Week 4: Symmetric-Key Cryptography

What is Symmetric-Key Encryption?

In symmetric-key encryption:

- **Same key** is used for both **encryption** and **decryption**.
- It's like using the same key to **lock and unlock** a door.
- The **sender and receiver must share the secret key** securely beforehand.

Symmetric vs. Asymmetric Cryptography

Feature	Symmetric-Key	Asymmetric-Key
Keys	Same key for encryption/decryption	Public/private key pair
Speed	Fast	Slower
Key distribution	Challenging	Easier (public keys can be shared)
Use case	Bulk data encryption	Key exchange, digital signatures

Role of Shared Keys & Confidentiality

- If a third party knows the key, they can decrypt your messages.
- **Key secrecy** is **essential** for confidentiality.
- Secure channels (e.g., using asymmetric cryptography) are used to exchange symmetric keys.

Importance of Key Size and Randomness

- **Key size**: Larger key → harder to brute-force.
    - 2^{40} = crackable, 2^{128} = extremely secure.
- **Randomness**: Predictable keys are vulnerable. Must use a cryptographically secure random number generator.

BLOCK vs. STREAM CIPHERS

Both are **symmetric-key algorithms**, but they differ in **how they process the data**:

Block Ciphers

Block ciphers **break the plaintext into fixed-size blocks** (like 64 or 128 bits) and then encrypt each block **independently or in a chained way**.

🔹 Key Characteristics:

- Operates on **fixed-size blocks** (e.g., AES uses 128-bit blocks).
- Requires **modes of operation** (like ECB, CBC, etc.) to handle large data or introduce randomness.
- Encryption is **deterministic** if using ECB (same input = same output).
- Needs **padding** if the message is not a multiple of block size.

Example:

If you want to encrypt: `“HELLOHELLO" (in binary)`

It’s split into blocks like:

`Block 1 → HELLO        Block 2 → HELLO`

Each block goes through the **same encryption logic**, but modes like CBC add chaining/randomness.

🔐 **Popular Block Ciphers:**

- **AES** (Advanced Encryption Standard)
- **DES** (Data Encryption Standard)
- **Blowfish**
- **Twofish**

Stream Ciphers

Stream ciphers **encrypt data one bit or byte at a time**, often by combining it with a **keystream**.

🔹 Key Characteristics:

- Processes data in a **continuous flow**, not fixed blocks.
- Keystream (pseudo-random bits) is **XOR-ed** with plaintext.
- **No padding needed**, works naturally with variable-length data.
- Very fast and ideal for **real-time applications** like voice calls or streaming.

Example:

If your message is: `"HELLO"`

The cipher creates a keystream like: `KEY → X Y Z P Q`

Then does:

`Ciphertext = H ⊕ X, E ⊕ Y, L ⊕ Z, ...`

Popular Stream Ciphers:

- **RC4** (no longer secure)
- **ChaCha20** (modern, fast, secure — used in TLS 1.3)
- **Salsa20**

Stream Cipher (XOR-based) in Python

- We'll generate a **keystream** using a pseudo-random number generator (PRNG).
- We'll XOR the plaintext with the keystream.
- Since XOR is reversible, decrypting = encrypting with the same keystream.

Simplified DES (S-DES)

What is S-DES?

Simplified DES (S-DES) is a *miniature version* of the real **Data Encryption Standard (DES)**. It helps us learn how DES works by using:

- Shorter **plaintext** (8 bits)
- Shorter **key** (10 bits)
- **2 rounds** of a Feistel network instead of 16

STRUCTURE OF S-DES

Feistel Structure

S-DES uses a **Feistel network**, which divides the 8-bit block into two 4-bit halves:

```
      Initial Permutation (IP)
                 ↓
        +-------------------+
        |  Left  |  Right  |
        |  4-bit |  4-bit  |
        +-------------------+
               ↓   ↓
        Feistel Function fK1
               ↓   ↓
            Swap Halves
               ↓   ↓
        Feistel Function fK2
               ↓   ↓
      Final Permutation (IP⁻¹)
```

Step-by-Step Transformations

Initial Permutation (IP)

- Permutes the 8-bit plaintext using a fixed table: IP = [1, 5, 2, 0, 3, 7, 4, 6]

If plaintext = `P0 P1 P2 P3 P4 P5 P6 P7`,

then output = `P1 P5 P2 P0 P3 P7 P4 P6`

Key Generation (from 10-bit key)

You start with a **10-bit key** (e.g., `1010000010`) and generate **two 8-bit round keys (K1 and K2)** via:

- A **P10 permutation**
- Two **left shifts**
- A **P8 permutation**

The Feistel Function f(Right, Key) =

- Expand and permute the 4-bit right half → 8 bits
- XOR with round key (8 bits)
- Apply **S-boxes** (S0 and S1) → 4-bit output
- Apply **P4 permutation** → 4-bit output

- XOR with the left half

Round 1 and Swap

After first fK1:

- XOR left with f(output), keep right unchanged
- Swap the two halves

Round 2 (fK2)

- Same as Round 1 but using second subkey (K2)
- **No swap** after this round

Let's see with an example :

Start with a 10-bit key:
Key = 1 0 1 0 0 0 0 0 1 0

Apply P10 permutation
P10 = [2, 4, 1, 6, 3, 9, 0, 8, 7, 5]

Original: 1 0 1 0 0 0 0 0 1 0
Positions: 0 1 2 3 4 5 6 7 8 9

P10 Result: 1 0 0 0 0 0 1 1 0 0
↑ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓
2 4 1 6 3 9 0 8 7 5

Split into two halves (5-bits each)
Left = [1, 0, 0, 0, 0]
Right = [0, 1, 1, 0, 0]

Left Shift (LS-1) both halves

Left Shift each half *circularly* by 1 bit:

LS-1(Left) = 0 0 0 0 1
LS-1(Right) = 1 1 0 0 0

Combine and apply P8 to get K1
Combined = 00001 11000 → 0000111000

P8 = [5, 2, 6, 3, 7, 4, 9, 8]

P8 Result =
bit at 5 → 1
bit at 2 → 0
bit at 6 → 1
bit at 3 → 0
bit at 7 → 0
bit at 4 → 1
bit at 9 → 0
bit at 8 → 0

K1 = 1 0 1 0 0 1 0 0

Left Shift again (LS-2)

Now shift **both halves by 2 bits** circularly:

LS-2(Left) = 0 0 1 0 0
LS-2(Right) = 0 0 0 1 1

Combine and apply P8 to get K2

Combined = 00100 00011 → 0010000011

Apply P8:

P8 Result =
bit at 5 → 0
bit at 2 → 1
bit at 6 → 0
bit at 3 → 0
bit at 7 → 0
bit at 4 → 0
bit at 9 → 1
bit at 8 → 1

K2 = 0 1 0 0 0 0 1 1

Final Subkeys

For 10-bit key `1010000010`, we got:

- **K1 = 10100100**
- **K2 = 01000011**

What is the Feistel Function `f(R, K)` in S-DES?

- **R** = 4-bit right half of the message
- **K** = 8-bit subkey (K1 or K2)

The Feistel function `f(R, K)` transforms `R` using `K` and returns a **4-bit output**.

Expansion & Permutation (E/P)

Take 4 bits of `R` and turn it into 8 bits using:

E/P = [3, 0, 1, 2, 1, 2, 3, 0]
For example, if `R = [1, 0, 1, 1]`,
E/P(R) = `[1, 1, 0, 1, 0, 1, 1, 1]`

XOR with Subkey (8 bits)

Now XOR this 8-bit expanded result with the 8-bit subkey K1 or K2.

Split into Left & Right 4-bits

This gives us two 4-bit halves. Each half will be sent to an **S-box.**

S-Boxes: S0 and S1

Each S-box takes a 4-bit input and returns a 2-bit output.

S0:

C0 C1

R0 01 00 11 10

R1 11 10 01 00

R2 00 10 01 11

R3 11 01 11 10

S1:

C0 C1

R0 00 01 10 11

R1 10 00 01 11

R2 11 00 01 00

R3 10 01 00 11

To use them:

- For each 4-bit half:
  - Use outer bits as row (2 bits)
  - Use inner bits as col (2 bits)

Combine 2-bit outputs → 4 bits

Apply P4 permutation : P4 = [1, 3, 2, 0]

Output: Return this 4-bit result. In the S-DES round, we'll XOR it with the left half of the message.

Simplified AES (S-AES)

High-Level Architecture of S-AES

Simplified AES is a reduced version of AES intended for learning. Instead of operating on 128-bit blocks and 128/192/256-bit keys, S-AES:

- Operates on **16-bit plaintext blocks**
- Uses a **16-bit key**
- Performs **2 rounds** of encryption (plus an initial AddRoundKey)

S-AES Encryption Overview

Rounds:

1. **Initial AddRoundKey**
2. **Round 1**
   - SubBytes
   - ShiftRows
   - MixColumns
   - AddRoundKey (with Round Key 1)
3. **Round 2 (Final Round)**
   - SubBytes
   - ShiftRows
   - AddRoundKey (with Round Key 2)

What is SubBytes?

In AES and S-AES, **SubBytes** is a **non-linear byte substitution** step that uses an **S-box** (Substitution box). It adds **confusion** to the cipher by replacing each nibble (4-bit half-byte) with another according to a fixed table.

The S-AES S-box

In S-AES, each 4-bit nibble is replaced using this table:

Input (Hex)	Output (Hex)
0	9
1	4
2	A
3	B
4	D
5	1
6	8
7	5
8	6
9	2
A	0
B	3
C	C
D	E
E	F
F	7

How SubBytes Works in S-AES

1. We treat the 16-bit state as **4 nibbles** (each 4 bits).
2. We substitute each nibble using the S-box.

For example:

State: `1011 0010 1111 0101`
→ Nibbles: `B`, `2`, `F`, `5`
→ S-Box maps: `B → 3`, `2 → A`, `F → 7`, `5 → 1`
→ Output: `0011 1010 0111 0001`

What Is ShiftRows?

In AES and S-AES, **ShiftRows** is a **simple permutation** step that:

- Keeps the first row **unchanged**
- **Shifts the second row** to the **left**

This adds **diffusion**, ensuring that bits from one part of the state influence another.

State Format in S-AES

We represent the 16-bit state as a **2×2 nibble matrix**:

```
+----+----+     +-----------------------------------+

| a0 | a2 |  →    | Row 0: a0 a2 (unchanged)

| a1 | a3 |  →    | Row 1: a3 a1 (left shift by 1)

+----+----+     +-----------------------------------+
```

- First row stays the same: `a0`, a2
- Second row rotates left: a1, a3 → a3, a1

What Does MixColumns Do?

This step **mixes the data within each column** using a special kind of multiplication in a field called **GF(2^4)** (Galois Field of 4 bits). It provides **diffusion**, meaning it spreads the effect of one input nibble across multiple output nibbles.

The State Matrix in S-AES

We'll still use the 2×2 matrix representation. Suppose after SubBytes and ShiftRows, your state is:

[s0 s2]

[s1 s3]

These columns are now mixed using matrix multiplication in GF(2^4):

New state =

[1 4] ⊗ [s0 s2] = [s0 ⊕ (4 • s1)   s2 ⊕ (4 • s3)]

[4 1]   [s1 s3]   [(4 • s0) ⊕ s1   (4 • s2) ⊕ s3]

"⊕" means XOR

"4 · a1" is multiplication in $GF(2^4)$

This is NOT like decimal multiplication. We do:

1. Multiply like binary polynomials
2. Then reduce mod $x4+x+1\rightarrow$ `0b10011` or `0x13`

Example: Multiply 4 · 7 in $GF(2^4)$

Step 1: Represent numbers as 4-bit binary

- $4 = 0100 = x^2$
- $7 = 0111 = x^2 + x + 1$

So: $(4 \cdot 7) = (x^2) \cdot (x^2 + x + 1) = x^4 + x^3 + x^2$

Now reduce modulo $x^4 + x + 1$ (i.e., $x^4 = x + 1$):

$x^4 + x^3 + x^2 \equiv (x + 1) + x^3 + x^2 = x^3 + x^2 + x + 1$

$\rightarrow$ binary = 1111 = 0xF

What is AddRoundKey?

This is the **simplest but most important** step in every AES round. It mixes the data (plaintext) with the key using **bitwise XOR**.

> **Each byte/nibble of the state is XORed with the corresponding byte/nibble of the round key.**

It's where the **confusion** comes in — without the key, the ciphertext becomes meaningless.

In S-AES: the State and Key

In S-AES, both:

- **State** = 4 nibbles (4 × 4-bit = 16-bit)
- **RoundKey** = 4 nibbles (from key expansion)

State Matrix Format (after SubBytes, ShiftRows):

We usually store the 4 nibbles as:

[s0 s2]

[s1 s3]

Key Format:

Round key is also 4 nibbles, laid out similarly:

[k0 k2]

[k1 k3]

new_s0 = s0 ⊕ k0

new_s1 = s1 ⊕ k1

new_s2 = s2 ⊕ k2

new_s3 = s3 ⊕ k

When is AddRoundKey used?

- At the start of AES: plaintext ⊕ first round key (initial AddRoundKey)
- After every round's MixColumns (except final round)
- At the end: just before the output

What is Key Expansion?

In S-AES, you start with a **16-bit key** (4 nibbles), and expand it into **three 16-bit round keys**:

- `K0` (used in the initial AddRoundKey)
- `K1` (for round 1)
- `K2` (for round 2 / final round)

Input:

A 16-bit key, for example:

Key = 1010 0111 0011 1011 = 0xA73B

Step-by-step output:

1. Split into two 8-bit words:
   `w0, w1`
2. Generate w2, w3, w4, w5 using rules below
3. Build:
    - RoundKey0 = w0 || w1

- RoundKey1 = w2 || w3
- RoundKey2 = w4 || w5

Steps: Word Generation Rules

- `w0 = first 8 bits`
- `w1 = second 8 bits`
- `w2 = w0 ⊕ g(w1)`
- `w3 = w2 ⊕ w1`
- `w4 = w2 ⊕ g(w3)`
- `w5 = w4 ⊕ w3`

What is the `g()` function?

This is the **core of key expansion** and involves:

1. **Rotate**: Swap the 2 nibbles of the byte
2. **Substitute**: Apply S-AES S-box to each nibble
3. **XOR with Round Constant**

S-AES S-box:

0000 → 1001	0001 → 0100	0010 → 1010	0011 → 1011
0100 → 1101	0101 → 0001	0110 → 1000	0111 → 0101
1000 → 0110	1001 → 0010	1010 → 0000	1011 → 0011
1100 → 1100	1101 → 1110	1110 → 1111	1111 → 0111

Round Constants (Rcon):

- `Rcon1 = 10000000` (0x80)
- `Rcon2 = 00110000` (0x30)

Let's say the key is:

Key = 1010011100111011 = 0xA73B

w0 = 10100111 = 0xA7

w1 = 00111011 = 0x3B

1. **w2 = w0 ⊕ g(w1)**
   - Rotate w1 = 00111011 → 10110011 (0xB3)

    - Substitute: apply S-box on each nibble
        - B (1011) → 0011
        - 3 (0011) → 1011

        → g(w1) = 00111011 = 0x3B
    - g(w1) ⊕ Rcon1 = 00111011 ⊕ 10000000 = 10111011 (0xBB)
    - w2 = w0 ⊕ g(w1 ⊕ Rcon1) = 10100111 ⊕ 10111011 = 00011100 = 0x1C
2. **w3 = w2 ⊕ w1 = 00011100 ⊕ 00111011 = 00100111 = 0x27**
3. **w4 = w2 ⊕ g(w3)**
    - Rotate w3 = 00100111 → 01110010 (0x72)
    - Substitute:
        - 7 → 0101, 2 → 1010 → 01011010
    - g(w3) = 01011010 ⊕ Rcon2 = 01011010 ⊕ 00110000 = 01101010 (0x6A)
    - w4 = 00011100 ⊕ 01101010 = 01110110 = 0x76
4. **w5 = w4 ⊕ w3 = 01110110 ⊕ 00100111 = 01010001 = 0x51**

Final Round Keys:

- RoundKey0 = w0 || w1 = 0xA7 || 0x3B = `A73B`
- RoundKey1 = w2 || w3 = 0x1C || 0x27 = `1C27`
- RoundKey2 = w4 || w5 = 0x76 || 0x51 = `7651`

Example

Input:

- **Plaintext** = `0x6565` (binary: `0110010101100101`)
- **Key** = `0x0100` (binary: `0000000100000000`)

Key Expansion

From key `0x0100`, generate 3 round keys:

RoundKey0 = w0 || w1 = 01 00

RoundKey1 = w2 || w3 = EF 6F

RoundKey2 = w4 || w5 = 85 97

Initial AddRoundKey

State = Plaintext ⊕ RoundKey0 = 6565 ⊕ 0100 = 6445

Round 1

a. SubBytes

Apply S-box to each nibble of 6445:

6 → 1000 (8)   4 → 1101 (D)   4 → 1101 (D)   5 → 0001 (1)   → Output = 8DD1

b. ShiftRows

S-AES operates on 2×2 nibbles:

[8 D] [D 1] → After ShiftRows → [8 D] [1 D] → 8 1 D D

c. MixColumns

Do GF(2^4) matrix multiplication — let's assume (from standard S-AES definition) this transforms `81DD` to: State after MixColumns = 9EB3

d. AddRoundKey

State ⊕ RoundKey1 = 9EB3 ⊕ EF6F = 71DC

Final Round (Round 2)

a. SubBytes

Apply S-box to each nibble of 71DC:

7 → 0101 (5)   1 → 0100 (4)   D → 1110 (E)   C → 1100 (C)   → Output = 54EC

b. ShiftRows

[5 4] [E C] → After ShiftRows → [5 4] [C E] → 5C4E

c. No MixColumns in final round

d. Final AddRoundKey

State ⊕ RoundKey2 = 5C4E ⊕ 8597 = D9D9

Final Ciphertext: CIPHERTEXT = D9D9

Decryption Steps

1. Ciphertext = `D9D9`
2. ⊕ RoundKey2 = `5C4E`
3. Inverse ShiftRows → `54EC`
4. Inverse SubBytes → `71DC`

5. ⊕ RoundKey1 = `9EB3`
6. Inverse MixColumns → `81DD`
7. Inverse ShiftRows → `8DD1`
8. Inverse SubBytes → `6445`
9. ⊕ RoundKey0 = `6565` = Plaintext

Modes of Operation for Block Ciphers

Why we need modes:

- Block ciphers encrypt fixed-size blocks only.
- Real messages are longer and not always a multiple of block size.
- Modes let us securely encrypt messages of arbitrary length and add security properties.

Common Modes:

1. ECB (Electronic Codebook)

- Encrypt each block independently.
- Simple but insecure: identical plaintext blocks produce identical ciphertext blocks.
- Vulnerable to pattern leakage.

2. CBC (Cipher Block Chaining)

- Each plaintext block XORed with previous ciphertext block before encryption.
- Uses an Initialization Vector (IV) for the first block.
- Prevents identical blocks from producing identical ciphertext.
- Widely used, but requires sequential processing.

3. CFB (Cipher Feedback)

- Converts block cipher into a self-synchronizing stream cipher.
- Encrypts previous ciphertext to generate a keystream, XOR with plaintext.
- Can encrypt data in units smaller than block size.

4. OFB (Output Feedback)

- Similar to CFB but keystream is generated independently of plaintext/ciphertext.
- Turns block cipher into a synchronous stream cipher.
- Errors in ciphertext do not propagate.

5. CTR (Counter Mode)

- Generates keystream by encrypting incrementing counters.

- Allows parallel encryption/decryption.
- Very efficient and widely used.

Initialization Vectors (IVs)

- Random, unique values used to start modes like CBC, CFB, OFB.
- Ensure that identical plaintexts encrypt differently each time.
- Must be unpredictable and never reused with the same key.

Padding Schemes

- Messages not multiple of block size need padding.
- Common schemes:
    - **PKCS#7**: Pad with bytes equal to the number of padding bytes.
    - **ANSI X.923**: Pad with zeros and last byte is number of padding bytes.

Week 4 : Resources Used - Introduction To Modern Cryptography (Jonathan Katz, Yehuda Lindell) |
sage-cryptography | Simplilearn