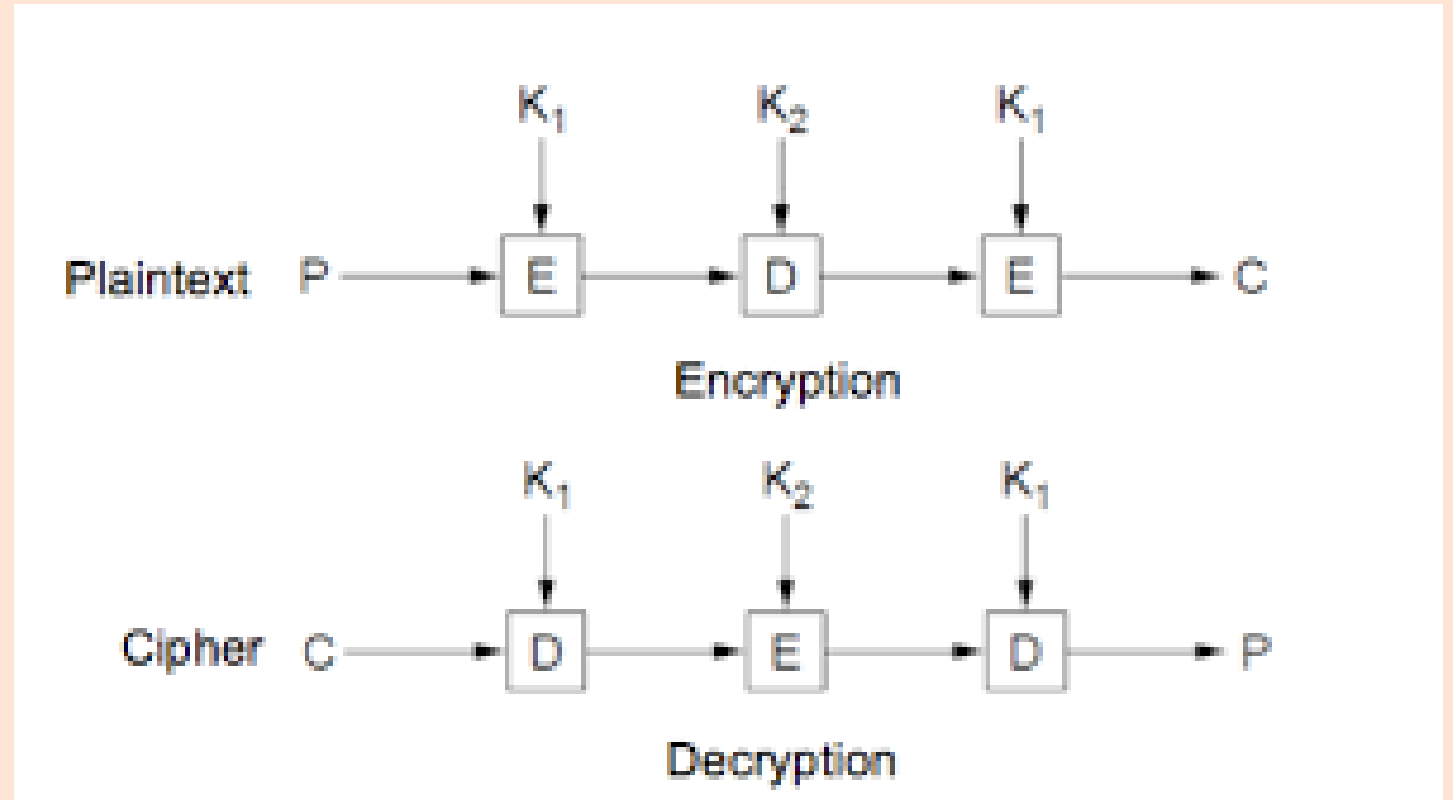
- $R_j = L_{j+1}$
- $L_j = R_{j+1} \oplus F_{K_j}(R_j)$



# Symmetric Encryption Evolution - 3DES

- Late 1990s: **3DES** as an *interim* solution. Key size =  $56 \times 2$
- The idea is very simple, it uses two keys to encrypt-decrypt-encrypt to produce a cipher.
- Triple times in computation, double times in storage.

Both DES and 3DES should not be adopted today.

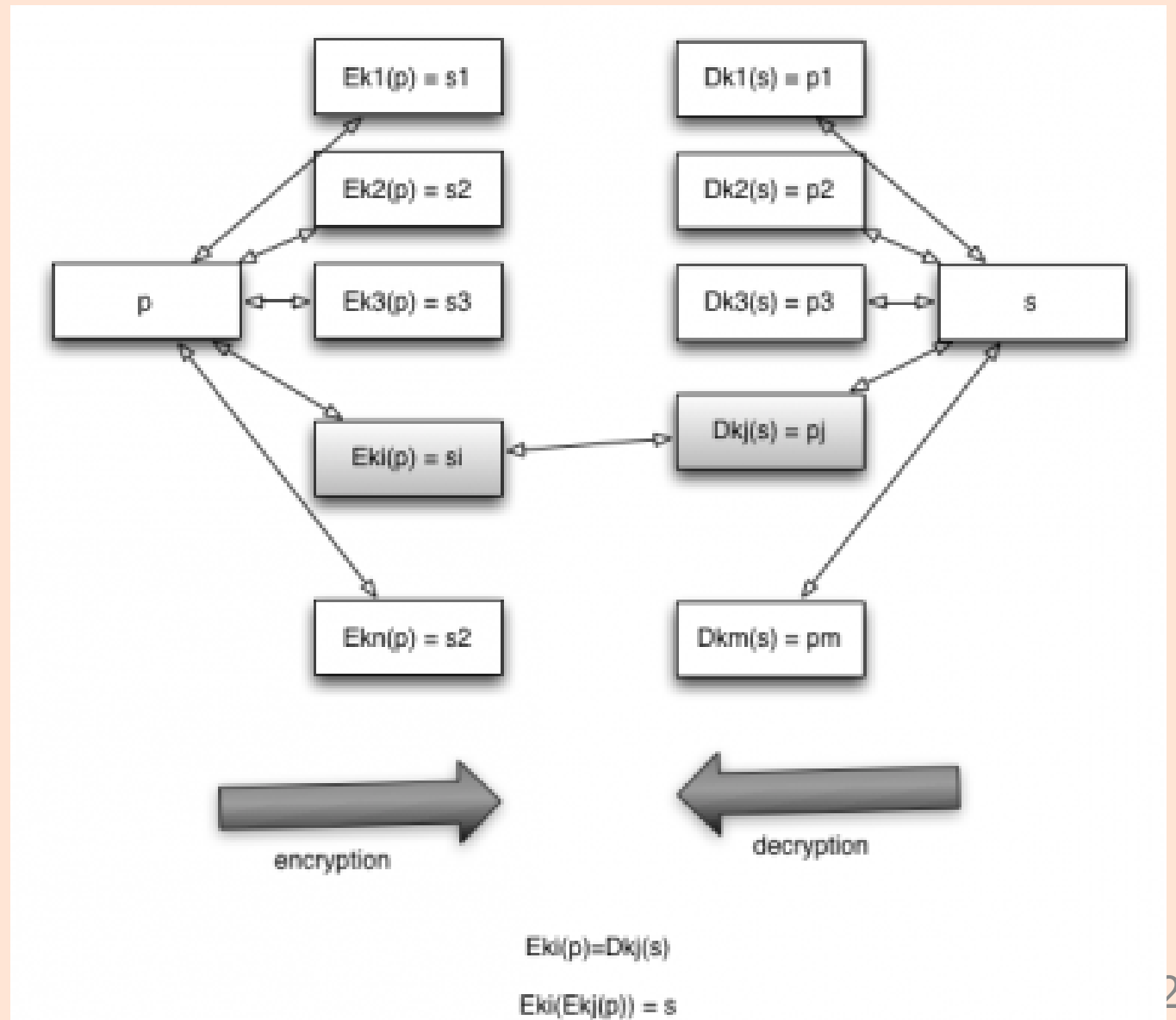


# Why not just two?

Meeting-in-the-middle attack:  
With a known plaintext pair, the attacker casts a brute-force on the plaintext  $p$  and cipher  $s$ :

- $p \rightarrow s_1, s_2, s_3, \dots, s_k$
- $s \rightarrow p_1, p_2, p_3, \dots, p_k$

Match  $s_i = p_j$



# Symmetric Encryption Evolution

2001: Advanced Encryption Standard, AES, was released as a NIST standard.

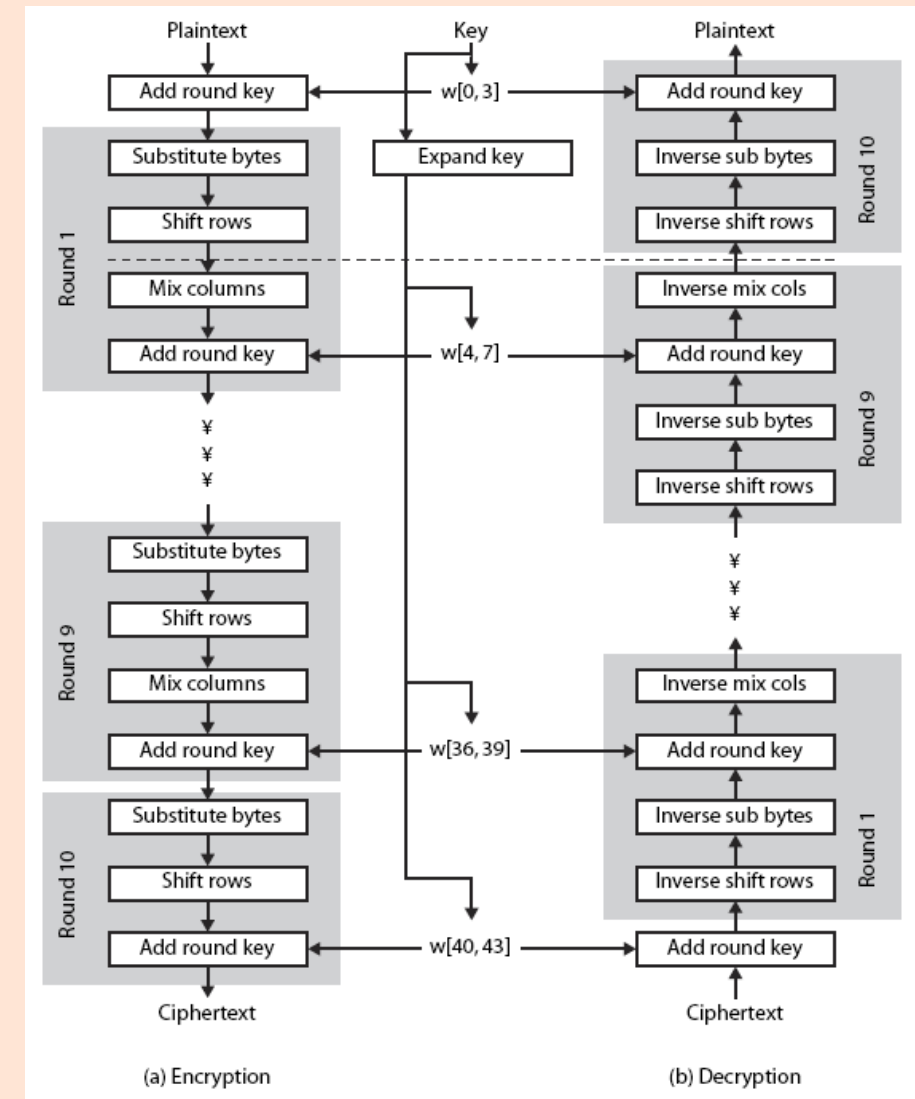
- NIST use 5 years to hold a competition and select **Rijndael**, by Daemen and Rijmen, from 15 candidates.
- Key size: 128, 192 or 256 bits.
- Block size: 128 bits.

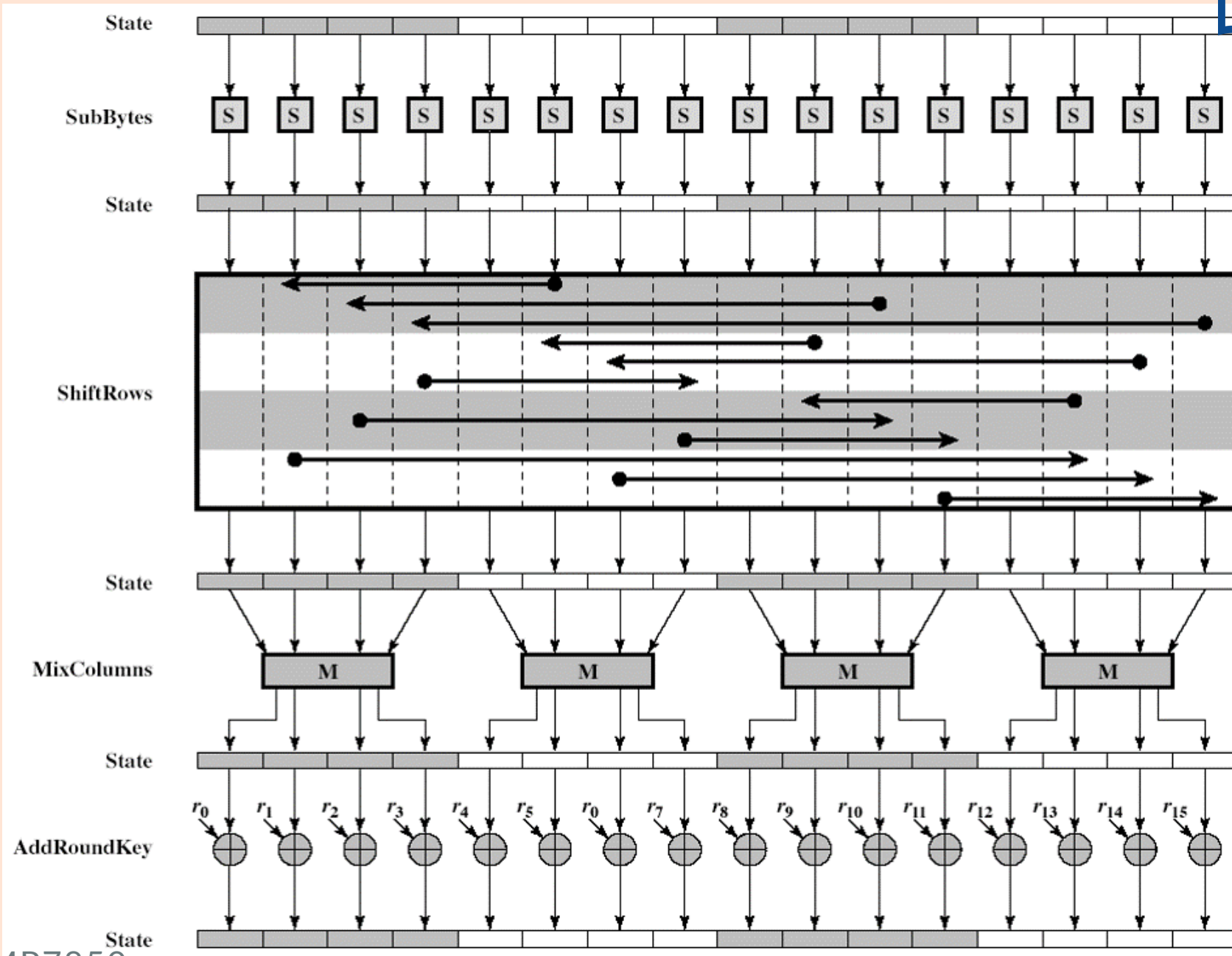
2008: x86 processors included AES into their instruction set.

2021: still using (e.g. WPA, SSL)

# Overview of AES

Key Size (words/bits)	Round	Expanded key size (words)
4/128	10	44
6/192	12	52
8/256	14	60







# Oh wait

Before you decide to go to sleep, read this:

<https://heathead.wordpress.com/2013/03/12/understanding-aes-encryption-for-real-h4x0rs/>





# Confusion and Diffusion Property of AES

- **Confusion:** each bit of the ciphertext is affected by several parts of the key.
  - Say, a naïve encryption only does shuffling and  $\oplus$  (or other linear operations), even if it is performed many times, it can be reduced to only a shuffle and one  $\oplus$ .
- **Diffusion:** change a single bit of the plaintext (or the key), statistically half of the bits in the ciphertext will be changed.
  - It is also known as the *Avalanche* effect.



Let say an attacker has a set of key  $S = \{k_1, k_2, k_3, \dots\}$ , what is the probability that he has correctly guess the key?

$$\frac{|S|}{2^{128}}$$

We consider  $2^{80}$  operations are very impossible to perform, while we consider a probability of  $2^{-30}$  is impossible to happen.

🛡️ AES defences against this!

Rough Idea: Create many pairs of plaintext like

- $M = 1100111010101010 \dots 1110\underline{1}1101 \dots$
- $M' = 1100111010101010 \dots 1110\underline{0}1101 \dots$

and obtain the ciphertexts  $C, C'$ . Compare the result  $C \oplus C'$  and spot the patterns.

🛡️ AES defences against this!

- Smallest token 8bits (ready for lightweight devices like sensors, IoT devices)
- Many reuseable circuit (S-Box is reused in two sections)
- Allow parallel processing in certain stage.

- 2000: Attacks on 7 rounds version of AES-192 (Gilbert and Minier)
- 2004: Attacks on 5 and  $^6$  rounds version of AES-128 (Alex Biryukov)
- 2008:  $^$ Attacks on 8 rounds version of AES-256 (Demirci and Selçuk)
- 2009:  $^$ Attacks on full version of AES Side-Channel Attack



These attacks require ridiculously many computation power and no successful implementation (or plan of implementation) has yet been announced.

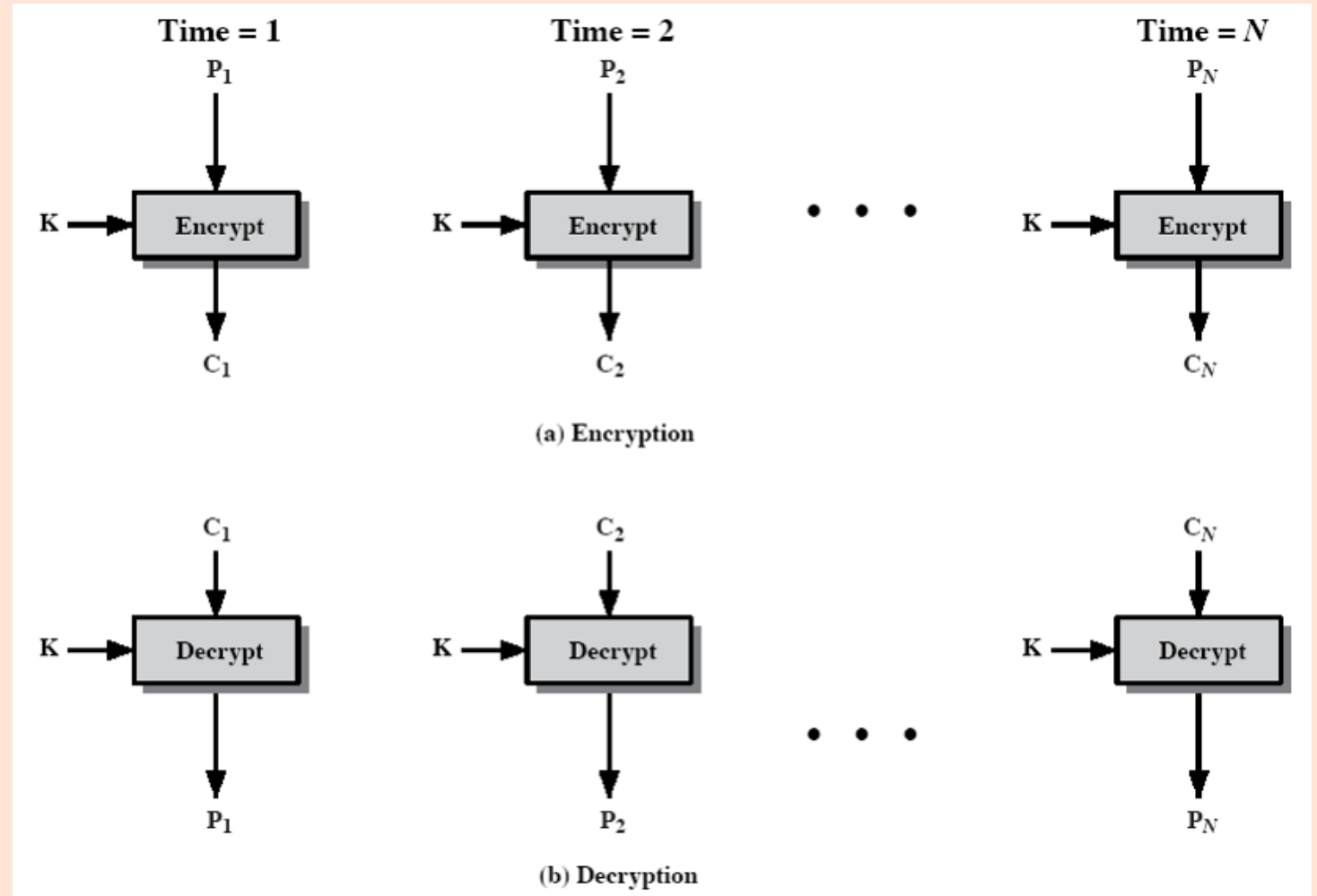
Remember AES only encrypt 128 bits (~24 alphanumeric characters). We need some ways to handle more realistic applications.

NIST defined 5 **modes of operation** for different applications:

- ECB
- CBC
- CFB
- OFB
- CTR

# ECB - Electronic Codebook Mode

- Message is broken into independent blocks for encryption.
- Each block is encrypted independently and can be decrypted independently:
  - $C_i = E_k(P_i)$
  - $P_i = D_k(C_i) = D_k(E_k(P_i))$
- Usage:
  - Secure transmission of single values (e.g. key)



## Pros:

- Simple.
- Message can be decrypted in different order.
- Can be encrypted/decrypted in parallel.
- Error in transmission will not propagate to other blocks.

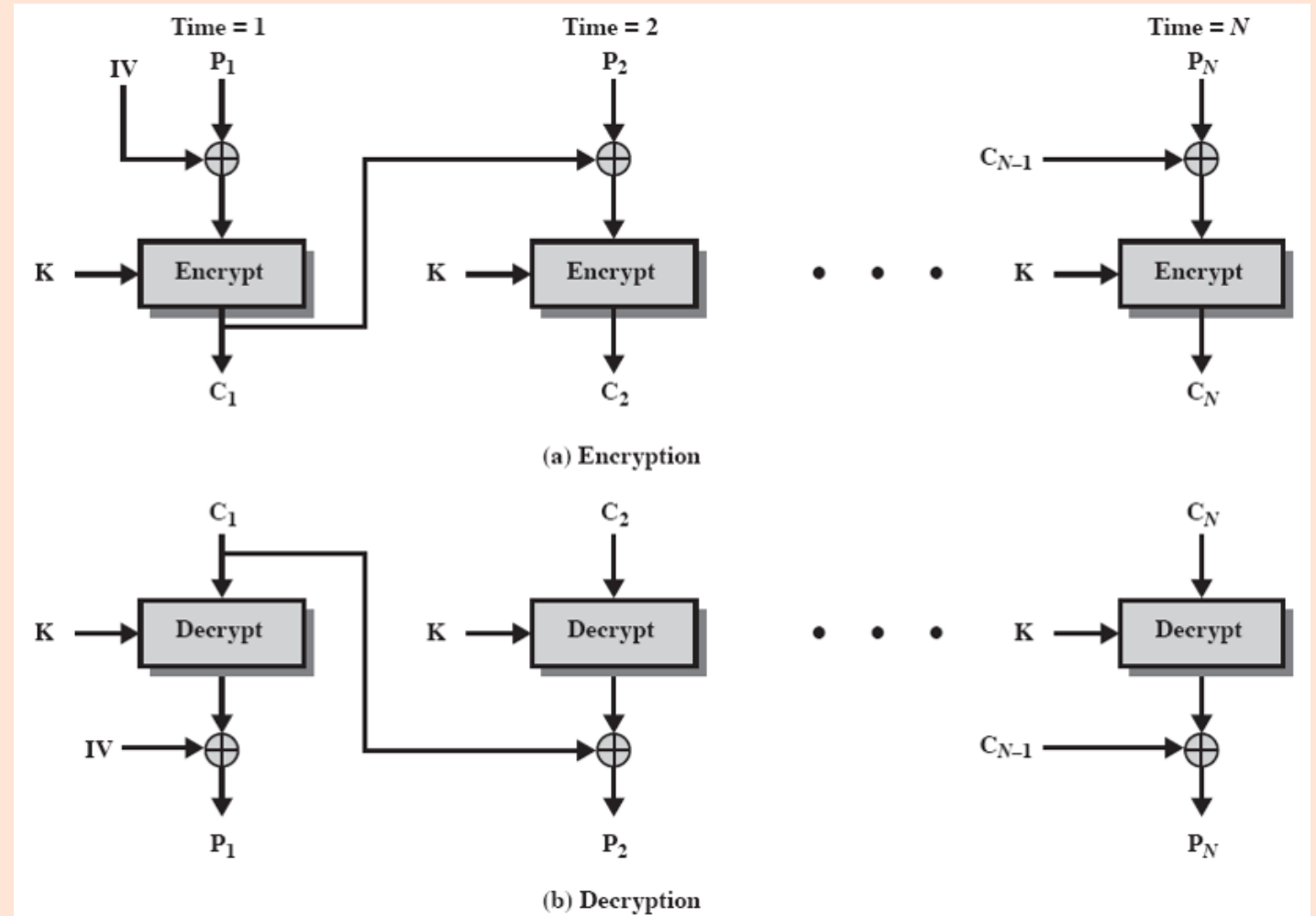
## Cons:

- Repeated encryption can be detected very easily.



# CBC - Cipher Block Chaining Mode

- Encryption requires input from previous block.
- Use an Initial Vector (IV) to start the encryption
  - $C_i = E_k(P_i \oplus C_{i-1})$   
where  $C_0 = IV$
  - $P_i = D_k(C_i) \oplus C_{i-1} = D_k(E_k(P_i \oplus C_{i-1})) \oplus C_{i-1}$
- Usage: bulk data encryption



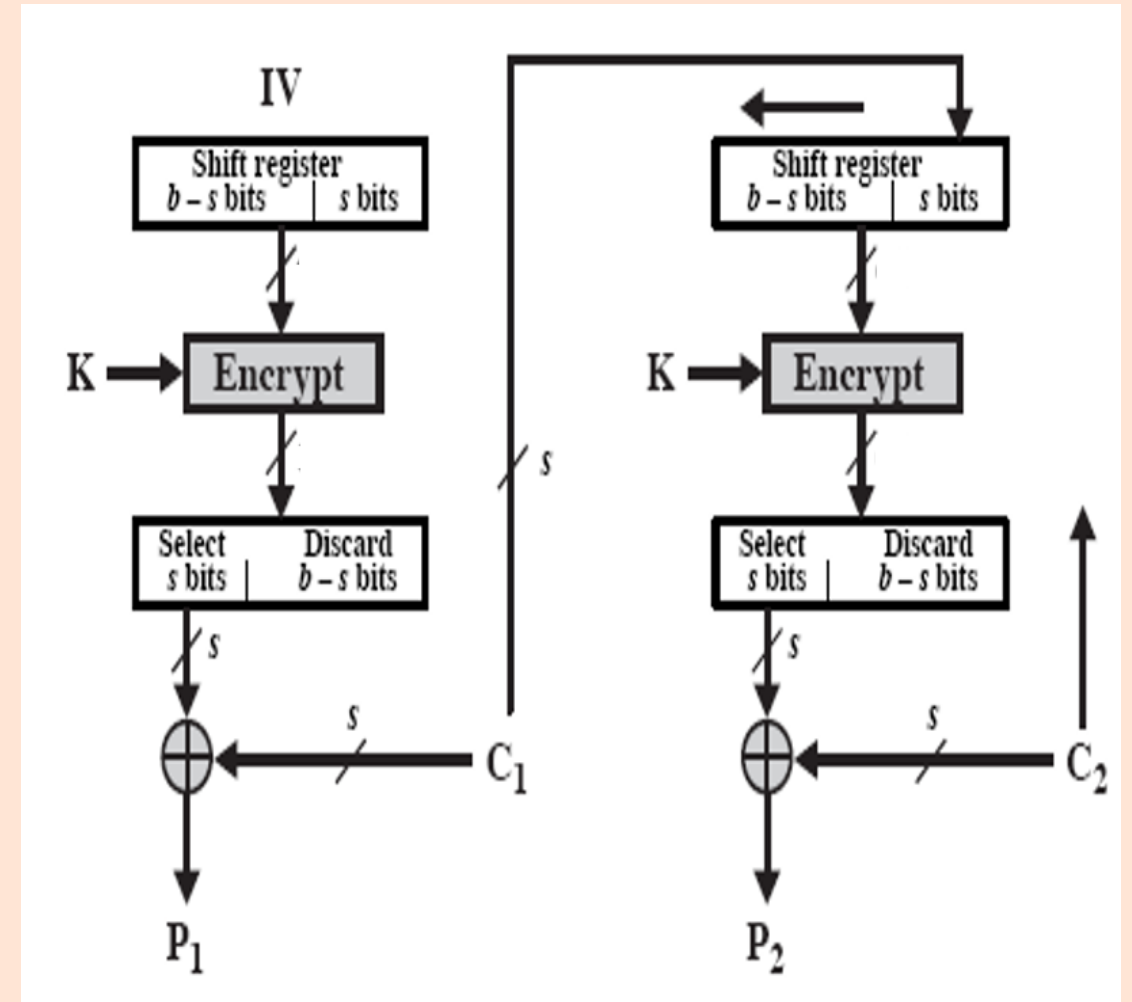
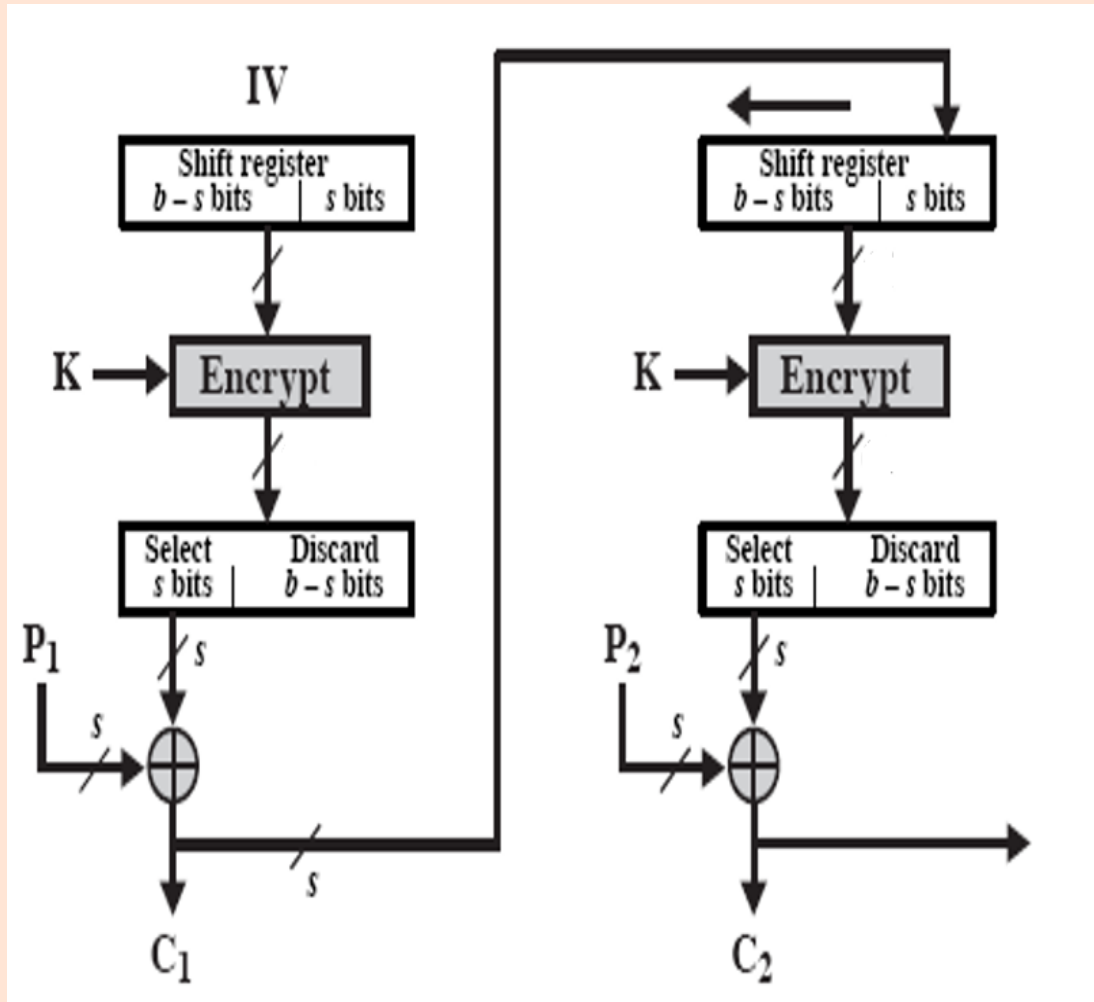
## Pros:

- High Security
- Repeated block in a session will not be detected easily as ECB.
- Can change IV for better security if same content is sent in first block.

## Cons:

- IV management; Attacks on IV is possible.
- Message must be encrypted in sequence and no parallel processing is possible

# CFB - Cipher Feedback Mode



- A **shift register** stores  $b$  bits. Each time the first  $s$  bits are discarded and appended new  $s$  bits to its end.
- Allow a smaller size of block ( $s$  bits) than block size of the encryption.
- Trim some bits after DES/AES and  $\oplus$  the remaining with the plaintext.
- Use a shift register  $R_i$ 
  - $R_i = \text{Right}_{b-s}(R_{i-1}) || C_{i-1} \# b-s + s$
  - $C_i = \text{Left}_s(E_k(R_i)) \oplus P_i \# s \text{ bits only}$
  - $P_i = \text{Left}_s(E_k(R_i)) \oplus C_i \# s \text{ bits only}$
- CFB-1, CFB-8, CFB-128 means  $s = 1$  or  $8$  or  $128$

## Pros:

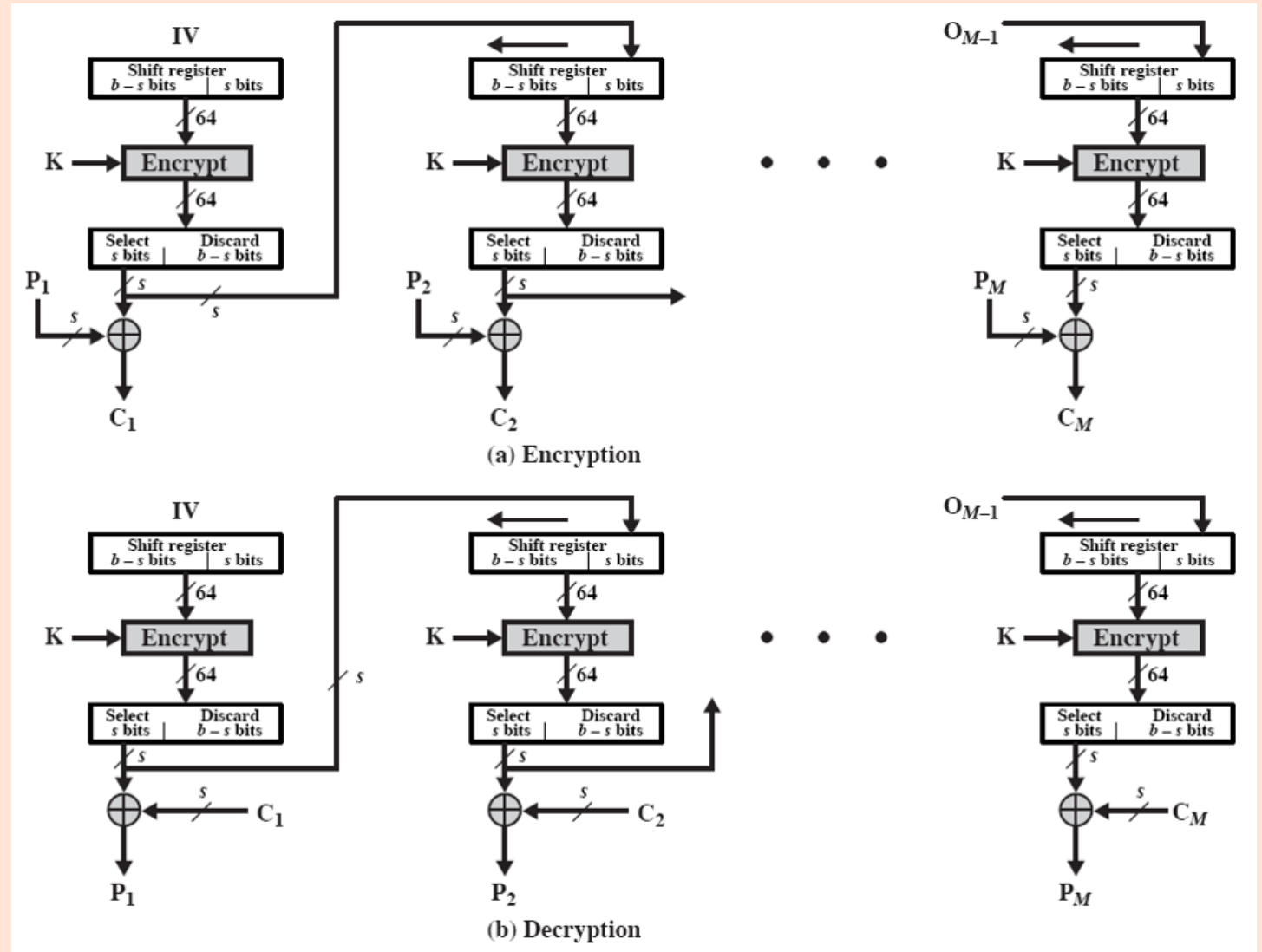
- Flexible choose of bits
- Only encryption is needed (no need to implement decryption)

## Cons:

- Parallel processing is still not possible
- Error propagation, but will fixed after  $\frac{b}{s}$ -blocks

# OFB - Output Feedback Mode

Noted that OFB is different than CFB in the way that the shift register after encryption and trimming is directly fed to next register.



- Use a shift register  $R_i$ 
  - $T_i = \text{Left}_s(E_k(R_i))$  # s bits only
  - $R_i = \text{Right}_{b-s}(R_{i-1}) || T_{i-1}$  # b-s + s
  - $C_i = M_i \oplus T_i$  # s bits only
  - $P_i = T_i \oplus C_i$  # s bits only

$T_i$  is actually independent of  $P_i$ , it can be precomputed

Pros:

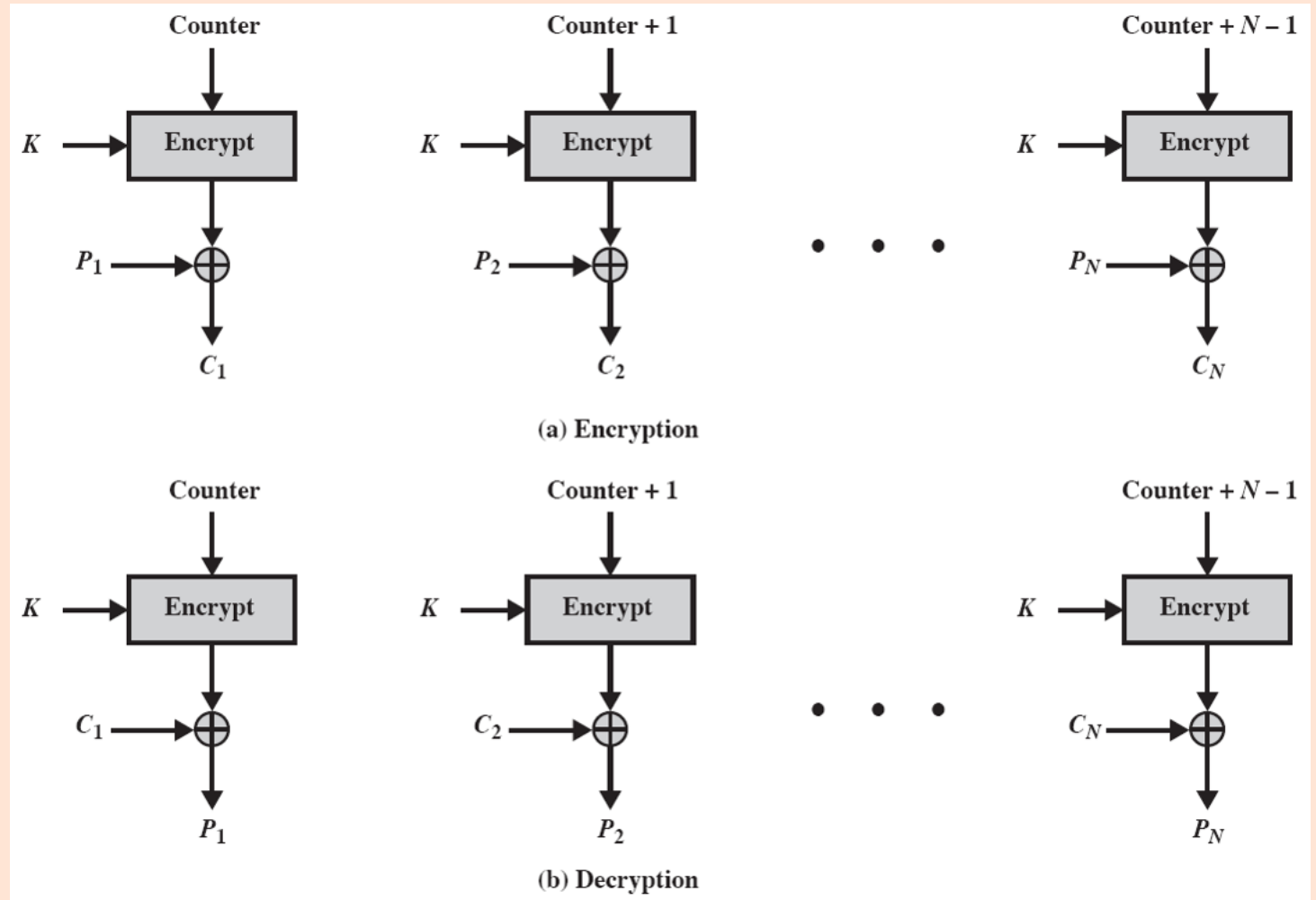
- Can be used for stream cipher (encrypt one bit and send one bit)
- Insensitive to transmission errors

Cons:

- Encrypted data can be easily modified

# CTR - Counter (CTR) Mode

- Similar to OFB but encrypts counter value rather than feedback value
- **Counter** is a value agreed between the sender and receiver. It can be recorded or sent separately.
- It is important that the counter is never reused.





- Act as a stream-cipher (no need to wait until a whole block to send)
  - $C_i = E_k(i) \oplus M_i$
  - $M_i = D_k(i) \oplus C_i$

Usage: stream cipher

Pros:

- Parallel encryption and decryption are possible
- Preprocessing is possible
- Resilient to data lost / transmission error

Cons:

- Ciphertext is easily malleable
- Reusing counter (with the same key) is very dangerous

- **Galois/Counter Mode (GCM):** a variant of counter mode while it also produce an additional **Auth Tag** to ensure the cipher is authentic.
  - Used in IPSec/TLS
- **Counter with cipher block chaining message authentication code (CCM):** combining CTR model and CBC-MAC to provide confidentiality and authenticity of data.
  - Used in WPA2/IPSec/TLS/BLE

**OpenSSL** is a very convenient tool to perform standard crypto operations in both command line and program. Installing OpenSSL under Ubuntu is very convenient

```
sudo apt-get install openssl
```

You might also consider to run a docker image to taste it:

- Windows


```
docker run --rm -it -v "%CD%:/home" emberstack/openssl sh
```

- macOS

```
docker run --rm -it -v "$(pwd) :/home" emberstack/openssl sh
```

This will make you run into a linux instance.

## Survival Minimum Linux operations:

- `cd`
- `more`
- `rm`
- `cat`
- symbols `>`, `*`
- helper keys , `tab`

Guide here: <https://www.computerhope.com/issues/chshell.htm>



Ask us on Piazza if it does not work!

OpenSSL provides a list of ciphers for you to choose, type `openssl enc -ciphers` to check the supported ciphers for your openssl.

```
Usage: enc [options]
Valid options are:
  -ciphers          List ciphers
  -in infile        Input file
  -out outfile      Output file
  -pass val         Passphrase source
  -e               Encrypt
  -d               Decrypt
  -nopad           Disable standard block padding
  -base64          Base64 encode/decode, depending on encryption flag
  -k val           Passphrase
  -kfile infile    Read passphrase from file
  -K val           Raw key, in hex
  -iv val          IV in hex
  -pbkdf2          Use password-based key derivation function 2
  -*              Any supported cipher
```

For example, you want to encrypt a file `input.pdf` using AES-128-ECB, you will need to prepare a key with 128 bits long. You can either prepare 128-bits key using hexadecimal string (each symbol carry 4 bits, need exactly 32 symbols) or use a *passphrase* to generate the key.

Encrypt with hex key:



```
openssl enc -K 00112233445566778899aabbccddeeff -in input.pdf -aes-128-ecb -out cipher.enc
```

Decrypt with hex key:

```
openssl enc -d -K 00112233445566778899aabbccddeeff -in cipher.enc -aes-128-ecb -out decrypt.pdf
```

Encrypt with passphrase:

```
openssl enc -k kevin_is_sleeping -in input.pdf -aes-128-ecb -out cipher.enc
```

  How do we print our data (txt/pdf/img/dat/iso) on paper or other txt processor (e.g. programming code/html)?

- Representing data in binary digits are not suitable for printing (too long).
  - 11101101001010101010...
- Representing data in hexadecimal digits is better.
  - ED2AA... (4 times shorter than binary)

Binary	1110	1101	0010	1010	1010
Hexadecimal	E	D	2	A	A

## Can we do better?

- Representing data in ASCII character is not suitable for printing as some characters do not print.

# Base64 encoding

- Base64 encoding use `[A-Z][a-z][0-9]+/`, 64 different combinations.

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	θ
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		=									



# Playing with Base64

To convert a binary file into base64 representation

```
base64 abc.txt
```


To convert a binary file into base64 representation and save it to another file

```
base64 abc.txt > newfile.txt
```

To convert a base64 file back to binary

```
base64 -d newfile.txt
```

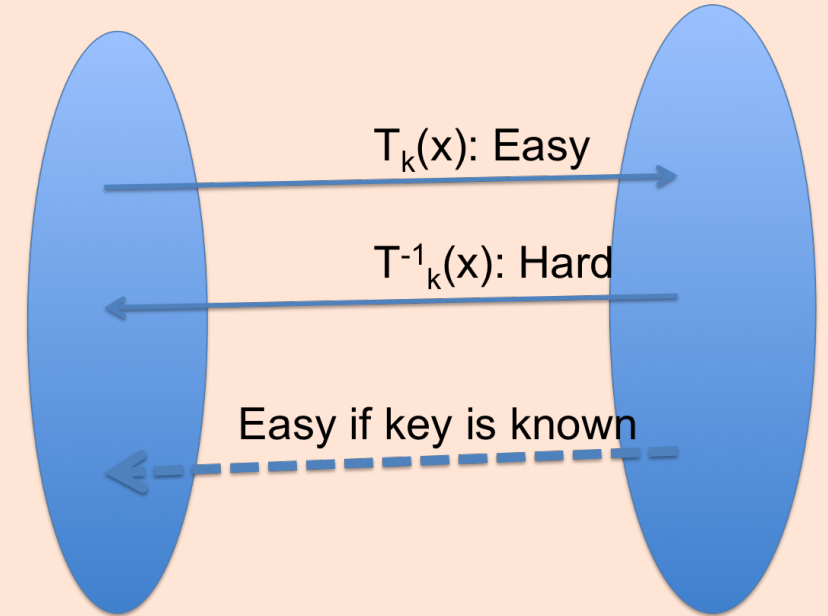
# Asymmetric Key and Trapdoor function

Recall that Asymmetric key encryption works like a Padlock , everyone can lock (encrypt) but only the one with \*private key can unlock (decrypt).

Trapdoor functions are essential building blocks of modern cryptography. We need some advanced math in *Numbers Theory* to build such trapdoor function.


Rough Idea of a trapdoor function  $T_k$ :

1. Compute  $T_k(C)$  is always easy. (similar to the computable property of a MAC)
2. Compute  $C$  from  $T_k(C)$  is difficult without knowing  $k$ . (similar to one-wayness property of a hash function)
3. Compute  $C$  from  $T_k(C)$  is easy if we know  $k$ .



# If trapdoor function exists...

Assume a trapdoor function  $T_k$  exists, we can perform public key encryption in such a way:

1. Alice generates a function  $T_k$  and keeps  $k$  in private.
2. She publishes the function  $T_k$  as her public key .
3. Bob generates a random session key  $s$  and computes  $T_k(s)$ .
4. Bob encrypts the private message  $m$  using symmetric key encryption with the session key  $s$ .
5. Bob sends both  $T_k(s)$  and the cipher to Alice.
6. Alice computes  $s$  from  $T_k(s)$  with her private key  $k$ .
7. Alice decrypts the cipher using symmetric key encryption with  $s$ .

# Does trapdoor function really exist?

## Factorization

Factorization problem refers to compute all prime factor of a compound number  $n$ . That is, output  $p_1, p_2, \dots, p_n$  such that  $n = p_1^{k_1} p_2^{k_2} \dots p_n^{k_n}$ .

Factorization problem is believed to be a hard problem if  $n$  is sufficiently large and does not contain small prime factors [and ...]

There are more constraint to make factorization be a hard problem. But we don't discuss in this course.

$$15 = 5 \times 3, 3441 = 37 \times 93$$

How about

1143816257578888676692357799761466120102182967212423625625618  
4293570693524573389783059712356395870505898907514759929002687  
9543541

The following equation can be computed very easily even if  $n$  is a very large number.

$$P^3 \bmod n$$

However, only if you know the factorization of  $n$ , you can compute the following equation very easily:

$$C^{\frac{1}{3}} \bmod n$$

So Alice would create two large prime numbers and keep them in secret. (RSA)

$$T_k(C) = C^3 \bmod n$$

$$T_k^{-1}(C) = C^{\frac{1}{3}} \bmod n$$

# Other examples of trapdoor functions

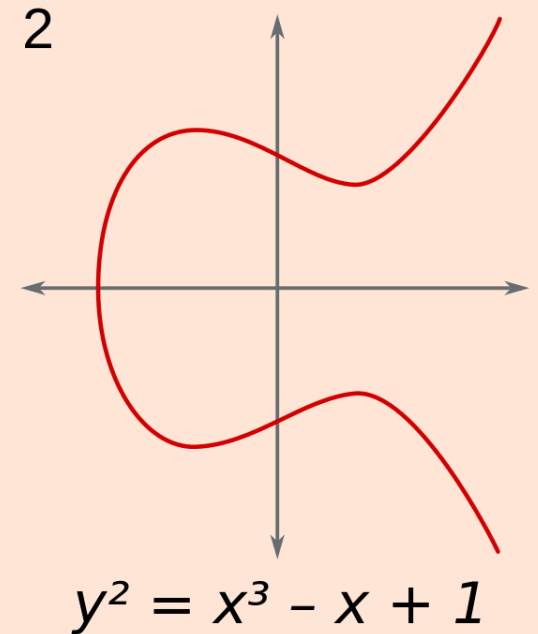
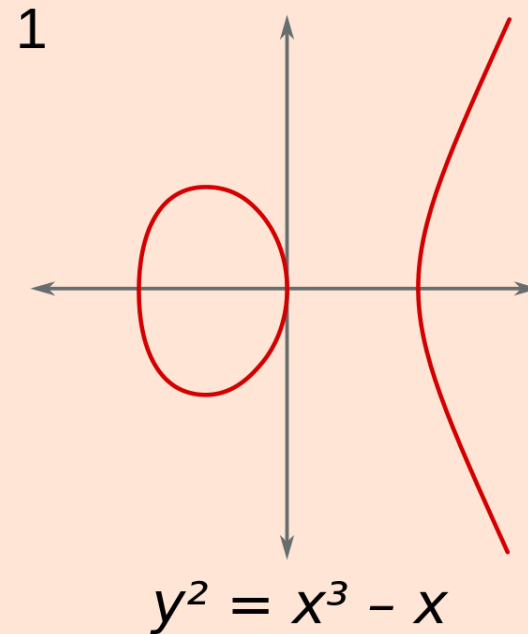
**Discrete Log Problem (DLP):** given an integer  $g$  and large integers  $p$ , and  $g^x \bmod p$ , it is difficult to compute  $x$ .

**Diffie-Hellman Problem (DH):** given an integer  $g$  and large integers  $p$ ,  $g^a \bmod p$ ,  $g^b \bmod p$ , it is difficult to compute  $g^{ab} \bmod p$ .

**Elliptic Curve Discrete Log Problem (ECDLP):** a discrete log problem on elliptic curve

**Elliptic Curve Diffie-Hellman Problem (EC-DH):** a Diffie-Hellman problem on elliptic curve.

These things are good to hear, but no need to understand in our course.



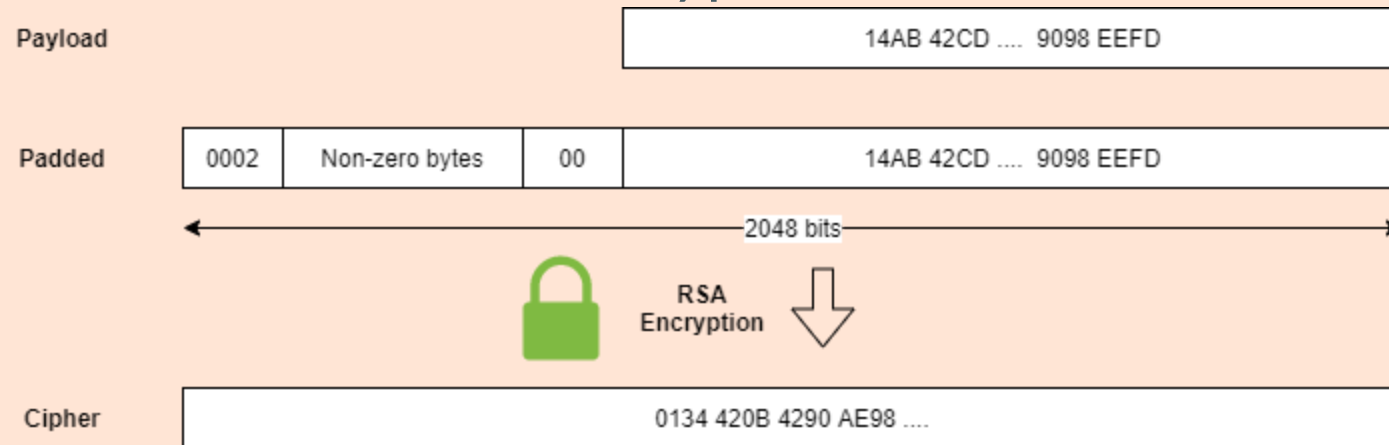
# Public Key Algorithms - RSA

- Ron Rivest, Adi Shamir, and Len Adleman proposed RSA encryption algorithm in 1977.
- The algorithm is believed to be secure as long as factorizing problem is difficult.
- Nowadays, we need at least 2048-bits long RSA. Nevertheless, it starts to be deprecated.
- Starting from TLS 1.3, RSA is not included in its standard.



# RSA Padding Schemes

RSA is a **deterministic** algorithm (every time it produces the same ciphertext, if the same public key and same message are used). It needs an appropriate **padding scheme** to randomize the encryption.



Famous Padding schemes:

- PKCS 1.5 (figure. Deprecating)
- OAEP (recommended)
- SSLv2



# How does public key encryption work?

- A user Alice generates a pair of public key  $P_k$  and private key pair  $x$ .
- Alice sends the public key  $P_k$  to her friends, Bob and Carol.
- Alice keeps the private key  $x$  to herself only.
- Bob uses public encryption algorithm to encrypt a message with  $P_k$ .
- Bob sends the ciphertext to Alice.
- Alice decrypts the message using the private key  $x$ .
- Carol, who knows the public key  $P_k$ , cannot decrypt the ciphertext sent from Bob!



The encryption algorithm involves a trapdoor function.

# Comparison the two Encryptions

	Public Key Encryption	Symmetric Key Encryption
Computation Performance	Slow (+ Key generation)	~10x-100x faster
Storage Requirement	1 set of private key	$k$ set of 128 bits up
Message Size per block	< 2048 bits	128 bits
Key Deployment Requirement	Authenticity	Confidentiality
Security	strong - reduction to a known hard problem	strong - examined by cryptographers

We need to generate a public key and private key pair. Assume we are working with RSA.

```
openssl genrsa -out private.pem 2048
openssl rsa -in private.pem -pubout -out public.pem
```

The above statements create two files: `private.pem` and `public.pem`. Make sure you are keeping the `private.pem` secure so that no other can access it. The public key looks like the following, encoded in **Base64** format.

```
-----BEGIN PUBLIC KEY-----
MIIBIjANBgkqhkiG9w0BAQEFAAOCAQ8AMIIBCgKCAQEAWwhQqH8lfVTZ1OY35BIS
JKjxvAEBkhZ+fky5hCDm0w56Dkt0abFCIWMQnrgnc1aP+1LPpceegl vYGVRl9Vma
r5x5cG+AlpUUixI+MG1Iv2/IxoGsP1nlX4kKLwKGDbSQuZHFGMw9mXPkS/NND4RL
NuoIvuvvm2LIp5iYnsyaMh/UnsxNPg1V2kmL8KuUssd5gCC+lnwGkX2vX+w6Iy4qY
MtvqmGhzPRXvUK6rc7UxEbACCa6s81GOoQ0o9lrpPwa7InDBZGu679VA62TSfsRi
3guTJsBlcDbRzjD2GhIBHWR7cd6tdl888tM0/E1U7MUa8I06t1zXCZwZ3HHDQwlI
CswIDAQABOMP7850
-----END PUBLIC KEY-----
```

```
-----BEGIN RSA PRIVATE KEY-----
MIIEpAIBAAKCAQEAwWhQqH81fVTZ1OY35BISJKjxvAEBkhZ+fky5hCDm0w56Dkt0
abFCIWMQnrgnc1aP+1LPpceeglVYGVRL9Vmar5x5cG+AlpUUIxI+MG1Iv2/IxoGs
P1n1X4kKLwKGDbSQuZHFGMw9mXPkS/NND4RLNuoIvuvvm2LIp5iYnsyaMh/UnsxNP
g1V2kmL8KuUssd5gCC+lnwGkX2vX+w6Iy4qYMTvqmGhzPRXvUK6rc7UxEbACCa6s
81GOoQ0o9lrpPwa7InDBZGu679VA62TSfsRi3guTJsBlcdbRzjD2GhIBHWR7cd6t
d1888tM0/E1U7MUa8I06t1zXCZwZ3HHDQwlIswIDAQABAoIBAQC86fv/CpHM0fji
.....
boUD/VQM4m22An4FfxR+b9RiqE7UCGNLOdl2WCzEJd9rrZclLVE4NpKfUPgMnY/l
xdSL2wKBgQC4h9/OAtUMGudlrFL1CAFLFUuf21Rqi/xICREkQqmwoSX9JvmTfoCK
/1cs3SI3BzjnAdA7y38avbMUMR+kMlK8ubWVe52jCjDP6lewSsHkU19JfMx2NTg1
uvzdWOEQi/R54C5GDRdOb1uX8i9XbAE/M8M/JDJjFsK0uyQcgbtpKQ==
-----END RSA PRIVATE KEY-----
```

The size of the private key is larger than the public key where the **private key is always capable to deduce the public key, but not vice versa.**

Remember RSA (or other public key encryption) are supposed to be used to encrypt only a little amount of data (e.g. data < 2048 - 64 bits; -64 because of padding).

## Encrypt

```
openssl rsautl -encrypt -pubin -inkey public.pem -in plaintext.txt -out rsa_cipher.enc
```

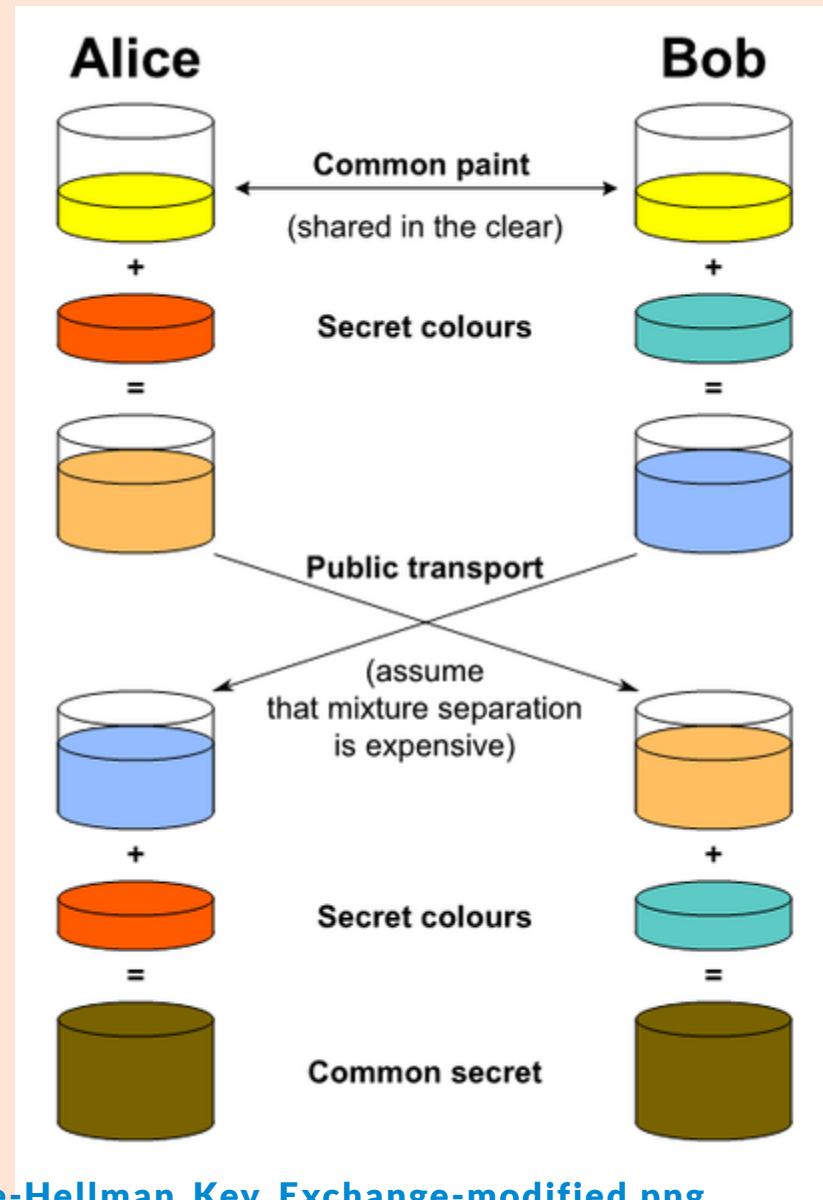
By default openssl (version 1.1.1), use PKCS1.5 for RSA padding. You can make it explicit by adding one of the following flag

- `-pkcs`
- `-oaep`
- `-ssl`
- `-raw` (no padding)

## Decrypt

```
openssl rsautl -decrypt -inkey private.pem -in rsa_cipher.enc -raw -hexdump
0000 - 00 02 30 c9 36 9f 8a 4d-65 74 24 ea 5d 8f 34 31 ..0.6..Met$.].41
0010 - 46 e1 fd 5a c6 6b e1 87-c3 e3 5f 31 3b 33 34 6b F..Z.k...._1;34k
0020 - ef b4 b7 6d 3a be c4 a9-b6 8a 28 96 23 bb b5 e6 ...m:.....(.#...
0030 - af 5a a3 6d f8 93 15 43-1e be 69 5f b3 b6 c1 55 .Z.m...C..i...U
0040 - aa d8 3a e5 dd 2b 8f 0a-1c 3a ab 87 4a 33 33 c4 ...:..+....._J33.
0050 - a1 08 50 1e fc e8 9f de-f9 b3 04 90 bb 82 16 6a ..P.....j
0060 - 19 92 f3 fc a8 f2 6e 56-41 2c 4a 4d de 02 df 71 .....nVA,JM...q
0070 - d1 5b e9 61 74 13 d1 50-b2 4d 0c 59 66 87 d0 10 [.at..P.M.Yf...
0080 - 4e db 65 f3 e0 9b 13 69-8c 6a 8f 13 d2 9b ef 49 N.e....i.j....I
0090 - bc a3 9c a0 1f a7 31 a1-11 c0 0d 22 ea 28 f4 e7 .....1....".(..
00a0 - 18 e4 5c 49 f0 35 ad 8f-34 e6 76 1a a3 c1 b0 18 ..\I.5..4.v.....
00b0 - b6 c9 3c 8e df 4c 5d 7d-3d de 26 f7 8f 20 a8 de ..<..L]}=.&...
00c0 - 14 32 95 92 eb 03 7b 44-d9 e9 61 2c eb 56 6d b2 .2.....{D..a,.Vm.
00d0 - 2f d6 1e e5 53 c3 e1 0c-26 a5 f7 92 d1 1c f0 3b /...S...&.....;
00e0 - 62 04 e3 49 2f 04 9b f8-96 00 48 65 6c 6c 6f 20 b..I/.....Hello
00f0 - 57 6f 72 6c 64 0a 42 79-65 20 57 6f 72 6c 64 0a World.Bye World.
```

- ECC is a newer, faster, safer algorithm comparing with RSA
- The usage is a little different. It requires both sender and receiver to generate a pair of public key and private key.
- Alice uses her private key + Bob's public key to generate a secret.
- Bob uses his private key + Alice's public key to generate a secret.
- The secret generated by Alice and Bob are the same.
- This process is called **Diffie-Hellman Key Exchange**.





Act as the receiver, you should have done the following steps:

## 1. Generate ECC Private key

```
openssl ecparam -name secp256k1 -genkey -out ec_private.pem
```

The option `-name secp256k1` selects a particular elliptic curve (with 256 bits). You can use `openssl ecparam -list_curves` to view all supported curves.

## 2. Generate ECC Public key from Private key

```
openssl ec -in ec_private.pem -pubout -out ec_public.pem
```

Now you should have two files `ec_private.pem` and `ec_public.pem`. Make sure you keep the file `ec_private.pem` private and send the file `ec_public.pem` to others.

Act as the sender, you should have the file `ec_public.pem`. Repeat both steps above to generate the sender public key pairs. You should do this step for each time you encrypt a new message.

```
openssl ecparam -name secp256k1 -genkey -out s_ec_private.pem  
openssl ec -in s_ec_private.pem -pubout -out s_ec_public.pem
```



Then, derive a shared secret by the command (type on the same line):

```
openssl pkeyutl -derive -inkey s_ec_private.pem  
-peerkey ec_public.pem | hexdump -e '32 "%x""\n''
```

- The first half of the command (before `|`) is to derive a 256-bits share secret in binary, from the sender private key `s_ec_private.pem` and the receiver public key `ec_public.pem`.
- The second half of the command (after `|`) is to convert the binary data into the hex format.

You can then use the derived secret as a AES key to encrypt content. Send the encrypted content together with the file `s_ec_public.pem` to the receiver.

Diffie-Hellman Key Exchange: <https://sandilands.info/sgordon/doc/diffie-hellman-secret-key-exchange-with-openssl>

  DH allow you to agree an session key with two given public keys. Content needs to be encrypted by the session key. It is much preferable to generate a temporary public key for the **sender** when using DH key exchange. Send the cipher together with the temporary public key to the receiver.

The idea is very similar except we are not using ECC version.

In assymetric key encryption...

1. Assymetric key encryption is also called.... public key encryption
2. Who has the public key? every one
3. Who has the private key?
4. If Bob wants to send a message to Alice, he should use.....
5. To decrypt the message sent from Bob, Alice should use....
6. Which file(s) from the hands-on work represent public key/private key?

- Symmetric Key - DES, AES
- Stream Cipher vs Block Cipher
- Mode of Operations for symmetric key
- Asymmetric Key
- Advantage of ECC
- Hands-on with openssl