# Symmetric Key Crypto

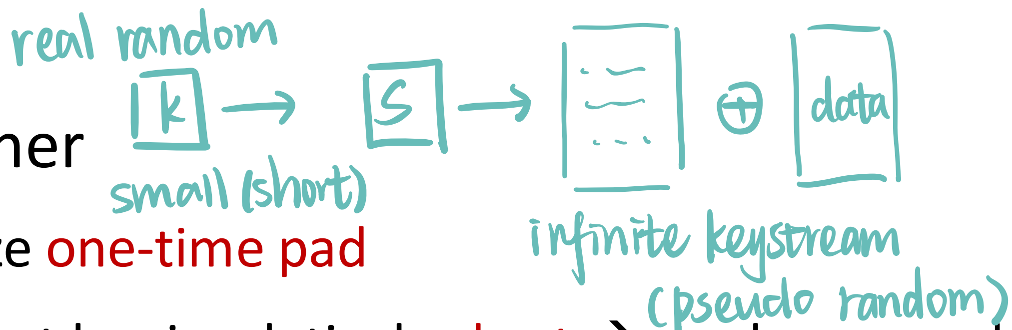
Shuai Wang

香 港 科 技 大 學
THE HONG KONG UNIVERSITY OF SCIENCE AND TECHNOLOGY

Some slides are written by Mark Stamp.

# (Modern) Symmetric Key Crypto

- Stream cipher




  - generalize one-time pad
  - Except that key is relatively short → works on one bit/byte at a time

    A5/1 RC4

  - Key is stretched into a long keystream
  - Keystream is used just like a one-time pad
- Block cipher
  - More popular than stream cipher
  - Works on larger chunks of data (blocks) at a time
  - Can even combine blocks for additional security

# Stream Ciphers

Generates keystream of any length
from **random seed**

- Keystream is pseudorandom
- ~~Key is truly uniformly random~~
- Key is only used once, ever

$$Enc_{seed}(M) = K_{seed} \oplus M$$

# Stream Ciphers

- Stream ciphers were the king of crypto
- Today, not as popular as block ciphers
- We'll discuss two stream ciphers:
- A5/1
  - Based on shift registers → hardware
  - Used in GSM mobile phone system
- RC4 → Rivest Cipher 4
  - Based on a changing lookup table → check book
  - Used many places
  - *Note: RC5 is a block cipher*
  - *RSA*
    - *Shamir*

Rivest Shamir Algorithm

# A5/1: Shift Registers 线性反馈位移寄存器

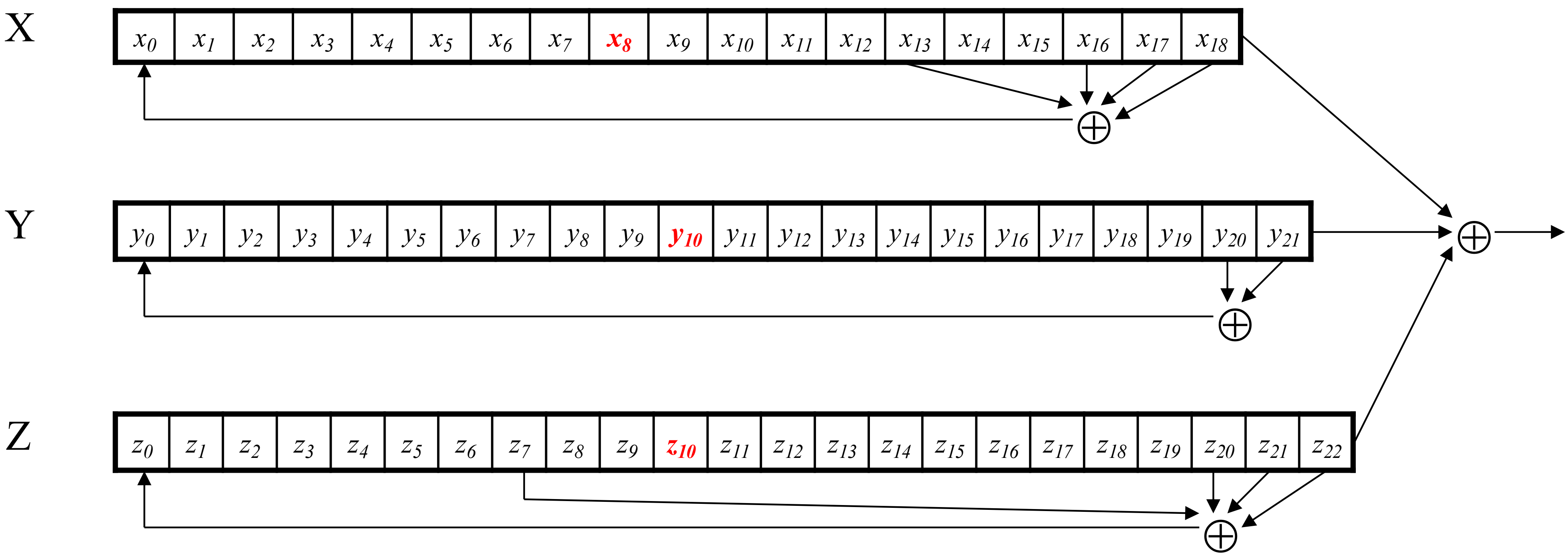
LFSR: linear-feedback shift register

- A5/1 uses 3 *shift registers*
  - X: 19 bits $(x_0, x_1, x_2, \ldots, x_{18})$
  - Y: 22 bits $(y_0, y_1, y_2, \ldots, y_{21})$
  - Z: 23 bits $(z_0, z_1, z_2, \ldots, z_{22})$
- Use these bits to generate as many bits as we want to serve as One Time Pad.

# A5/1: Keystream

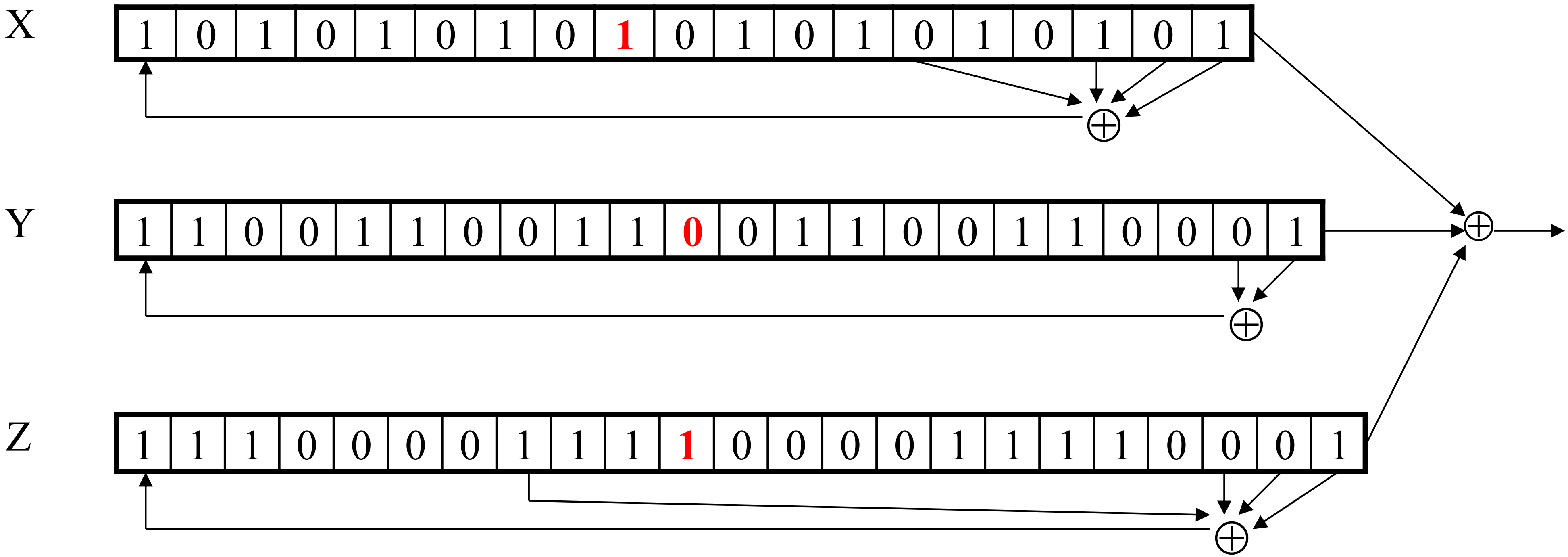
- At each iteration: $m = \text{maj}(x_8, y_{10}, z_{10})$
  - Examples: $\text{maj}(0,1,0) = 0$ and $\text{maj}(1,1,0) = 1$
- If $x_8 = m$ then $\text{X}$ *steps*
  - $t = x_{13} \oplus x_{16} \oplus x_{17} \oplus x_{18}$
  - $x_i = x_{i-1}$ for $i = 18,17,\ldots,1$ and $x_0 = t$
- If $y_{10} = m$ then $\text{Y}$ *steps*
  - $t = y_{20} \oplus y_{21}$
  - $y_i = y_{i-1}$ for $i = 21,20,\ldots,1$ and $y_0 = t$
- If $z_{10} = m$ then $\text{Z}$ *steps*
  - $t = z_7 \oplus z_{20} \oplus z_{21} \oplus z_{22}$
  - $z_i = z_{i-1}$ for $i = 22,21,\ldots,1$ and $z_0 = t$
- Keystream **bit** is $x_{18} \oplus y_{21} \oplus z_{22}$

# A5/1




- Each variable here is a single bit
- Key is used as **initial fill** of registers
- Each register steps (or not) based on $\text{maj}(x_8, y_{10}, z_{10})$
- Keystream bit is XOR of rightmost bits of registers

# A5/1




- In this example, $m = \text{maj}(x_8, y_{10}, z_{10}) = \text{maj}(\mathbf{1},\mathbf{0},\mathbf{1}) = \mathbf{1}$
- Register X steps, Y does not step, and Z steps
- Keystream bit is XOR of right bits of registers
- Here, keystream bit will be $0 \oplus 1 \oplus 0 = 1$

# Shift Register Crypto

- Shift register crypto efficient in hardware
- Often, slow if implemented in software
- In the past, very, very popular
- Today, more is done in software due to fast processors
- Shift register crypto still used some
  - E.g. in resource-constrained devices; embedded devices.

# RC4

- A self-modifying lookup table
- Table always contains a permutation 排列 of the byte values 0,1,…,255
- Initialize the permutation using key
- At each step, RC4 does the following
  - Swaps elements in current lookup table
  - Selects a keystream byte from table
- Each step of RC4 produces a **byte**
- Each step of A5/1 produces only a bit
  - Efficient in hardware

# RC4 Initialization

- `S[]` is permutation of 0,1,...,255
- `key[]` contains N bytes of key

```
for i = 0 to 255
        S[i] = i
        K[i] = key[i (mod N)] ← any key length!
next i
j = 0
for i = 0 to 255
        j = (j + S[i] + K[i]) mod 256
        swap(S[i], S[j])
next i
i = j = 0
```

# RC4 Keystream

- At each step, swap elements in table and select keystream byte

```
i = (i + 1) mod 256
j = (j + S[i]) mod 256
swap(S[i], S[j])
t = (S[i] + S[j]) mod 256
keystreamByte = S[t]
```

- Use keystream bytes like a one-time pad
- **Note:** first 256 bytes should be discarded
  - Otherwise, attack exists

# Stream Ciphers

- Stream ciphers were popular in the past
  - Efficient in hardware → speed
  - Speed was needed to keep up with voice, etc.
    - Less memory consumption as well.
  - Today, processors are fast, so software-based crypto is usually more than fast enough
- Future of stream ciphers?
  - “the death of stream ciphers”?

# Block Ciphers

$* \; C = S[\underline{K} \oplus P] \oplus \underline{K}$ ← $k_1$, $k_2$

$k_1 \neq k_2$ but related.

S-box: $S[P \oplus K] = C$ (P: plaintex, C: cyphertext)

① non-secret

② non-linear function

$P \oplus C = P \oplus S[P \oplus K]$

yet: $S \to S^{-1}$ easy to get

$\Downarrow$

$S^{-1}[C] = S^{-1} \circ S[P \oplus K] = P \oplus \underline{\underline{K}}$ easy to get (not secure)

$P \oplus C = P \oplus (S[K \oplus P] \oplus K)$

即使attacker有$S^{-1}$也难解出K

$\Downarrow$

more rounds of peradim

more secure, but slow.

# Block Cipher

$$\text{eg. } k_4 \oplus S[k_3 \oplus \overbrace{S[k_2 \oplus}^{\text{round}} S[k_1 \oplus S[k_0 \oplus S]]]]$$

- Plaintext and ciphertext consist of fixed-sized blocks → bits

每一组叠加加密为一round

- Ciphertext obtained from plaintext by iterating a **round function**
- Input to round function consists of ***key*** and ***output*** of previous round
- Usually implemented in software
  - Used to be slow, but it's fine for modern CPUs
  - Intel has specific hardware instructions to speedup AES
    - AES-NI
    - AESENC AESDEC

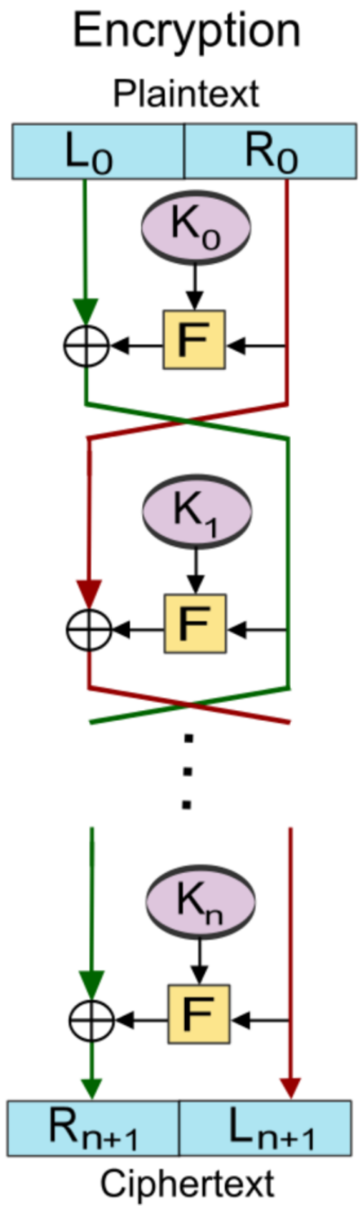
# Feistel Cipher: Encryption

- **Feistel cipher** is a “type” of block cipher
  - ***Not*** a specific block cipher but a framework
  - Instances: DES; blowfish; RC5; TEA; Twofish…
- Split plaintext block into left and right components: $P = (L_0, R_0)$
- For each round $i = 0, 1, \ldots, n$, compute

  $L_{i+1} = R_i$

  $R_{i+1} = L_i \oplus F(R_i, K_i)$

  where $F$ is round function and $K_i$ is subkey
- Ciphertext: $C = (R_{n+1}, L_{n+1})$



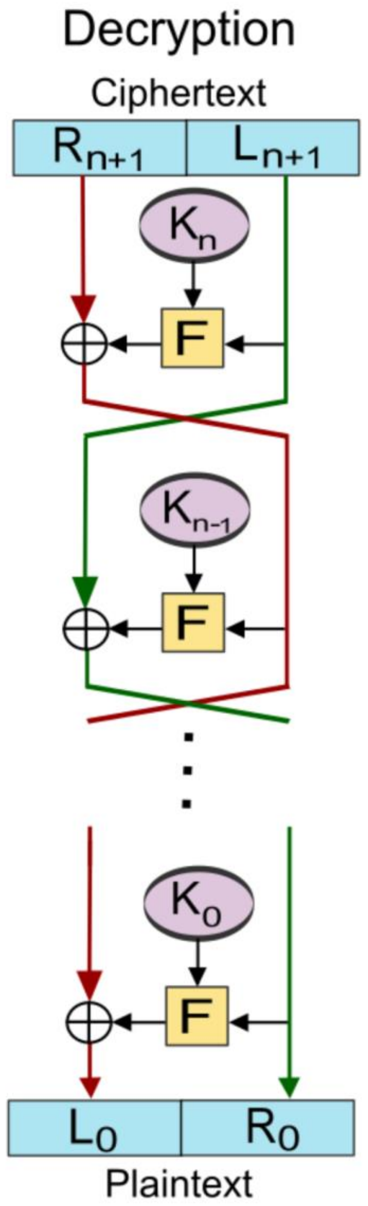
# Feistel Cipher: Decryption

- Start with ciphertext $C = (R_{n+1}, L_{n+1})$
- For each round $i = n+1, n, \ldots, 1$, compute

  $R_{i-1} = L_i$

  $L_{i-1} = R_i \oplus F(L_i, K_{i-1})$

  where $F$ is round function and $K_i$ is subkey
- Plaintext: $P = (L_0, R_0)$
- Decryption works for any function $F$
  - But only secure for certain functions $F$
  - What about $F = 0$? X



# Data Encryption Standard

→16 times (rounds)

$k = 56$ bits

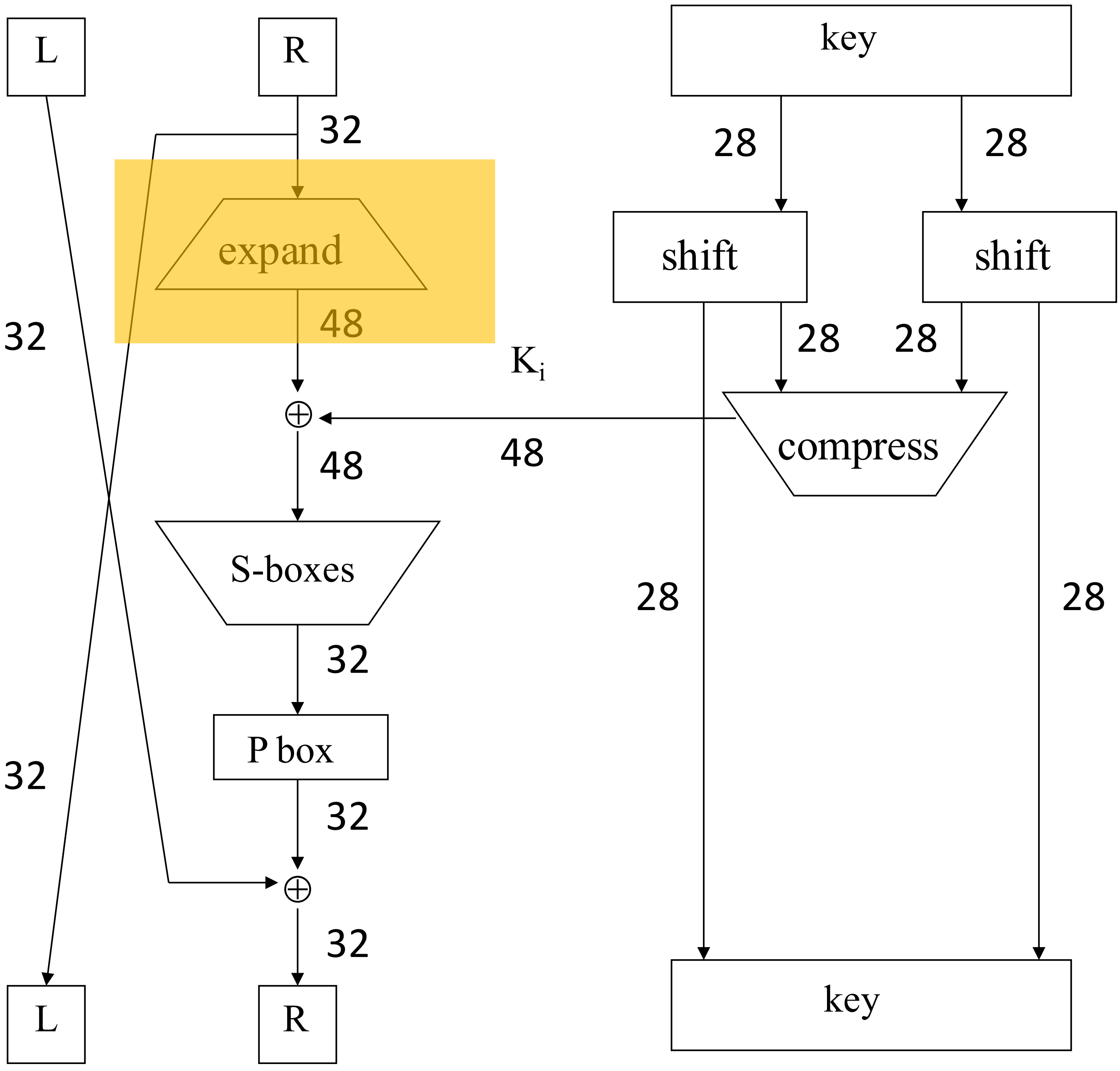
- **DES** developed in 1970's

$k_1 \dots k_{16} = 48$ bits

- An implementation of Feistel Cipher
- DES was U.S. government standard
  - NSA "secured" the algorithm.
  - But it has been cracked in 1998
- Many successors
  - Triple DES; AES; G-DES; …

# DES

- DES is a Feistel cipher with…
  - 64 bit block length
  - 56 bit key length
  - 16 rounds
  - 48 bits of key used each round (subkey)
- Round function is simple (for block cipher)
- Security depends heavily on "S-boxes"
  - Each S-box maps 6 bits to 4 bits




One Round of DES

# DES Expansion Permutation

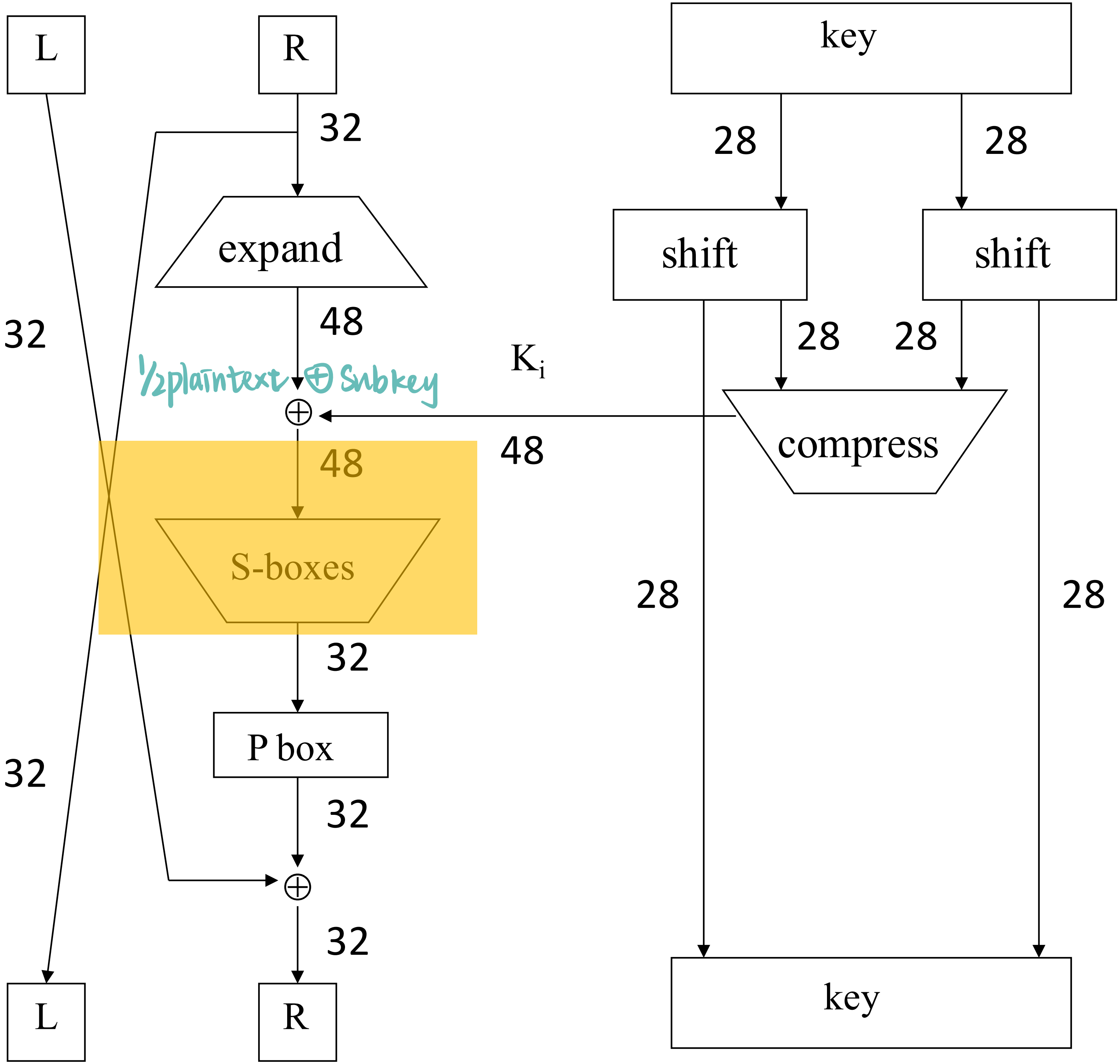
- Input 32 bits

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

- Output 48 bits

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

- Output contains eight 6-bit (8 * 6 = 48 bits) pieces




# One Round of DES

# DES S-box

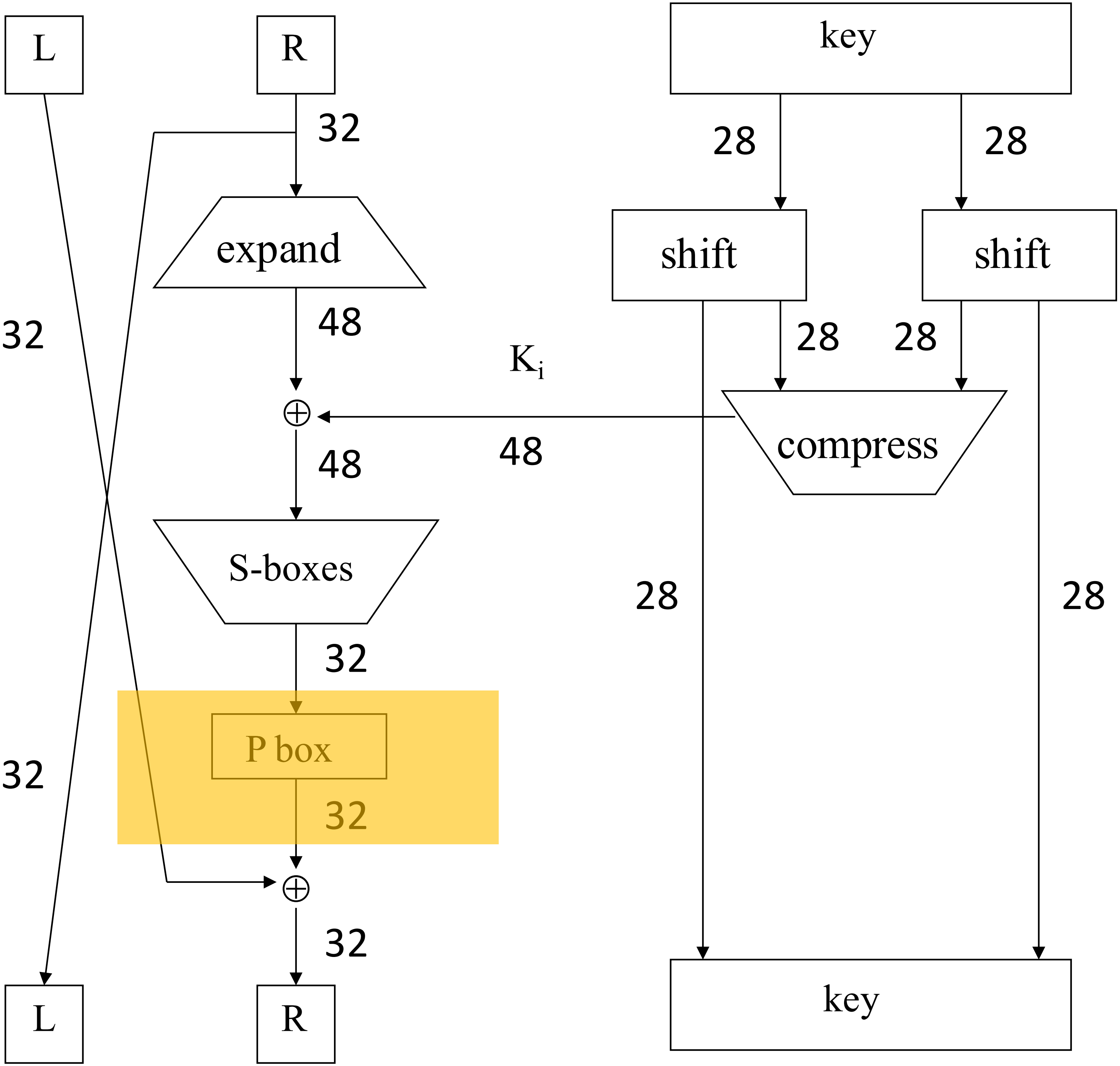
- 8 “substitution boxes” or S-boxes
  - In DES S-boxes are carefully chosen to resist cryptanalysis.
  - Thus, that is where the security comes from.
- Each S-box maps 6 bits to 4 bits

input bits (0,1,2,3,4,5)

↓ Middle four bits of the input bits

| | 0000 | 0001 | 0010 | 0011 | 0100 | 0101 | 0110 | 0111 | 1000 | 1001 | 1010 | 1011 | 1100 | 1101 | 1110 | 1111 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 00 | 1110 | 0100 | 1101 | 0001 | 0010 | 1111 | 1011 | 1000 | 0011 | 1010 | 0110 | 1100 | 0101 | 1001 | 0000 | 0111 |
| 01 | 0000 | 1111 | 0111 | 0100 | 1110 | 0010 | 1101 | 0001 | 1010 | 0110 | 1100 | 1011 | 1001 | 0101 | 0011 | 1000 |
| 10 | 0100 | 0001 | 1110 | 1000 | 1101 | 0110 | 0010 | 1011 | 1111 | 1100 | 1001 | 0111 | 0011 | 1010 | 0101 | 0000 |
| 11 | 1111 | 1100 | 1000 | 0010 | 0100 | 1001 | 0001 | 0111 | 0101 | 1011 | 0011 | 1110 | 1010 | 0000 | 0110 | 1101 |

non-linear transformation, provided in the form of a lookup table




# One Round of DES

# DES P-box

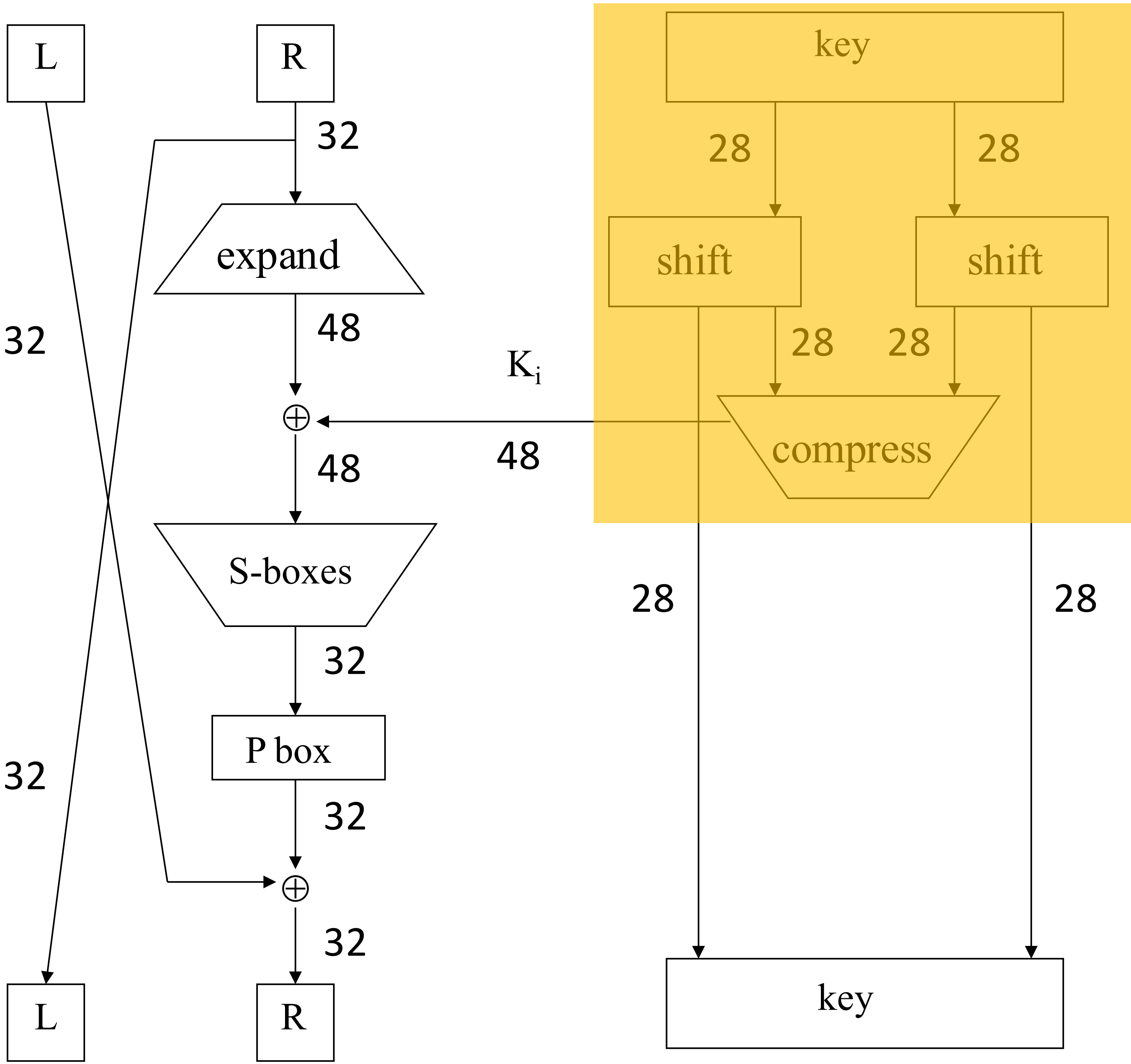
- Basically we further permutate the output of S-box.

- Input 32 bits

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

- Output 32 bits

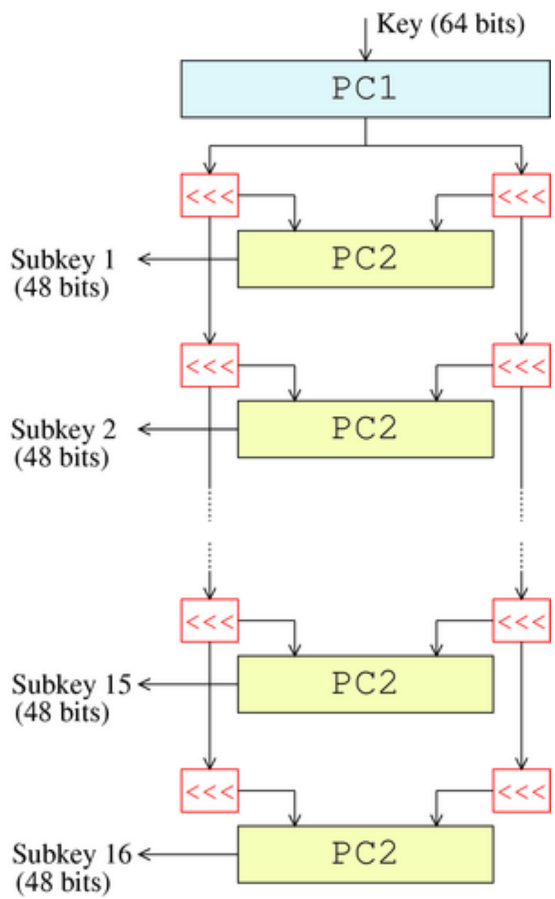
```
15  6 19 20 28 11 27 16  0 14 22 25  4 17 30  9
 1  7 23 13 31 26  2  8 18 12 29  5 21 10  3 24
```




# One Round of DES

# DES Subkey

- 56 bit DES key out of 64-bit, numbered 0,1,2,…,55
  - 24 * 2 bits of key are eventually extracted from 28 * 2 bits.

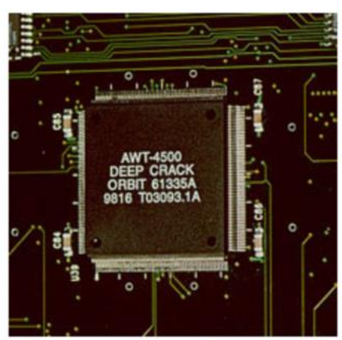


# DES: Prologue and Epilogue

- An initial permutation before round 1
- Halves are swapped after last round
- A final permutation (inverse of initial perm) applied to $(R_{16}, L_{16})$
- None of this serves any security purpose
  - But this is how the algorithm is designed…

# Security of DES

- Security depends heavily on S-boxes
- Attacks by exhaustive key search
  - given message $x$ and a ciphertext $c$ such that $DES_k(x)$=$c$ → find k
  - Wiener: $1,000,000 - 3.5 hours; never built in public
  - 1998, the EFF DES Cracker, which was built for less than $250,000 < 3 days
  - 1999, Distributed.Net (*idle time of your computer*), 22 hours and 15 minutes (over many machines)
  - You can assume that NSA and agencies likely can crack DES in milliseconds



“Deep crack” DES cracker

2-DES: $C = E(K_1, E(K_2, P))$ "meet-in-the-middle attack"

↳ security: $2^{56} + 2^{56} = 2^{57}$ probabilities.

# Triple DES

- Today, 56 bit DES key is too small
  - As aforementioned, exhaustive key search is feasible
- **Triple DES** or **3DES** (with 112 bit key, why?)
  - $C = E(D(E(P,K_1),K_2),K_1)$
  - $P = D(E(D(C,K_1),K_2),K_1)$
- Why Encrypt-Decrypt-Encrypt with 2 keys?
  - Why not encrypt-encrypt-encrypt mode?
    - Backward compatible: $E(D(E(P,K),K),K) = E(P,K)$
  - And 112 is a lot of bits

# 3DES

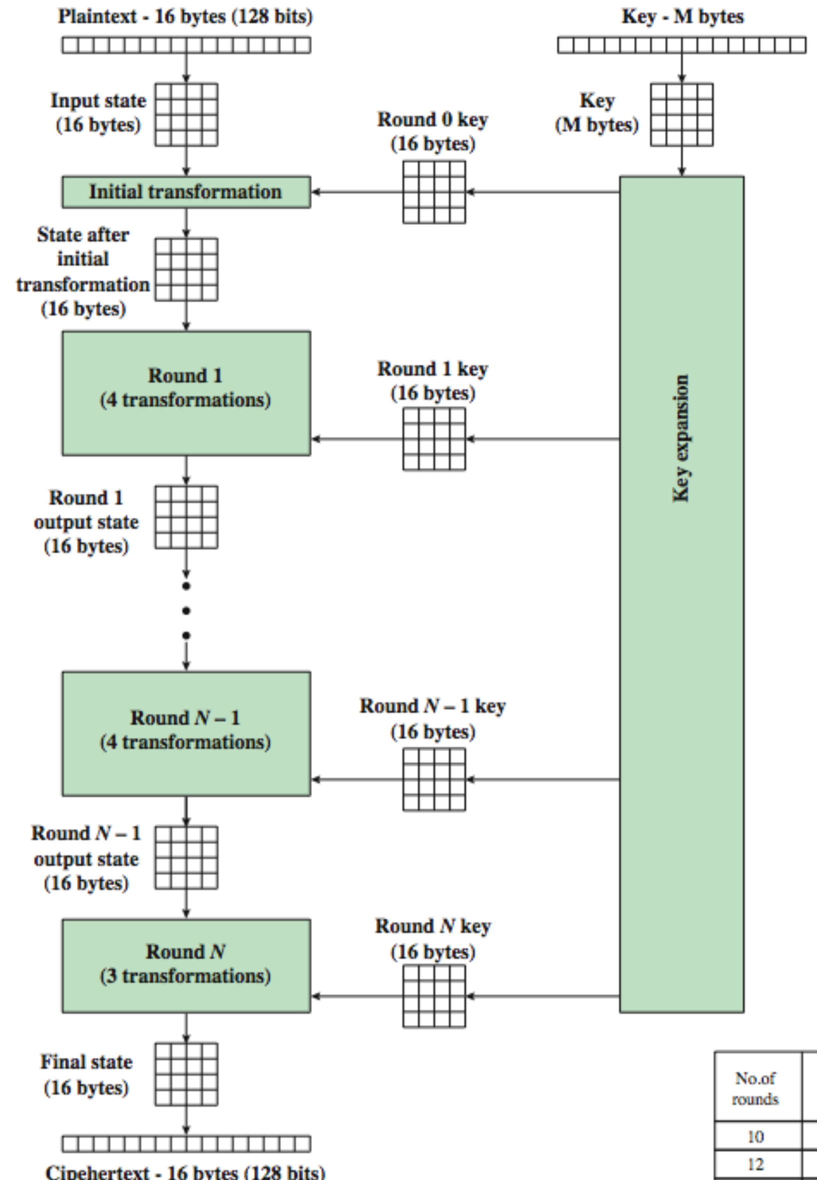
- Why not $C = E(E(P,K),K)$ instead?
    - still just 56 bit key
- But why not $C = E(E(P,K_1),K_2)$ instead?
    - Unfortunately, brute force search difficulty can be reduced from 2^112 to 2^57
- Meet-in-the-middle attack

# Advanced Encryption Standard

- Replacement for DES
  - The de-facto symmetric cipher since late 1990
- Not a Feistel cipher (unlike DES)
  - Variable key lengths
  - Fast implementation in hardware and software
  - Small code and memory footprint → but still too much
- We will provide a (simplified) implementation of AES after today's class.
  - Just for your reference, you are NOT required to remember its implementation.

# AES: Executive Summary

- **Block size:** 128 bits
- **Key length:** 128, 192 or 256 bits
- 10 to 14 rounds (depends on key length)
- Each round uses 4 functions
  - ByteSub (S-box layer)
  - ShiftRow
  - MixColumn
  - AddRoundKey (key addition layer)

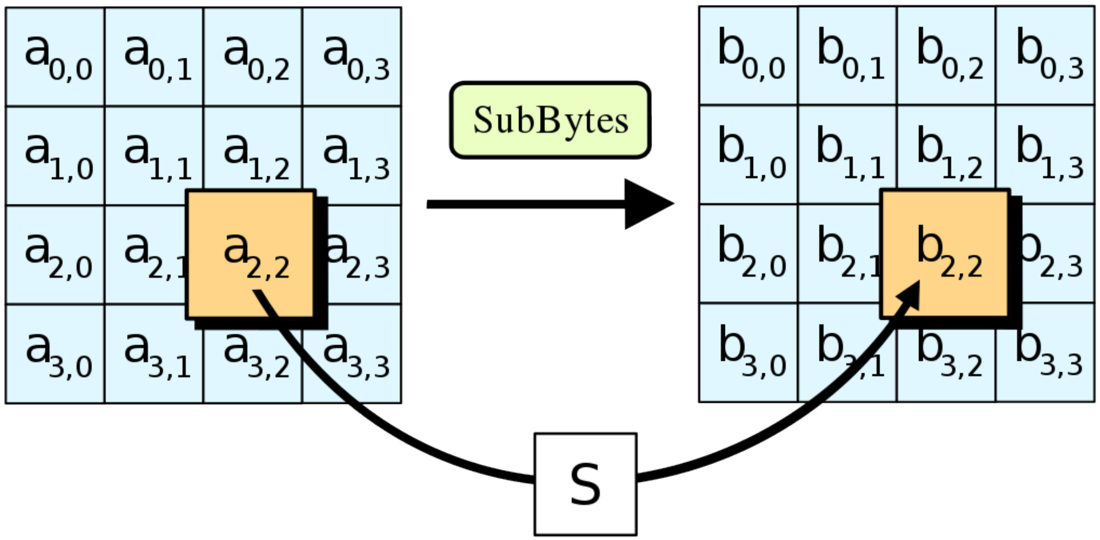



| No.of rounds | Key Length (bytes) |
|---|---|
| 10 | 16 |
| 12 | 24 |
| 14 | 32 |

You don't need to remember this figure!!!!!

# AES ByteSub

- Treat 128 bit block as 4x4 byte array




ByteSub is AES's "S-box"

- No fixed point $a_{i,j} = S(a_{i,j})$

# AES “S-box”

Last 4 bits of input

First 4 bits of input

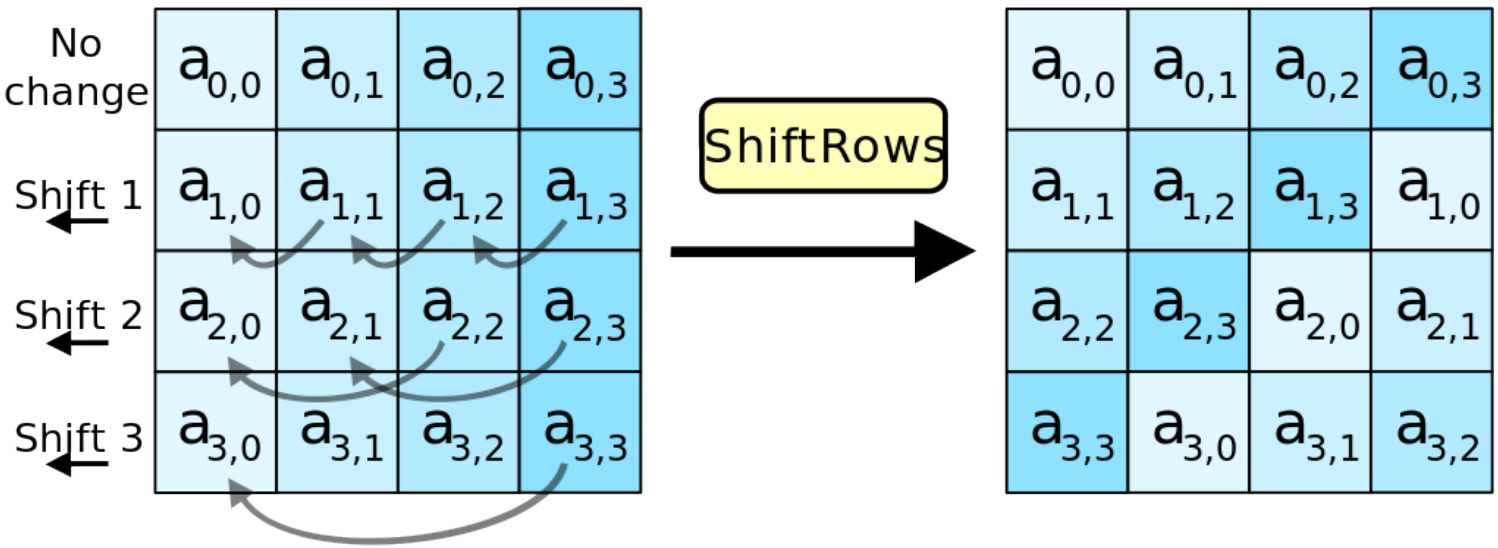
| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | a | b | c | d | e | f |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 63 | 7c | 77 | 7b | f2 | 6b | 6f | c5 | 30 | 01 | 67 | 2b | fe | d7 | ab | 76 |
| 1 | ca | 82 | c9 | 7d | fa | 59 | 47 | f0 | ad | d4 | a2 | af | 9c | a4 | 72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

Then, how to invert this table? (need another table)

- 0x62 → 0xaa ➔ 0xaa → 0x62

# AES ShiftRow

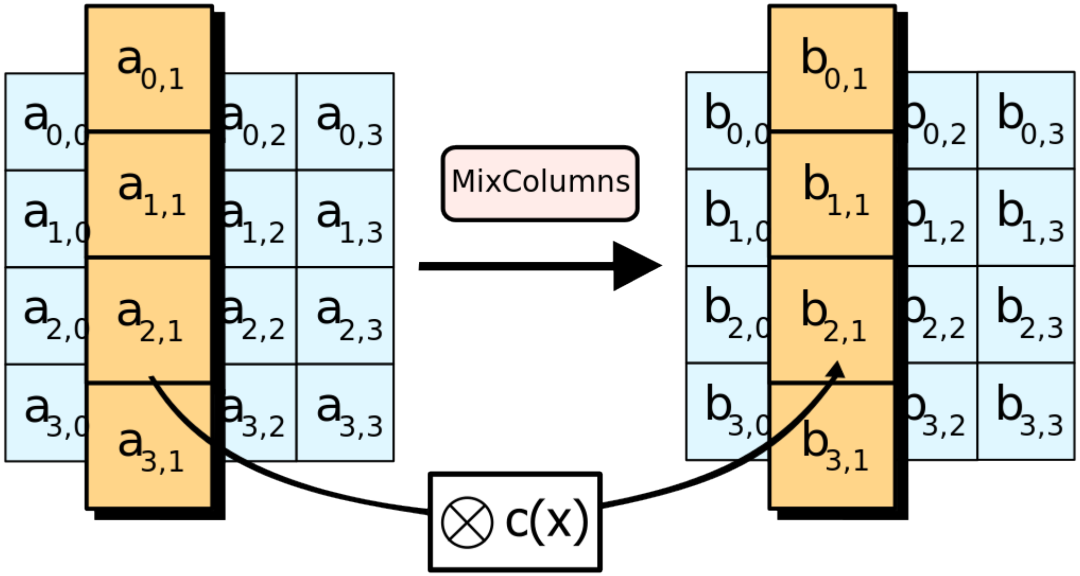
- Cyclic shift rows




To avoid the columns being encrypted independently

# AES MixColumn

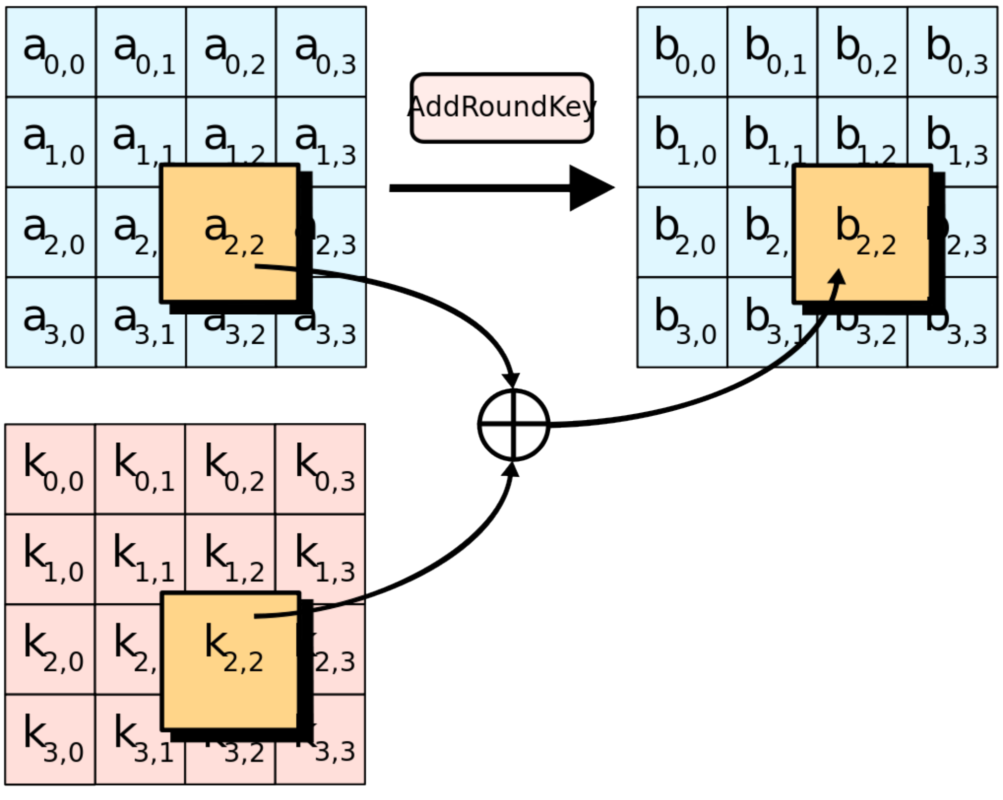
- Invertible operation applied to each column




- It's a matrix multiplication.
  - Implemented as a (big) lookup table

# AES AddRoundKey

- XOR subkey with block



# AES Comments

- Can be effectively implemented in 8-bit and 32-bit CPU.
  - One key reason it is becoming so popular
    - The franchise player in block cipher
  - Specific hardware instructions.
    - AES-NI
    - AESENC AESDEC
- Some other good properties
  - Avalanche effect: one bit difference in plaintext causes a big change in cipher text.
    - Talk about that later..

# Tiny Encryption Algorithm

- 64 bit block, 128 bit key
- Assumes 32-bit arithmetic
- Number of rounds is variable (32 is considered secure)
- Uses “weak” round function, so large number of rounds required

# TEA Encryption

Assuming 32 rounds:

$(K[0], K[1], K[2], K[3]) = 128$ bit key

$(L,R) =$ plaintext (64-bit block)

$\text{delta} = \text{0x9e3779b9}$ Magic number!

$\text{sum} = 0$

for $i = 1$ to $32$

$\quad \text{sum} \mathrel{+}= \text{delta}$

$\quad L \mathrel{+}= ((R \ll 4)+K[0]) \oplus (R+\text{sum}) \oplus ((R \gg 5)+K[1])$

$\quad R \mathrel{+}= ((L \ll 4)+K[2]) \oplus (L+\text{sum}) \oplus ((L \gg 5)+K[3])$

ciphertext $= (L,R)$

Again, you don't need to remember this detailed algorithm

# TEA Decryption

Assuming 32 rounds:

$(K[0], K[1], K[2], K[3]) = 128$ bit key

$(L,R) =$ ciphertext (64-bit block)

delta = 0x9e3779b9

sum = delta << 5

for i = 1 to 32

$\quad R \mathrel{-}= ((L \ll 4)+K[2]) \oplus (L+\text{sum}) \oplus ((L \gg 5)+K[3])$

$\quad L \mathrel{-}= ((R \ll 4)+K[0]) \oplus (R+\text{sum}) \oplus ((R \gg 5)+K[1])$

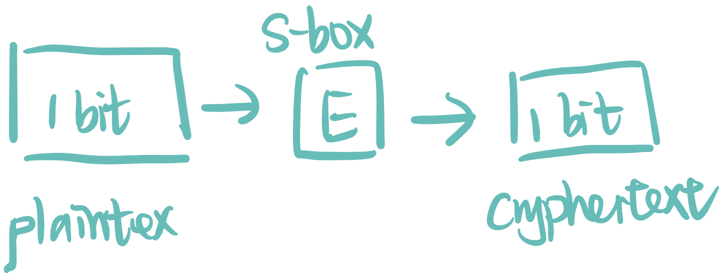
$\quad \text{sum} \mathrel{-}= \text{delta}$

plaintext = (L,R)

Again, you don't need to remember this detailed algorithm

# TEA Comments

- **"Almost"** a Feistel cipher?
  - Uses + and - instead of ⊕ (XOR)
  - Each step both left and right halves are used.
- Simple, easy to implement, fast, low memory requirement, etc.
  - Can you memorize the implementation like how you memorize quick sort algorithm?
    - Well, that's why it's called "tiny".
- Possibly enable attacks?
  - four different keys that all give the exact same encrypted output
  - 2^128 → 2^126
  - X-box attacks.




# Overview of Linear Cryptanalysis

| C \ P | 0 | 1 |
|---|---|---|
| 0 | 50% | 50% |
| 1 | 50% | 50% |

$\Rightarrow$ independent

$C \nRightarrow P$ ✓

| C \ P | 0 | 1 |
|---|---|---|
| 0 | 90% | 10% |
| 1 | 10% | 90% |

$\Rightarrow \begin{cases} C = P & 90\% \\ C \neq P & 10\% \end{cases}$

# Linear Cryptanalysis

- Non-linear functions (i.e., s-box) are the primary contribution of security of block ciphers.
- To attack, we approximate the nonlinearity with **linear equations**
- How well can we do this?

# Vulnerable S-box Linear Analysis

- Input $x_0x_1x_2$ where $x_0$ is row and $x_1x_2$ is column
- Output $y_0y_1$

| | column | | | |
|---|---|---|---|---|
| row | 00 | 01 | 10 | 11 |
| 0 | 10 | 01 | 11 | 00 |
| 1 | 00 | 10 | 01 | 11 |

# Linear Analysis

- For example,
  $y_1 = x_1$
  with prob. 3/4
- And
  $y_0 = x_0 \oplus x_2 \oplus 1$
  with prob. 1
- And
  $y_0 \oplus y_1 = x_1 \oplus x_2$
  with prob. 3/4

| | column | | | |
|---|---|---|---|---|
| row | 00 | 01 | 10 | 11 |
| 0 | 10 | 01 | 11 | 00 |
| 1 | 00 | 10 | 01 | 11 |

# Linear Cryptanalysis of DES

- DES is linear except for S-boxes
- How well can we approximate S-boxes with linear functions?
- DES S-boxes designed so there are no good linear approximations to any one output bit
- But there **are** linear combinations of output bits that can be approximated by linear combinations of input bits
  - Still not very practical, requires a huge amount of plaintext

# Block Cipher Modes

# Multiple Blocks

- How to encrypt multiple blocks?
  - What’s the block size of AES/DES/TEA?
  - How large it is?
- Do we need a new key for each block?
  - If so, as impractical as a one-time pad!
- So, what can we do?
  - Encrypt each block individually?
  - Make encryption depends on previous block?
    - Then “chain” them together?
    - Partial block? Not considered in this course

# Modes of Operation

- Many modes — we discuss 3 most popular
- Electronic Codebook (**ECB**) mode
  - Encrypt each block independently
  - Most obvious approach, but a bad idea
- Cipher Block Chaining (**CBC**) mode
  - Chain the blocks together
  - More secure than ECB, a little bit extra work
- Counter (**CTR**) mode
  - Block ciphers acts like a stream cipher
  - Popular for random access

# ECB Mode

- Notation: $C = E(P, K)$
- Given plaintext $P_0, P_1, \ldots, P_m, \ldots$
- Most obvious way to use a block cipher:

| **Encrypt** | **Decrypt** |
|---|---|
| $C_0 = E(P_0, K)$ | $P_0 = D(C_0, K)$ |
| $C_1 = E(P_1, K)$ | $P_1 = D(C_1, K)$ |
| $C_2 = E(P_2, K) \quad \ldots$ | $P_2 = D(C_2, K) \quad \ldots$ |

- For fixed key $K$, this is "electronic" version of a codebook cipher (without additive)
  - With a different codebook for each key

# ECB Cut and Paste

- Suppose plaintext is

  `Eva. pings Bob. Ted. pings Tom.`
- Assuming 64-bit blocks and 8-bit ASCII:

  $P_0 =$ "`Eva. pin`", $P_1 =$ "`gs Bob. `",

  $P_2 =$ "`Ted. pin`", $P_3 =$ "`gs Tom. `"
- Ciphertext: $C_0, C_1, C_2, C_3$
- Attacker cuts and pastes: $C_0, C_3, C_2, C_1$
- Decrypts as

  `Eva. pings Tom. Ted. pings Bob.`

# Principles of CIA


Confidentiality:

- No one is supposed to read what data or information is sent unless he is authorized.

Integrity:

- The ability to ensure that data is an accurate and unchanged representation of the original secure information.

Availability:

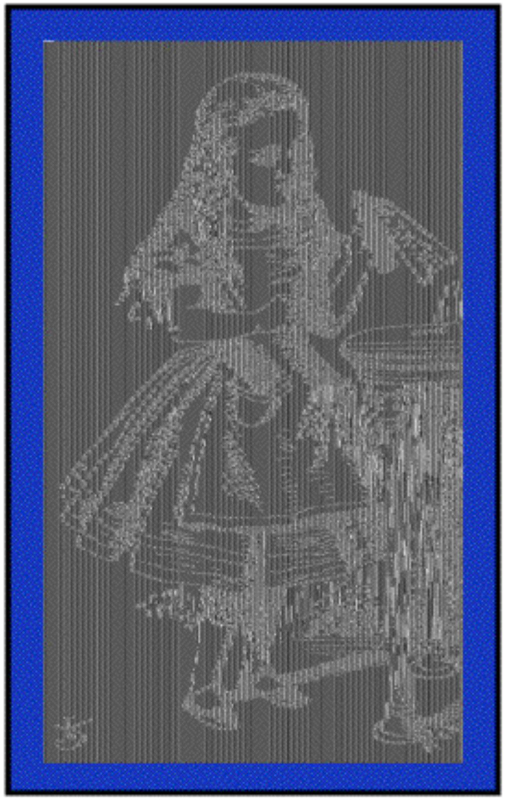
- the information concerned is readily accessible to the authorised viewer at all times

# ECB Weakness

- Suppose $P_i = P_j$
- Then $C_i = C_j$ and attacker knows $P_i = P_j$
- This gives attacker some information, even if he does not know $P_i$ or $P_j$
- Attacker might know $P_i$
- Is this a serious issue?
  - Any information can be leaked?

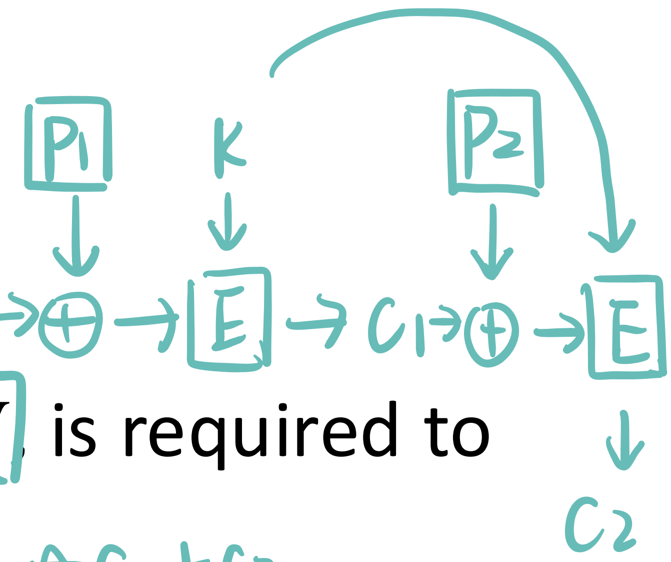
# Alice Hates ECB Mode

- Alice's uncompressed image, and ECB encrypted (TEA)



- ❑ Why does this happen?
- ❑ Same plaintext yields same ciphertext!

# CBC Mode




- Blocks are "chained" together
- A random initialization vector, or $\boxed{\text{IV}}$, is required to initialize CBC mode
- IV is random, but not secret

☆ $C_2 \neq C_3$

minimize repeated pattern!

**Encryption**

$$C_0 = E(IV \oplus P_0, K),$$
$$C_1 = E(C_0 \oplus P_1, K),$$
$$C_2 = E(C_1 \oplus P_2, K), \ldots$$

**Decryption**

$$P_0 = IV \oplus D(C_0, K),$$
$$P_1 = C_0 \oplus D(C_1, K),$$
$$P_2 = C_1 \oplus D(C_2, K), \ldots$$

- Analogous to classic codebook *with additive*

$C_n = E(C_{n-1} \oplus P_n, k)$

$P_n = C_{n-1} \oplus D(C_n, k)$

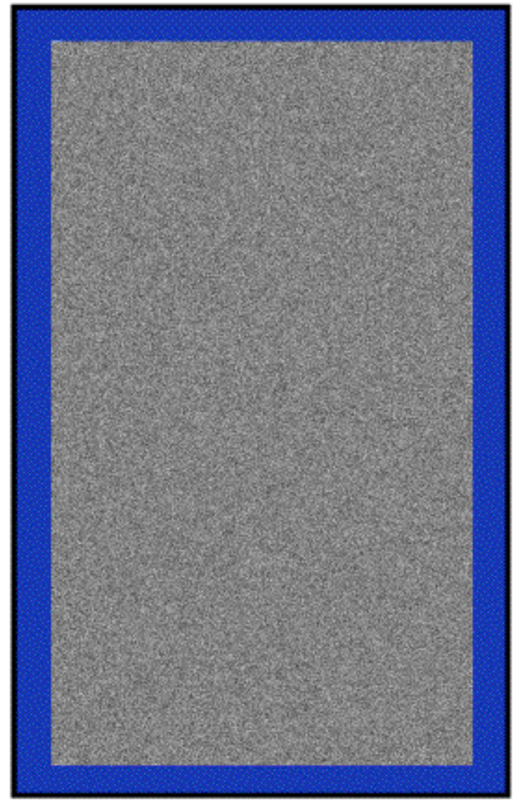
# CBC Mode

- Identical plaintext blocks yield different ciphertext blocks — this is very good!
- But what about errors in transmission?
  - If $C_1$ is garbled to, say, $G$ then

    $$P_1 \neq C_0 \oplus D(G, K),\ P_2 \neq G \oplus D(C_2, K)$$
  - But $P_3 = C_2 \oplus D(C_3, K),\ P_4 = C_3 \oplus D(C_4, K),\ \ldots$
  - "G" can only influence limited decryptions!
- Cut and paste is still possible
  - Talk more on "integrity" later today.

# Alice Likes CBC Mode

- Alice's uncompressed image, Alice CBC encrypted (TEA)



- ❑ Why does this happen?
- ❑ Same plaintext yields different ciphertext!

# Counter Mode (CTR)

- CTR is popular for random access
- Use block cipher like a stream cipher

**Encryption**

$C_0 = P_0 \oplus E(IV, K),$

$C_1 = P_1 \oplus E(IV+1, K),$

$C_2 = P_2 \oplus E(IV+2, K), \ldots$

**Decryption**

$P_0 = C_0 \oplus E(IV, K),$

$P_1 = C_1 \oplus E(IV+1, K),$

$P_2 = C_2 \oplus E(IV+2, K), \ldots$

E&D is independent

$\Downarrow$

平行加解密

# Integrity

[ 完整性 ]

# Data Integrity

- **Integrity** — detect unauthorized writing (i.e., detect unauthorized mod of data)
- Example: Inter-bank fund transfers
  - Confidentiality may be nice, integrity is ***critical***
- Encryption provides **confidentiality** (prevents unauthorized disclosure)
- Encryption alone does **not** provide integrity
  - One-time pad, ECB cut-and-paste, etc., etc.

# MAC

- Message Authentication Code (MAC)
  - Used for data **integrity**
  - Integrity not the same as confidentiality
- MAC is computed as **CBC residue**
  - That is, compute CBC encryption, saving only final ciphertext block, the MAC
  - The MAC serves as a cryptographic checksum for data
    - Any tempering of data will be detected.

# MAC Computation

- MAC computation (assuming N blocks)

$C_0 = E(IV \oplus P_0, K),$

$C_1 = E(C_0 \oplus P_1, K),$

$C_2 = E(C_1 \oplus P_2, K), \ldots$

$C_{N-1} = E(C_{N-2} \oplus P_{N-1}, K) = \text{MAC}$

- Send IV, $P_0$, $P_1$, …, $P_{N-1}$ and MAC
- Receiver does same computation and verifies that result agrees with MAC
- Both sender and receiver must know K

# Does a MAC work?

- Suppose Alice has 4 plaintext blocks
- Alice computes

  $C_0 = E(IV \oplus P_0, K)$, $C_1 = E(C_0 \oplus P_1, K)$,

  $C_2 = E(C_1 \oplus P_2, K)$, $C_3 = E(C_2 \oplus P_3, K) = \mathbf{MAC}$
- Alice sends IV, $P_0$, $P_1$, $P_2$, $P_3$ and **MAC** to Bob
- Suppose the attacker changes $P_1$ to X
- Bob computes

  $C_0 = E(IV \oplus P_0, K)$, $\boldsymbol{C_1} = E(C_0 \oplus X, K)$,

  $\boldsymbol{C_2} = E(\boldsymbol{C_1} \oplus P_2, K)$, $\boldsymbol{C_3} = E(\boldsymbol{C_2} \oplus P_3, K) = \boldsymbol{MAC} \neq \mathbf{MAC}$
- It works since error propagates into **MAC**
- The attacker can't make ***MAC*** == **MAC** without K

# Confidentiality and Integrity

- Encrypt with one key, MAC with another key
- Why not use the same key?
  - Send last encrypted block (MAC) twice?
  - This cannot add any security!

① Encrypt then MAC (Integrity check first then decryption)

② MAc the Encrypt $\boxed{P} \xrightarrow[k_2]{} \boxed{MAC} \xrightarrow[k_1]{} \boxed{E + MAC}$

③ Encrypt + MAC

$\boxed{P} \xrightarrow{P_1, k_1} \boxed{MAC}$

$\boxed{P} \xrightarrow[P_2, k_2]{} \boxed{E}$

# Confidentiality and Integrity

- Using different keys to encrypt and compute MAC works, even if keys are related
  - But, twice as much work as encryption alone
  - Can do a little better — about 1.5 “encryptions”
- Confidentiality *and* integrity with same work as one encryption is a still research topic…
  - Authenticated encryption (AE)
  - Optional reading materials on this topic.

# Quantum Computers and Symmetric Ciphers

- Quantum computers use weird properties of quantum mechanics
- These use “quantum bits” or “qubits”
  - Bits are either 0 or 1, but…
  - Qubits both 0 and 1? Or range 0 to 1?
- With N bits, in one of $2^N$ states
- With N qubits, $2^N$ quantum amplitudes
  - Which are continuous, not discrete, values, so potential for vastly more computing power

# Quantum Computers and Symmetric Ciphers

- Quantum computers are challenging to build and to program
- Special algorithms needed to take advantage of qubits
  - Need to use reversible logic operations
  - Results generally only probabalistic
- Today, quantum computers are small
  - IBM claims to have one with 433 qubits
  - Engineering challenges are immense

# Quantum Computers and Symmetric Ciphers

- Assuming big quantum computers are built, are they a threat to symmetric ciphers?
- Best quantum algorithm for exhaustive search is due to Grover (1996)
- Grover's algorithm is square root faster
- For n bit symmetric key…
  - Conventional computer: Work factor $2^{n-1}$
  - Grover's algorithm: Work factor about $2^{n/2}$

# Quantum Computers and Symmetric Ciphers

- Assuming big quantum computers are built, are they a threat to symmetric ciphers?
- Not really a serious threat
- If we double the length of the key, work factor for Grover's algorithm is same
- For AES, most common to use 128-bit key
  - But, we can use 256-bit keys
  - Just a little slower with 256 than 128-bit key

# Uses for Symmetric Crypto

- Confidentiality
  - Transmitting data over insecure channel
  - Secure storage on insecure media
- Integrity (MAC)
- Authentication protocols (later…)
- Hash function