# TABLE OF CONTENTS

# VERNAM CIPHER

The **Vernam Cipher** is a **symmetric encryption algorithm** that uses a **one-time pad**. It encrypts data using the **bitwise XOR (exclusive OR)** operation between each byte of the **plaintext** and a **randomly generated key** of the same length.

**Important rule**:

> The key must be exactly the same length as the message or file being encrypted.

If properly implemented with a truly random key used only once, the Vernam cipher is **provably unbreakable**.

implemented both **text** and **file** encryption using the Vernam Cipher.

| Step | Action |
|---|---|
| 1️⃣ | Generate a random key of the same length as the text/file |
| 2️⃣ | XOR each byte/character of plaintext with the key |
| 3️⃣ | Result is ciphertext (or encrypted file) |
| 4️⃣ | XOR ciphertext with the same key again to get original text/file back |

## ✅ Key Properties of Vernam in Your Code

- 🔁 **Symmetric**: Same key for encryption and decryption
- 🔒 **Secure if key is random, same length, and never reused**
- ⚠️ **Fails if key is reused or mismatched in length**
- 🔄 **XOR is self-reversible**: `C = M ⊕ K`, then `M = C ⊕ K`

- Text/File mode switch
- Separate Encrypt/Decrypt tabs under File mode
- Toggle to use key text input or upload `.txt` file for decryption
- Key generation with progress bar
- File download progress bar
- Auto-download of encrypted/decrypted file
- Optional key download as `.txt`
- Clear Fields button in both Encrypt and Decrypt sections

A **"Clear Fields"** button below the Encrypt and Decrypt buttons.

It will reset:

- `file`
- `key`
- `realKey`
- `keyFile`
- `message`
- `error`

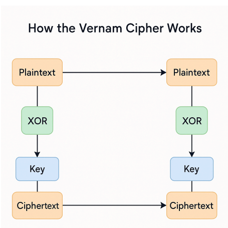
1. Tabs under the "File Encryption" section:

- 🟢 **Encrypt File**
- 🟡 **Decrypt File**

2. Inside Decrypt File tab:

- Toggle:
  - 🔘 **Paste key in textbox**
  - 🔘 **Upload key file (.txt)**

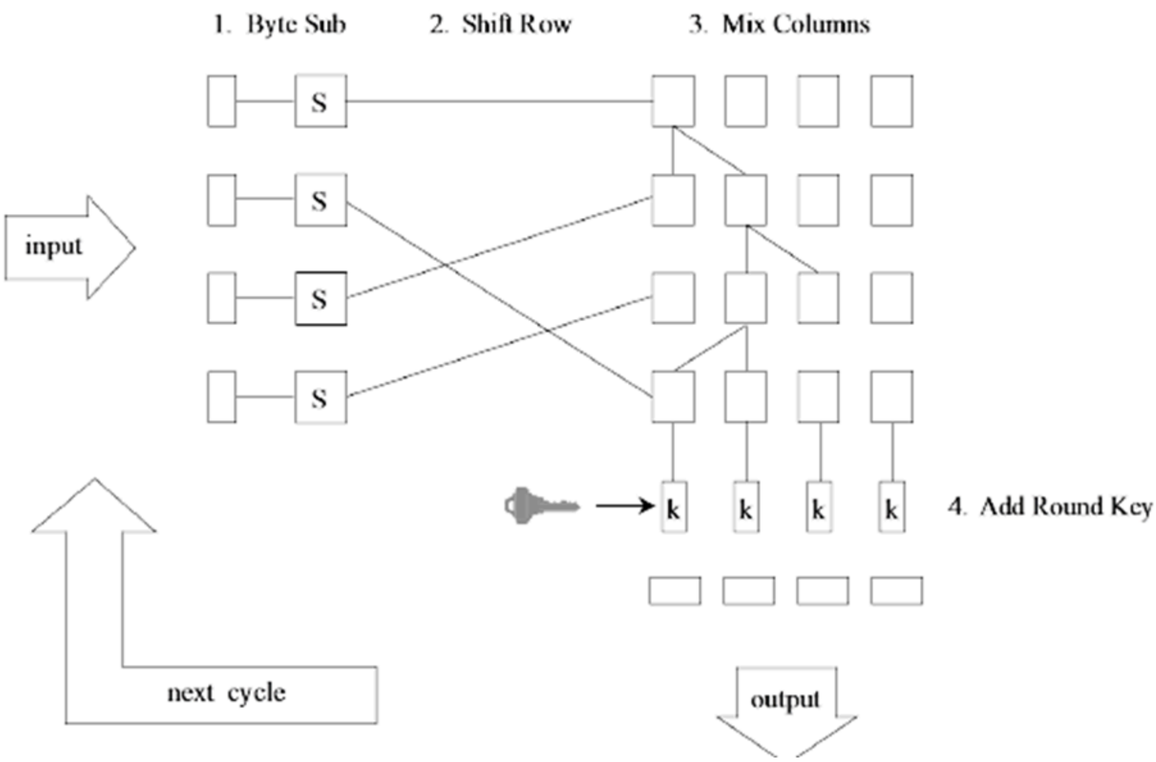
**Automatically download `.txt` for large keys**
**Prompt and optionally download for small keys**



# VIGENERE CIPHER

# TRANSPOSITION CIPHER

# AES




| | DES | AES |
|---|---|---|
| **Date designed** | 1976 | 1999 |
| **Block size** | 64 bits | 128 bits |
| **Key length** | 56 bits (effective length); up to 112 bits with multiple keys | 128, 192, 256 (and possibly more) bits |
| **Operations** | 16 rounds | 10, 12, 14 (depending on key length); can be increased |
| **Encryption primitives** | Substitution, permutation | Substitution, shift, bit mixing |
| **Cryptographic primitives** | Confusion, diffusion | Confusion, diffusion |
| **Design** | Open | Open |
| **Design rationale** | Closed | Open |
| **Selection process** | Secret | Secret, but open public comments and criticisms invited |
| **Source** | IBM, enhanced by NSA | Independent Dutch cryptographers |

## 🔐 Advanced Encryption Standard (AES)

### 🗓️ Overview

AES was designed in **1999** by **independent Dutch cryptographers** as a secure replacement for DES. It is based on the **Rijndael algorithm** and selected through an **open public competition**.

## 🧱 Block and Key Size

- **Block size**: Always **128 bits**.
- **Key sizes**: Can be **128, 192, or 256 bits**.
    - The number of **rounds** depends on the key size:
        - 128-bit key → 10 rounds
        - 192-bit key → 12 rounds
        - 256-bit key → 14 rounds

## 🔄 How AES Works (Structure)

AES follows a **Substitution-Permutation Network** structure and works on the entire data block at once:

1. **SubBytes**: Each byte is replaced using a substitution box (S-box) to introduce confusion.
2. **ShiftRows**: Each row of the 4×4 block matrix is shifted left by a certain offset to spread the data.
3. **MixColumns**: Each column is mixed using a mathematical function (matrix multiplication) to increase diffusion.
4. **AddRoundKey**: The current block is XORed with a round key derived from the original key.

These steps are repeated for the specified number of rounds, with the **MixColumns** step omitted in the final round.

## 🧠 Encryption Concepts Used

- **Primitives**: Substitution, shift, and bit mixing.
- **Cryptographic goals**: Confusion and diffusion.
- **Design**: Fully **open** and publicly reviewable.

- **Design rationale**: **Open**, with clear documentation and analysis.
- **Selection process**: Transparent, with public input and critique.

### 🛡️ Strengths of AES

- Supports long keys, which provide higher resistance to brute-force attacks.
- Designed with modern threats in mind.
- Open and publicly vetted.
- Efficient in both software and hardware.

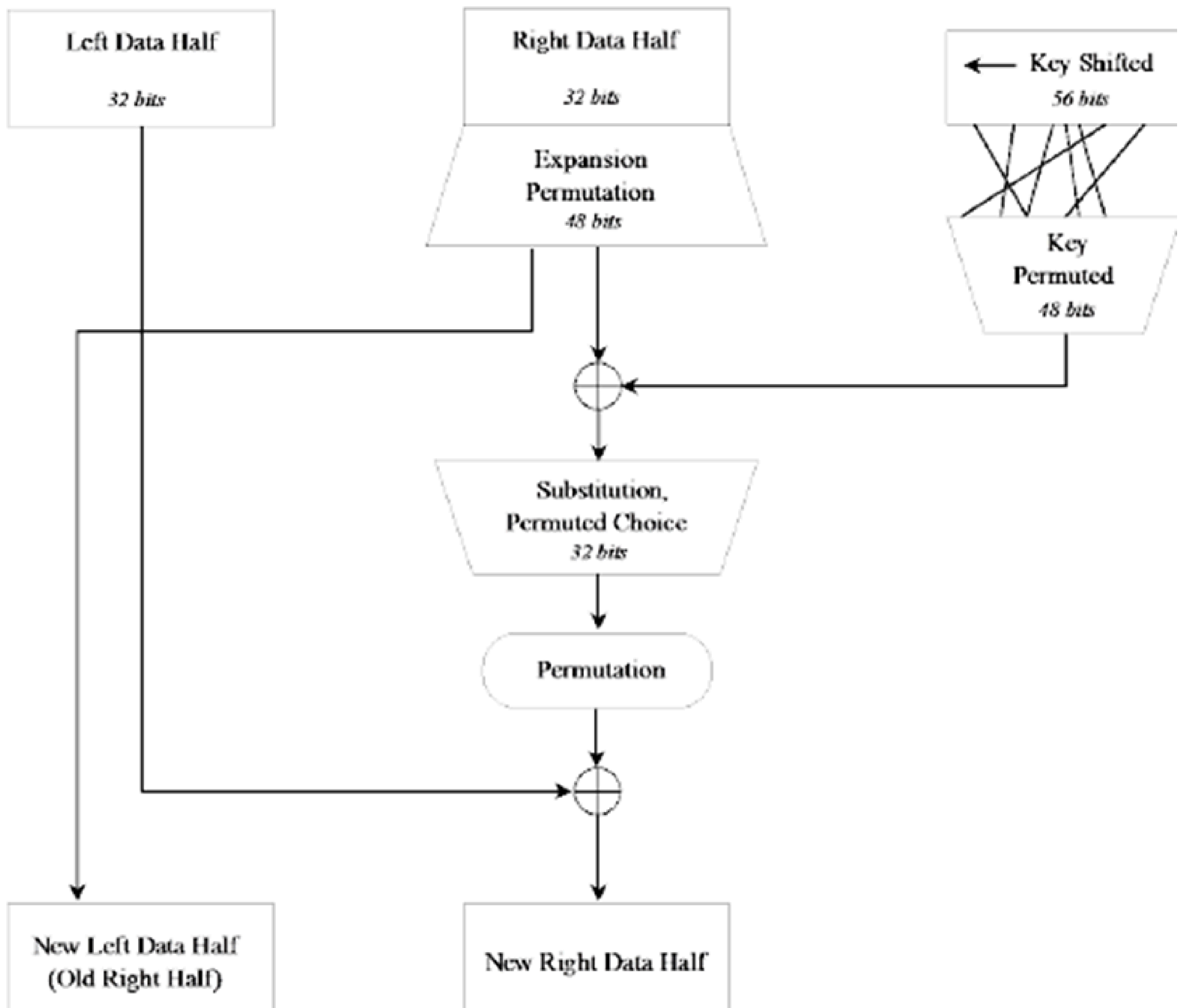
### Longevity of AES

• Since its initial publication in 1997, AES has been extensively analyzed, and the only serious challenges to its security have been highly specialized and theoretical

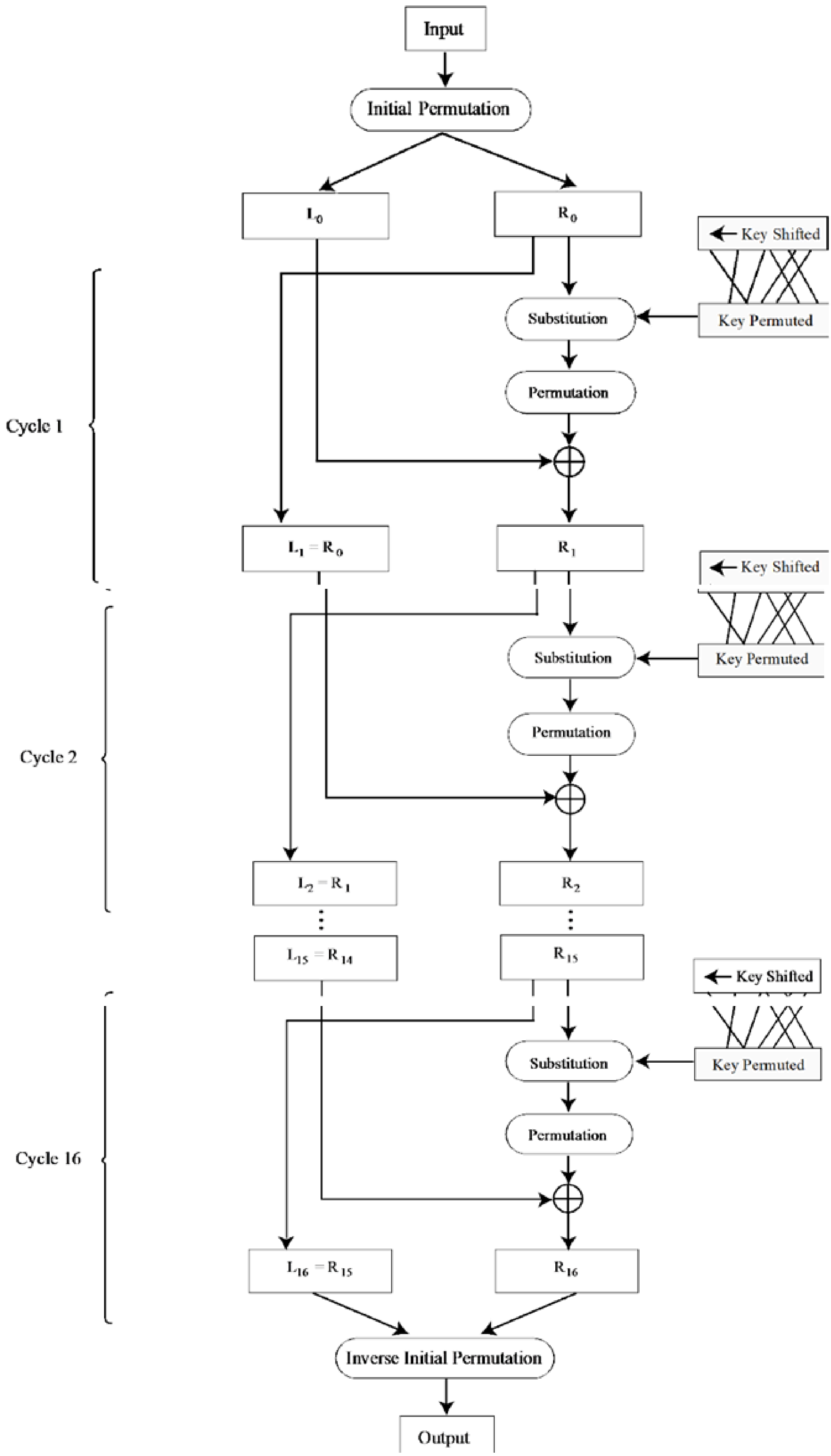
• Because there is an evident underlying structure to AES, it will be possible to use the same general approach on a slightly different underlying problem to accommodate keys larger than 256 bits when necessary

• No attack to date has raised serious question as to the overall strength of AES

# DES



Left Data Half
32 bits
Right Data Half
32 bits
Key Shifted
56 bits
Expansion Permutation
48 bits
Key Permuted
48 bits
Substitution, Permuted Choice
32 bits
Permutation
New Left Data Half (Old Right Half)
New Right Data Half



Input
Initial Permutation
$L_0$
$R_0$
Key Shifted
Substitution
Key Permuted
Permutation
Cycle 1
$L_1 = R_0$
$R_1$
Key Shifted
Substitution
Key Permuted
Permutation
Cycle 2
$L_2 = R_1$
$R_2$
$L_{15} = R_{14}$
$R_{15}$
Key Shifted
Substitution
Key Permuted
Permutation
Cycle 16
$L_{16} = R_{15}$
$R_{16}$
Inverse Initial Permutation
Output

| | DES | AES |
|---|---|---|
| Date designed | 1976 | 1999 |
| Block size | 64 bits | 128 bits |
| Key length | 56 bits (effective length); up to 112 bits with multiple keys | 128, 192, 256 (and possibly more) bits |
| Operations | 16 rounds | 10, 12, 14 (depending on key length); can be increased |
| Encryption primitives | Substitution, permutation | Substitution, shift, bit mixing |
| Cryptographic primitives | Confusion, diffusion | Confusion, diffusion |
| Design | Open | Open |
| Design rationale | Closed | Open |
| Selection process | Secret | Secret, but open public comments and criticisms invited |
| Source | IBM, enhanced by NSA | Independent Dutch cryptographers |

# 🔐 Data Encryption Standard (DES)

### 🗓️ Overview

DES was introduced in **1976**, originally developed by **IBM** and later modified by the **NSA**. It was once a U.S. government standard but is now considered insecure due to its short key length.

### 🧱 Block and Key Size

- **Block size**: **64 bits**.
- **Key length**: Officially **56 bits** (with 8 parity bits, total 64 bits).
    - **Multiple keys** can increase effective length up to **112 bits** (e.g., in Triple DES).

//WE ARE USING TRIPLEDES (I THINK) WITH A LONGER KEY SIZE

### 🔄 How DES Works (Structure)

DES uses a **Feistel structure** and operates in 16 rounds:

1. The 64-bit block is divided into two 32-bit halves: **Left (L)** and **Right (R)**.
2. For each round:
    - The right half (R) is passed through a function **F**, involving substitution and permutation, then XORed with the left half (L).

- The halves are swapped.

The final output is a recombination of the two halves after 16 rounds.

### 🔍 Encryption Concepts Used

- **Primitives**: Substitution and permutation.
- **Cryptographic goals**: Confusion and diffusion.
- **Design**: **Open**, but limited in transparency.
- **Design rationale**: **Closed**, with NSA's exact modifications not fully disclosed.
- **Selection process**: **Secret**, with no public involvement.

### ⚠️ Weaknesses of DES

- **Short key length (56 bits)** makes it vulnerable to brute-force attacks.

//WE ARE USING A LONGER KEY

- Susceptible to differential and linear cryptanalysis.
- **Lack of transparency** in the design raised concerns.
- Largely replaced by AES and more secure alternatives like **Triple DES** or **Blowfish**.

DES Decryption:

$$L_j = R_{j-1} \quad (1)$$

$$R_j = L_{j-1} \oplus f(R_{j-1}, k_j) \quad (2)$$

By rewriting these equations in terms of $R_{j-1}$ and $L_{j-1}$, we get

$$R_{j-1} = L_j \quad (3)$$

and

$$L_{j-1} = R_j \oplus f(R_{j-1}, k_j) \quad (4)$$

Substituting (3) into (4) gives

$$L_{j-1} = R_j \oplus f(L_j, k_j) \quad (5)$$

**Chaining**

• DES uses the same process for each 64-bit block, so two identical blocks encrypted with the same key will have identical output

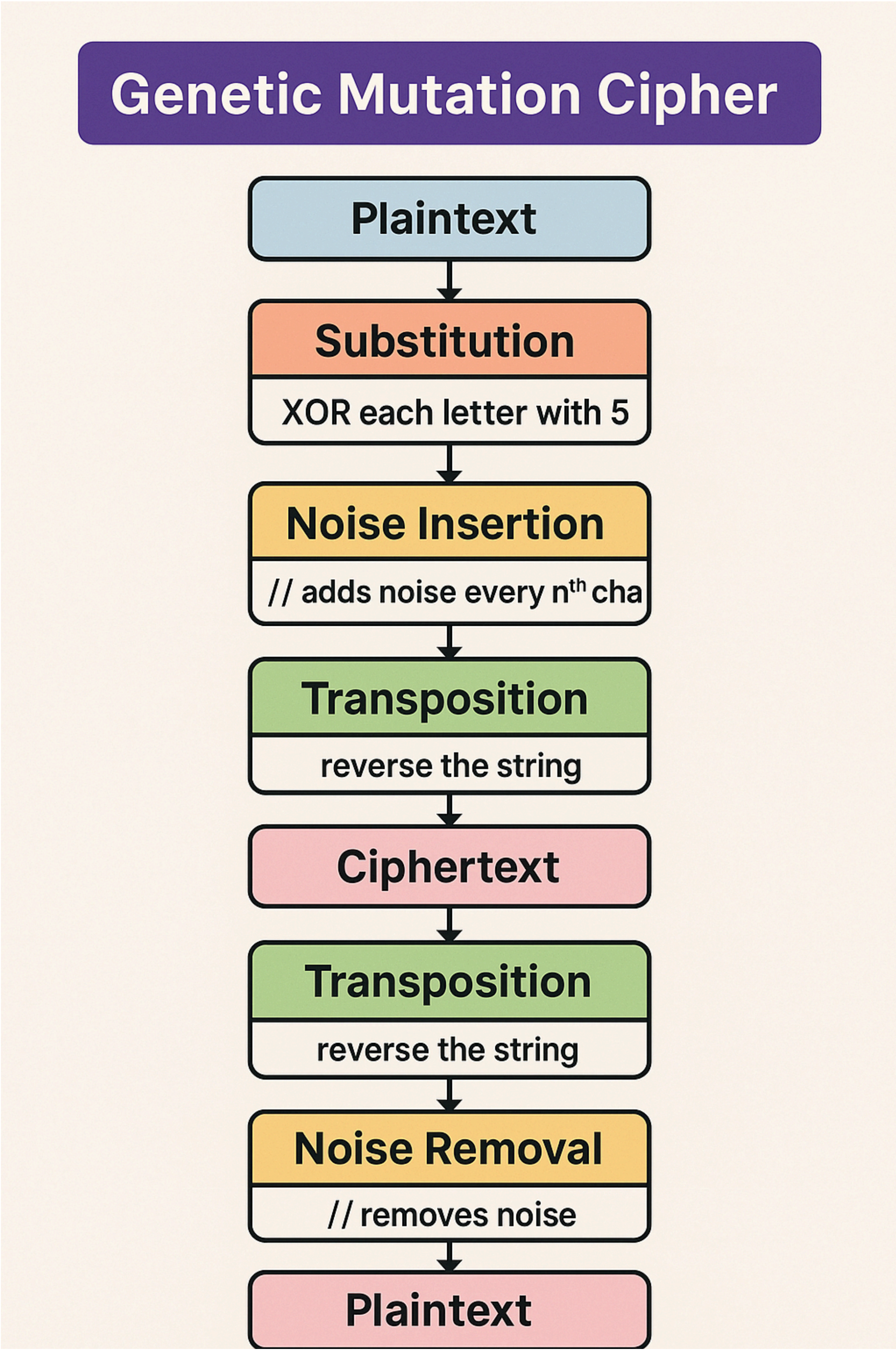
• This provides too much information to an attacker, as messages that have common beginnings or endings, for example, are very common in real life, as is reuse of a single key over a series of transactions

• The solution to this problem is chaining, which makes the encryption of each block dependent on the content of the previous block as well as its own content

# CEASAR CIPHER

CUSTOM - EMOJICODE

# CUSTOM - GENETIC MUTATION CIPHER




**KEY IMPORTANCE**

**The formula for noise interval is:**

```
interval = (key.length % 3) + 2
```

**If the key length is short (e.g., 3), interval = 2.**

**If the key is long (e.g., 12), interval = 2 again (12 % 3 = 0, +2 = 2).**

**If the key is 7, interval = (7 % 3) + 2 = 3.**

2. XOR Shifting Pattern

**Every character or byte is XORed with a character from the key:**

**js**
**CopyEdit**
```js
buffer[i] ^ key.charCodeAt(i % key.length)
```

- 
- **So the longer your key, the more diverse the XOR pattern will be.**
- **This helps in making the output less predictable and more secure.**

**A longer key means more XOR variety.**

**It spreads the pattern out more and makes brute-forcing harder.**

**But it still follows the same structure — just more scrambled!**

**Every few characters (depending on the length of your key), we add an asterisk `*` as "junk DNA" or noise**

**TEXT DECRYPTION**

**First, we decode the base64-encoded ciphertext.**

**Then, we remove the inserted noise/junk (all asterisks *).**

**We reverse the text again.**

**Finally, we XOR again with ^ 5 to restore the original characters (since XOR is reversible).**

## Genetic Mutation Cipher (GMC) - Simplified Explanation

The Genetic Mutation Cipher (GMC) is a custom encryption method that works on both text and files. It mimics genetic mutations by applying a combination of substitution, noise insertion, and transposition. These transformations obfuscate the original message and make it hard to reverse without the correct key.

### Text Encryption Process

Step 1: Substitution

Each character is XORed with the number 5. This means we modify the character's binary value by flipping certain bits to create a different character.

Step 2: Transposition

We reverse the entire substituted string. This adds further confusion by changing the character order.

Step 3: Noise Insertion

We insert a '*' symbol every few characters. The exact frequency is determined by the key length. This makes it harder for attackers to distinguish original characters from noise.

### Example Code for Text Encryption

```
plaintext.replace(/[a-z]/gi, c => String.fromCharCode(c.charCodeAt(0) ^ 5));

 text.split('').reverse().join('');

 ...map((c, i) => (i + 1) % ((key.length % 3) + 2) === 0 ? c + '*' : c).join('');

 base64 encoded final string
```

## Text Decryption Process

Step 1: Base64 Decode

We decode the base64 ciphertext to get the raw encrypted string.

Step 2: Noise Removal

We remove all '*' characters to restore the clean transposed string.

Step 3: Reverse Transposition

We reverse the string back to its original character order.

Step 4: Reverse Substitution

We apply XOR with 5 again to restore the original characters.

## File Encryption & Decryption

Files are encrypted by XORing each byte with a character from the key. Every few bytes, a noise byte is inserted.

Encryption Function (simplified):

```
byte = buffer[i] ^ key.charCodeAt(i % key.length);
 if ((i + 1) % interval === 0) add noise byte
```

Decryption Function (simplified):

```
if ((i + 1) % (interval + 1) === 0) skip;
 byte = buffer[i] ^ key.charCodeAt(skip % key.length)
```

## Key Importance in GMC

The key is central to GMC. It determines:

- How characters or bytes are XORed.

- How frequently noise is added (interval = (key.length % 3) + 2).

A longer key results in a more complex XOR pattern and less predictable encryption.