# Security and Vulnerability in Networks Assignment 1


## Abstract

This project explores the development and enhancement of a classical encryption scheme based on **transposition** and **Caesar cipher**. Initially, the transposition-Caesar cipher combination demonstrated minimal avalanche effect, highlighting its limitations in propagating small changes in the plaintext across the ciphertext. To improve the encryption's effectiveness, we introduced the **Vigenère cipher**, aiming to enhance diffusion. While this addition provided some improvement, it still failed to achieve significant avalanche effects.

Next, I implemented **Cipher Block Chaining (CBC) mode** over the transposition-Caesar encryption, introducing a chaining mechanism between blocks to amplify diffusion. Although this approach increased the complexity of the encryption, the overall impact on the avalanche effect remained limited due to the inherent weaknesses of the classical ciphers.

Finally, I applied **AES (Advanced Encryption Standard)** in **CBC mode** as a modern solution to significantly improve the security and avalanche effect. AES's inherent strength in diffusion and chaining, combined with CBC mode, resulted in a robust encryption system capable of achieving the desired avalanche effect. This project demonstrates how classical ciphers, even when enhanced with modern techniques, are outperformed by state-of-the-art cryptographic standards like AES in terms of security and performance.


## 1. Introduction

Cryptography provides various techniques to ensure confidentiality, integrity, and authenticity of data. Two fundamental categories of classical cryptographic methods are **substitution ciphers**, which replace characters with others, and **transposition ciphers**, which reorder characters without altering them.

Despite their historical significance, these ciphers have inherent weaknesses, such as susceptibility to **frequency analysis** and limited **diffusion**, making them insufficient for modern security standards. To address these limitations, more advanced encryption algorithms, such as the **Advanced Encryption Standard (AES)**, have been developed. AES, particularly in **Cipher Block Chaining (CBC) mode**, provides robust security by incorporating techniques that ensure a high avalanche effect, where small changes in the plaintext result in significant changes in the ciphertext.

This project explores the implementation of classical and modern cryptographic methods, including the limitations of transposition and substitution ciphers, and the improvements introduced by block ciphers like AES. Through practical experimentation, the goal is to demonstrate the importance of using strong encryption techniques to achieve higher security, focusing on the **avalanche effect** as a key indicator of cryptographic strength.

# 2. Design and implementation

## 2.1. Apply encryption

I've chosen option A: **Transposition followed by Substitution**.

- **Transposition**:

  In my case, my phone's last digits are 23311. Based on the explained logic, this would result in the transposition key **34512**.

  The plaintext is divided into columns based on the length of the key. Then, the columns are reordered according to the key's sequence and finally, the ciphertext is generated by concatenating the characters column by column. The process of applying transposition to the original text results in:

| 3 | 4 | 5 | 1 | 2 |
|---|---|---|---|---|
| A | n | t | o | n |
| _ | M | a | e | s |
| t | r | e | _ | S |
| e | c | u | r | i |
| t | y | _ | a | n |
| d | _ | V | u | l |
| n | e | r | a | b |
| i | l | i | t | y |
| _ | i | n | _ | N |
| e | t | w | o | r |
| k | s | _ | _ | _ |

```
Original plain text:  Anton Maestre Security and Vulnerability in Networks
Transposed plain text:  oe rauat o nsSinlbyNr A tetdni eknMrcy elitstaeu Vrinw
```

- **Substitution**:
  In this part, each letter in the transposed text is shifted according to the Caesar cipher key (in this case, **1**). The final encrypted text is obtained by shifting each letter one position forward:

```
Transposed plain text:  oe rauat o nsSinlbyNr A tetdni eknMrcy elitstaeu Vrinw
Final ciphertext:  pf sbvbu p otTjomczOs B ufueoj floNsdz fmjutubfv Wsjox
```

This combination is chosen because it first disorganizes the structure of the original text, making it less susceptible to pattern recognition. Then, the Caesar cipher adds another layer of complexity by modifying each character individually. This approach enhances the overall security of the encrypted text.

## - Implementation

To perform the transposition on the text string, I have designed the **transposition_encrypt** function. This function first ensures that the plaintext's length is suitable for matrix processing by padding it with spaces. It then constructs a matrix of rows based on the key's length, and finally creates an encrypted text by rearranging the characters based on the order of the key.

```python
# Transposition function
def transposition_encrypt(plaintext, key):
    # Ensure the plaintext length is divisible by the key length
    while len(plaintext) % len(key) != 0:
        plaintext += ' '

    # Create a matrix based on the key length
    n = len(key)
    matrix = [plaintext[i:i+n] for i in range(0, len(plaintext), n)]

    # Create transposed text based on key order
    sorted_key = sorted(list(key))
    transposed_text = ''
    for k in sorted_key:
        column_index = key.index(k)
        for row in matrix:
            transposed_text += row[column_index]

    return transposed_text
```

On the other hand, to perform substitution using the Caesar Cipher, I have developed the **caesar_cipher_encrypt** function. It performs encryption by shifting each alphabetical character in the plaintext by a specified number passed as a parameter. While non-alphabetic characters are not changed, those that are alphabetic have the shift applied to their ASCII value, avoiding exceeding the limits of the alphabet.

```python
# Caesar cipher encryption function
def caesar_cipher_encrypt(plaintext, shift):
    encrypted_text = ''
    for char in plaintext:
        if char.isalpha():  # Encrypt letters only
            shift_amount = 65 if char.isupper() else 97
            encrypted_text += chr((ord(char) - shift_amount + shift) % 26 + shift_amount)
        else:
            encrypted_text += char  # Leave other characters unchanged

    return encrypted_text
```

## 2.2. Evaluate the Avalanche Effect

With the aim of testing the avalanche effect, I performed the encryption on both the original plaintext and on one with a small modification that changed a single character to another. When I analysed the differences between both results by observing the bits, I noticed that they are minimal.

```
Avalanche Effect (Single Round): 0.45% difference in bits
```

Even after conducting different tests in which I increased the number of encryptions by one in each round, the difference between both encryptions remained minimal, and therefore, the avalanche effect was practically non-existent.

```
Round 1: Avalanche Effect = 0.45%, Time = 0.0083 seconds
Round 2: Avalanche Effect = 0.45%, Time = 0.0055 seconds
Round 3: Avalanche Effect = 0.45%, Time = 0.0121 seconds
Round 4: Avalanche Effect = 0.45%, Time = 0.0078 seconds
Round 5: Avalanche Effect = 0.68%, Time = 0.0138 seconds
Round 6: Avalanche Effect = 0.45%, Time = 0.0066 seconds
Round 7: Avalanche Effect = 0.45%, Time = 0.0148 seconds
Round 8: Avalanche Effect = 0.45%, Time = 0.0119 seconds
Round 9: Avalanche Effect = 0.68%, Time = 0.0079 seconds
Round 10: Avalanche Effect = 0.68%, Time = 0.0153 seconds
Round 11: Avalanche Effect = 0.68%, Time = 0.0097 seconds
Round 12: Avalanche Effect = 0.45%, Time = 0.0155 seconds
Round 13: Avalanche Effect = 0.45%, Time = 0.0084 seconds
Round 14: Avalanche Effect = 0.45%, Time = 0.0149 seconds
Round 15: Avalanche Effect = 0.68%, Time = 0.0152 seconds
Round 16: Avalanche Effect = 0.68%, Time = 0.0149 seconds
```

### - Implementation

Since the goal is to compare two strings based on the difference in bits, the first thing I programmed is the **text_to_binary** function. For each character in the input text, this function converts the character to its ASCII value using ord(char), format the ASCII value as an 8-bit binary string using format (..., '08b'), and finally concatenate these binary strings into a single binary string.

```python
# Convert a string of text into a binary string representation.
def text_to_binary(text):
    return ''.join(format(ord(char), '08b') for char in text)
```

In addition, the function **calculate_avalanche_effect** gets the number of differing bits between the two binary strings, and divides it by the total number of bits in one of the strings (both have the same size). Finally, it calculates the percentage of differing bits.

```python
# Calculate the avalanche effect between two binary strings
def calculate_avalanche_effect(bin1, bin2):
    differing_bits = sum(bit1 != bit2 for bit1, bit2 in zip(bin1, bin2))
    total_bits = len(bin1)
    return (differing_bits / total_bits) * 100 # The percentage of differing bits between the two binary strings
```

Finally, I programmed the **repeated_encryption_analysis1** function, which is responsible for performing encryption numerous times on both the original and modified text. In each iteration, it calculates the percentage of bit difference between the two. The round number indicates the number of consecutive encryptions (transposition and Caesar cipher) that have been performed.

```python
# Encryption function based on Transposition and Caesar cipher
def repeated_encryption_analysis1(original_plaintext, modified_plaintext, transposition_key, caesar_shift, rounds=16):
    avalanche_results = []
    for i in range(1, rounds + 1):
        start_time = time.time()

        original_text = original_plaintext
        modified_text = modified_plaintext

        # Apply encryption i times
        for _ in range(i):
            original_transposed = transposition_encrypt(original_text, transposition_key)
            original_text = caesar_cipher_encrypt(original_transposed, caesar_shift)

            modified_transposed = transposition_encrypt(modified_text, transposition_key)
            modified_text = caesar_cipher_encrypt(modified_transposed, caesar_shift)

        # Convert ciphertexts to binary
        original_binary = text_to_binary1(original_text)
        modified_binary = text_to_binary1(modified_text)

        # Calculate the avalanche effect
        avalanche_effect = calculate_avalanche_effect(original_binary, modified_binary)
        computation_time = time.time() - start_time

        # Store the results for this round
        avalanche_results.append((i, avalanche_effect, computation_time))

        print(f"Round {i}: Avalanche Effect = {avalanche_effect:.2f}%, Time = {computation_time:.4f} seconds")

    return avalanche_results
```

In this way, by simply calling the last function with the correct parameters, we obtain the data per round for the avalanche effect.

## 2.3. Analyse the Avalanche effect

Based on the results, the avalanche effect remains consistently low across the encryption rounds. Let's analyse why this is happening and explore possible reasons for the minimal differences between the original and modified texts after encryption.

**- Reasons for Low Avalanche Effect:**

**1. Simple Encryption Schemes:**

Transposition only rearranges the order of characters but does not alter the characters themselves. As a result, small changes in the input (such as modifying one character) might not propagate significantly through the ciphertext, especially if the key and input don't interact in a way that spreads the differences.

The Caesar cipher uses a simple shift operation. While this can change individual characters, the shift operation doesn’t diffuse changes across the entire text. This results in minimal changes between the ciphertexts of similar plaintexts.

**2. Minimal Diffusion:**

A good encryption scheme exhibits high diffusion, meaning that a small change in the input should result in significant changes across the entire ciphertext. The combination of transposition and a small Caesar cipher shift provides little diffusion. Each character is shifted independently of others, so a single change in

the plaintext only affects one character at a time. Since neither cipher spreads changes across the entire text, the modified character in the plaintext does not cause major differences in the ciphertext.

To sum up, the low avalanche effect observed in this experiment indicates that the combination of transposition and Caesar cipher does not provide strong diffusion of changes. A strong avalanche effect is essential for ensuring that small modifications in the input lead to significant differences in the output. To achieve a better avalanche effect, more complex and sophisticated encryption schemes are necessary.

## 2.4. Optimize Avalanche Effect with Reasonable Computation

As already mentioned, the current encryption combination lacks the avalanche effect. Therefore, and with the aim of optimizing its effect without incurring high computational costs, I have implemented another substitution algorithm: the **Vigenère Cipher**.

The Vigenère cipher is a method of encrypting text by using a series of different Caesar ciphers based on the letters of a keyword.

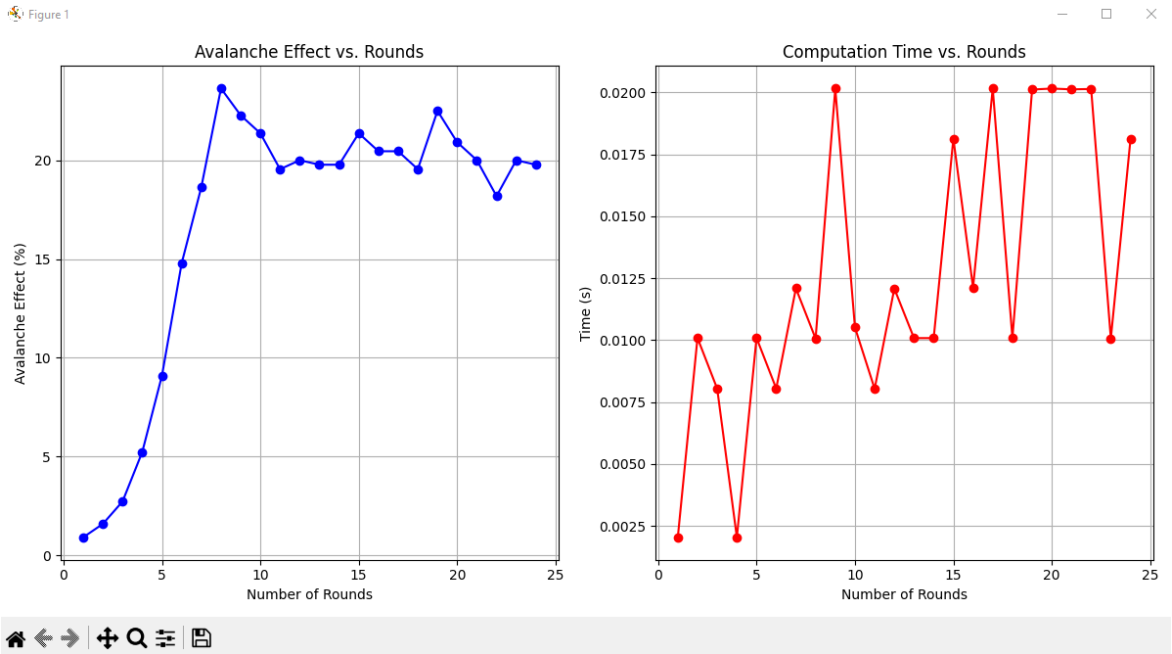
| | A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| A | A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T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

This algorithm uses each element of the key individually with its corresponding element in the plaintext. The sum of the positions in the alphabet of both characters results in the encrypted element. In this case, we will use the key: LONGKEYVIGENERE. However, since it is shorter than the plaintext, we will extend it each time with the beginning of the plaintext we are processing.

By adding this new algorithm, we achieve a new encryption combination consisting of a first phase of transposition, a second phase of substitution using the Vigenère Cipher, and a third phase with the Caesar Cipher. In this way, we achieve greater efficiency in the avalanche effect:

```
Round 1:  Avalanche Effect = 1.14%, Time = 0.0080 seconds
Round 2:  Avalanche Effect = 1.59%, Time = 0.0070 seconds
Round 3:  Avalanche Effect = 2.05%, Time = 0.0060 seconds
Round 4:  Avalanche Effect = 4.09%, Time = 0.0090 seconds
Round 5:  Avalanche Effect = 6.82%, Time = 0.0100 seconds
Round 6:  Avalanche Effect = 9.77%, Time = 0.0180 seconds
Round 7:  Avalanche Effect = 12.50%, Time = 0.0230 seconds
Round 8:  Avalanche Effect = 18.64%, Time = 0.0100 seconds
Round 9:  Avalanche Effect = 18.18%, Time = 0.0180 seconds
Round 10:  Avalanche Effect = 21.59%, Time = 0.0230 seconds
Round 11:  Avalanche Effect = 20.91%, Time = 0.0200 seconds
Round 12:  Avalanche Effect = 20.00%, Time = 0.0210 seconds
Round 13:  Avalanche Effect = 20.68%, Time = 0.0130 seconds
Round 14:  Avalanche Effect = 20.45%, Time = 0.0150 seconds
Round 15:  Avalanche Effect = 23.86%, Time = 0.0130 seconds
Round 16:  Avalanche Effect = 20.91%, Time = 0.0070 seconds
```

The next step is to analyse the relationship between the avalanche effect and the execution time with each round. This way, we can optimize the encryption by selecting the most efficient number of rounds:




The computational cost will depend on many factors, such as the type of device being used to run the program or its processing capacity. In my case, as seen in the graphs, the computational time is irregular, although the graph tends to increase with each round. On the other hand, the avalanche effect grows significantly with each round, up to a certain point. The leveling off at around 20% is a result of the limited diffusion capabilities of the transposition and Caesar ciphers, which restrict the degree of randomness and change propagation across multiple rounds.

Analysing both graphs, it can be concluded that running 15 rounds proves to be the most optimal in terms of the avalanche effect versus computational time.

```
Optimal number of rounds based on the balance: 15
```

## - Implementation

For the implementation of the Vigenère Cipher, I have developed the function **vigenere_cipher_encrypt**. This function works with the plaintext2 without spaces and in uppercase. If the plaintext length exceeds the key length, the key is extended by appending characters from the beginning of plaintext2. This ensures that the key matches the length of the plaintext. Then, for each character in the plaintext if it is a letter, the function calculates a shift based on the current character in the key. Depending on whether the letter is uppercase or lowercase, it is shifted by the corresponding value, wrapping around the alphabet using modulo 26.

The function uses 'A' to calculate the shift for the Vigenère cipher because 'A' represents the starting point of the alphabet in ASCII. By subtracting the ASCII value of 'A', the function effectively maps the letters of the alphabet to a 0-25 range (where 'A' becomes 0, 'B' becomes 1, and so on).

The encrypted characters are collected into the ciphertext list, which is finally joined into a single string and returned.

```python
# Vigenère cipher encryption function
def vigenere_cipher_encrypt(plaintext, key):
    key = key.upper()
    ciphertext = []
    key_index = 0
    plaintext2 = plaintext.upper().replace(' ', '')

    key_length = len(key)
    if len(plaintext) > key_length: # Fill the key with text from the plaintext.
        padding_length = len(plaintext) - key_length
        padding = plaintext2[:padding_length]
        key = key + padding
    else:
        key = key
    key = key.upper()

    for char in plaintext:
        if char.isalpha():
            shift = ord(key[key_index % len(key)]) - ord('A')  # Determine the shift based on the key
            if char.isupper():
                ciphertext.append(chr((ord(char) - ord('A') + shift) % 26 + ord('A')))
            else:
                ciphertext.append(chr((ord(char) - ord('a') + shift) % 26 + ord('a')))
            key_index += 1  # Move to the next character in the key
        else:
            ciphertext.append(char)  # Do not encrypt spaces or special characters
    return ''.join(ciphertext)
```

Once we have the transposition algorithm, the Vigenère cipher, and the Caesar cipher, I developed the function **combined_encryption**. This function will apply the three encryption algorithms to the text provided as a parameter.

```python
# Combined encryption function (Transposition -> Vigenère -> Caesar)
def combined_encryption(plaintext, transposition_key, vigenere_key, caesar_shift, prev_ciphertext=None):
    # Step 1: Apply transposition
    transposed_text = transposition_encrypt(plaintext, transposition_key)

    # Step 2: Apply Vigenère
    vigenere_text = vigenere_cipher_encrypt(transposed_text, vigenere_key)

    # Step 3: Apply Caesar
    final_ciphertext = caesar_cipher_encrypt(vigenere_text, caesar_shift)

    return final_ciphertext
```

Finally, the function **repeated_encryption_analysis2** has a role like that of the function repeated_encryption_analysis1, except that it adds the Vigenère cipher to the encryption process. In this way, it calculates the avalanche effect on the combined encryption (Transposition -> Vigenère -> Caesar).

```python
# Encryption function based on Transposition, Vigènere and Caesar cipher
def repeated_encryption_analysis2(original_plaintext, modified_plaintext, transposition_key, caesar_shift, vigenere_key, rounds=16):
    avalanche_results = []
    for i in range(1, rounds + 1):
        start_time = time.time()

        original_text = original_plaintext
        modified_text = modified_plaintext

        # Apply encryption i times
        for _ in range(i):
            original_text = combined_encryption(original_text, transposition_key, vigenere_key, caesar_shift)

            modified_text = combined_encryption(modified_text, transposition_key, vigenere_key, caesar_shift)

        # Convert the ciphertexts to binary
        original_binary = text_to_binary1(original_text)
        modified_binary = text_to_binary1(modified_text)

        # Calculate the avalanche effect
        avalanche_effect = calculate_avalanche_effect(original_binary, modified_binary)
        computation_time = time.time() - start_time

        # Store the results
        avalanche_results.append((i, avalanche_effect, computation_time))

        print(f"Round {i}:  Avalanche Effect = {avalanche_effect:.2f}%, Time = {computation_time:.4f} seconds")

    return avalanche_results
```

Additionally, after calling this last function, the obtained data is used to construct the **graphs**. First, the obtained data is separated into rounds, avalanche effect percentages, and computation times. We create the Avalanche Effect vs. Rounds plot using 'plt' (import matplotlib.pyplot as plt). For this, the matplotlib library needs to be installed. For the other plot, we do something similar, and finally, we calculate the optimal number of rounds based on the balance between the avalanche effect and computation time:

- **max():** Finds the index of the maximum value in the list of ratios of avalanche effect to computation time.
- **lambda x: avalanche_effects[x] / (computation_times[x]+0.00000001):** Defines the ratio of avalanche effect to computation time, with a small constant to avoid division by zero.
- **optimal_round + 1**: Adjusts the index to match the round number (1-based index).

```python
# Separate the data for plotting.
rounds = [result[0] for result in avalanche_results]
avalanche_effects = [result[1] for result in avalanche_results]
computation_times = [result[2] for result in avalanche_results]

# Plot Avalanche Effect vs. Rounds
plt.figure(figsize=(12, 6))
plt.subplot(1, 2, 1)
plt.plot(rounds, avalanche_effects, marker='o', color='b', label='Avalanche Effect (%)')
plt.title('Avalanche Effect vs. Rounds')
plt.xlabel('Number of Rounds')
plt.ylabel('Avalanche Effect (%)')
plt.grid(True)

# Plot Computation Time vs. Rounds
plt.subplot(1, 2, 2)
plt.plot(rounds, computation_times, marker='o', color='r', label='Computation Time (s)')
plt.title('Computation Time vs. Rounds')
plt.xlabel('Number of Rounds')
plt.ylabel('Time (s)')
plt.grid(True)

plt.tight_layout()
plt.show()

# Determine the optimal number of rounds based on a balance between avalanche effect and computation time
optimal_round = max(range(len(avalanche_effects)), key=lambda x: avalanche_effects[x] / (computation_times[x]+0.00000001))

print(f"Optimal number of rounds based on the balance: {optimal_round + 1}")
```

## 2.5. Enhance Security with Block Ciphers

Transposition ciphers and substitution ciphers both have inherent weaknesses that limit their security:

**1. Transposition Ciphers:**

- Susceptibility to pattern recognition: Since transposition only reorders characters without altering them, patterns from the original plaintext (like repeated characters or common digraphs) may still be visible in the ciphertext, making them easier to analyse.

- Lack of diffusion: Changes in the plaintext (such as modifying one letter) only affect the position of characters, not their values. This limits how much a small change in the plaintext impacts the ciphertext.

2. **Substitution Ciphers:**

- Susceptibility to frequency analysis: Since each letter is replaced consistently with another, the frequency of letters in the ciphertext reflects that of the plaintext. Common letters (like 'E' in English) can be easily deduced, making the cipher vulnerable to statistical attacks.

- Limited confusion and diffusion: Substitution ciphers typically replace characters independently, so changing a character in the plaintext only affects the corresponding character in the ciphertext, limiting how much change propagates.

Applied to our case, the fixed shift nature of Caesar Cipher extremely limits its ability to propagate changes. In addition, while an improvement over Caesar, the repetitive pattern and lack of significant change propagation across the text of Vigenère Cipher, also limits its effectiveness.

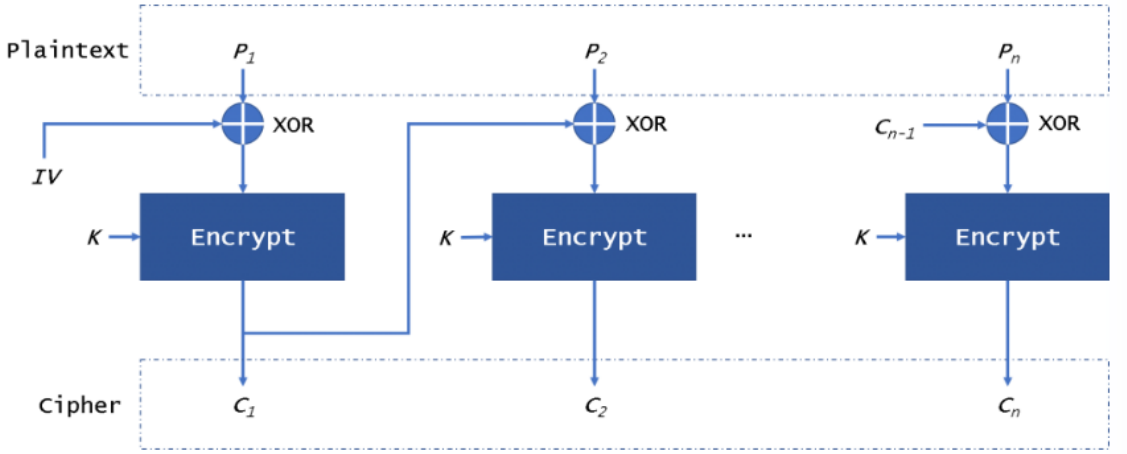
Overall, both ciphers lack the necessary confusion and diffusion that modern encryption algorithms provide to ensure security.

To achieve a better avalanche effect, enhanced security, and overall robustness, it is better to implement block cipher encryption. The block cipher, such as **CBC** mode, is widely recognized for its strength in cryptographic security due to its ability to handle fixed-size blocks of data. By using block ciphers, I can ensure that small changes in the plaintext will result in significant and unpredictable changes in the ciphertext, which is a key aspect of the avalanche effect.

- **BLOCK CIPHER:**

The decision to implement **Cipher Block Chaining (CBC)** mode on top of the existing transposition and Caesar cipher was made to enhance the security of these classical ciphers without introducing the complexity of modern block ciphers like AES or DES. The transposition and Caesar ciphers, while historically important, have significant weaknesses, including limited diffusion and vulnerability to pattern recognition. By adding CBC mode, we can address these weaknesses and improve overall encryption strength without replacing the existing methods.

CBC mode was chosen because it introduces **inter-block chaining**, where each plaintext block is XORed with the previous ciphertext block before encryption. This chaining ensures that even identical blocks of plaintext result in different ciphertexts, thus overcoming the limitations of classical ciphers. In contrast, using other modes like **Electronic Codebook (ECB)** would not provide this level of protection, as it treats each block independently, making the encryption more susceptible to attacks based on repeated patterns. CBC's ability to spread changes across blocks provides the enhanced diffusion that transposition and Caesar ciphers lack on their own.

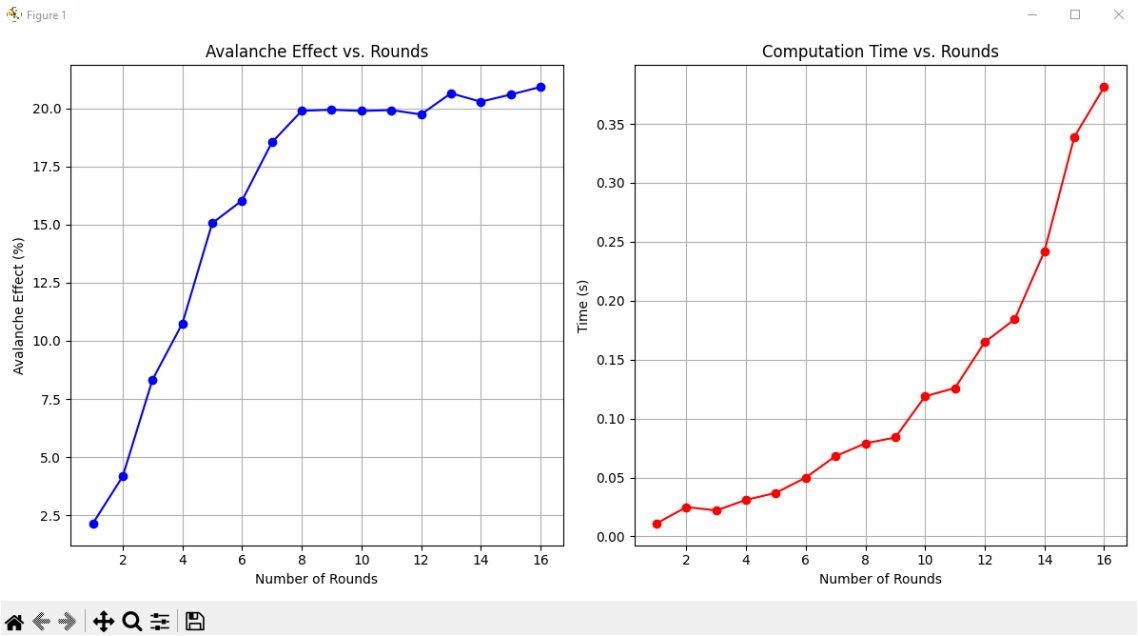



By applying CBC mode, we also significantly improve the **avalanche effect**. A minor change in the plaintext, such as altering a single character, leads to changes that propagate through all subsequent ciphertext blocks. This effect is essential for cryptographic security because it makes it difficult for attackers to deduce plaintext-ciphertext relationships. Without CBC, transposition and Caesar ciphers would show limited changes in the ciphertext, making them more vulnerable to cryptanalysis. The addition of CBC ensures that small plaintext modifications lead to significant differences in the entire ciphertext, strengthening the overall encryption process.

Furthermore, the implementation of a **dynamic Initialization Vector (IV)** enhances security even more. By using the last block of ciphertext as the new IV for the next round, we ensure that each round of encryption is unique, even if the same plaintext is used. This

approach prevents IV reuse, a common vulnerability in encryption schemes, and increases the randomness of the output, providing additional protection against attacks.

```
Round 1: Avalanche Effect = 2.14%, Time = 0.0110 seconds
Round 2: Avalanche Effect = 4.17%, Time = 0.0250 seconds
Round 3: Avalanche Effect = 8.33%, Time = 0.0220 seconds
Round 4: Avalanche Effect = 10.75%, Time = 0.0310 seconds
Round 5: Avalanche Effect = 15.07%, Time = 0.0370 seconds
Round 6: Avalanche Effect = 16.04%, Time = 0.0499 seconds
Round 7: Avalanche Effect = 18.54%, Time = 0.0682 seconds
Round 8: Avalanche Effect = 19.90%, Time = 0.0790 seconds
Round 9: Avalanche Effect = 19.95%, Time = 0.0840 seconds
Round 10: Avalanche Effect = 19.90%, Time = 0.1189 seconds
Round 11: Avalanche Effect = 19.93%, Time = 0.1259 seconds
Round 12: Avalanche Effect = 19.75%, Time = 0.1649 seconds
Round 13: Avalanche Effect = 20.66%, Time = 0.1842 seconds
Round 14: Avalanche Effect = 20.30%, Time = 0.2420 seconds
Round 15: Avalanche Effect = 20.60%, Time = 0.3390 seconds
Round 16: Avalanche Effect = 20.93%, Time = 0.3814 seconds
```




In this case, we observe how the avalanche effect grows exponentially up to a certain point (as well as with Vigènere cipher), as the number of rounds increases. However, the same happens with computational time.

## - Implementation

First, I programmed the **xor_blocks** function. It applies a bitwise XOR operation between two equal-length strings, returning a new string where each character results from the XOR of corresponding characters from the input strings.

```python
# XOR function to chain blocks
def xor_blocks(block1, block2):
    return ''.join(chr(ord(c1) ^ ord(c2)) for c1, c2 in zip(block1, block2))
```

The **combined_encryption_cbc** function applies CBC mode to encrypt plaintext using a combination of transposition and Caesar cipher. It splits the plaintext into blocks, XORs each block with the previous ciphertext block (starting with the IV), and then encrypts the result with the transposition and Caesar ciphers. Each block's ciphertext is chained to the previous one, ensuring that changes in plaintext affect the entire ciphertext.

```python
# Combined encryption with CBC applied to transposition + Caesar cipher
def combined_encryption_cbc(plaintext, transposition_key, caesar_shift, iv):
    block_size = len(iv)
    ciphertext = []

    # Break plaintext into blocks
    blocks = [plaintext[i:i+block_size] for i in range(0, len(plaintext), block_size)]

    prev_block = iv  # Use IV as the first "previous block"

    for block in blocks:
        # Apply XOR with the previous block (CBC mechanism)
        block = xor_blocks(block.ljust(block_size), prev_block)

        # Apply the combined transposition + Caesar cipher
        transposed_block = transposition_encrypt(block, transposition_key)
        encrypted_block = caesar_cipher_encrypt(transposed_block, caesar_shift)

        # Update the previous block with the current ciphertext block
        prev_block = encrypted_block
        ciphertext.append(encrypted_block)

    return ''.join(ciphertext)
```

And finally, I developed the **repeated_encryption_analysis3** function, which, like functions 1 and 2, collects data on the avalanche effect. In this case, the data comes from the encryption process that combines transposition and Caesar cipher in CBC mode.

```python
# Function for repeated encryption analysis with CBC
def repeated_encryption_analysis3(original_plaintext, modified_plaintext, transposition_key, caesar_shift, iv, rounds=16):
    avalanche_results = []
    original_text = original_plaintext
    modified_text = modified_plaintext
    for i in range(1, rounds + 1):
        start_time = time.time()

        # Encrypt with CBC
        original_text = combined_encryption_cbc(original_text, transposition_key, caesar_shift, iv)
        modified_text = combined_encryption_cbc(modified_text, transposition_key, caesar_shift, iv)

        # Calculate avalanche effect
        original_binary = text_to_binary1(original_text)
        modified_binary = text_to_binary1(modified_text)

        avalanche_effect = calculate_avalanche_effect(original_binary, modified_binary)
        computation_time = time.time() - start_time

        # Store results for each round
        avalanche_results.append((i, avalanche_effect, computation_time))
        print(f"Round {i}: Avalanche Effect = {avalanche_effect:.2f}%, Time = {computation_time:.4f} seconds")

    return avalanche_results
```

- **AES IN CBC MODE:**

Applying CBC mode to **AES** instead of transposition and substitution ciphers was driven by the need for a **modern, secure encryption method** that can withstand contemporary cryptographic challenges.

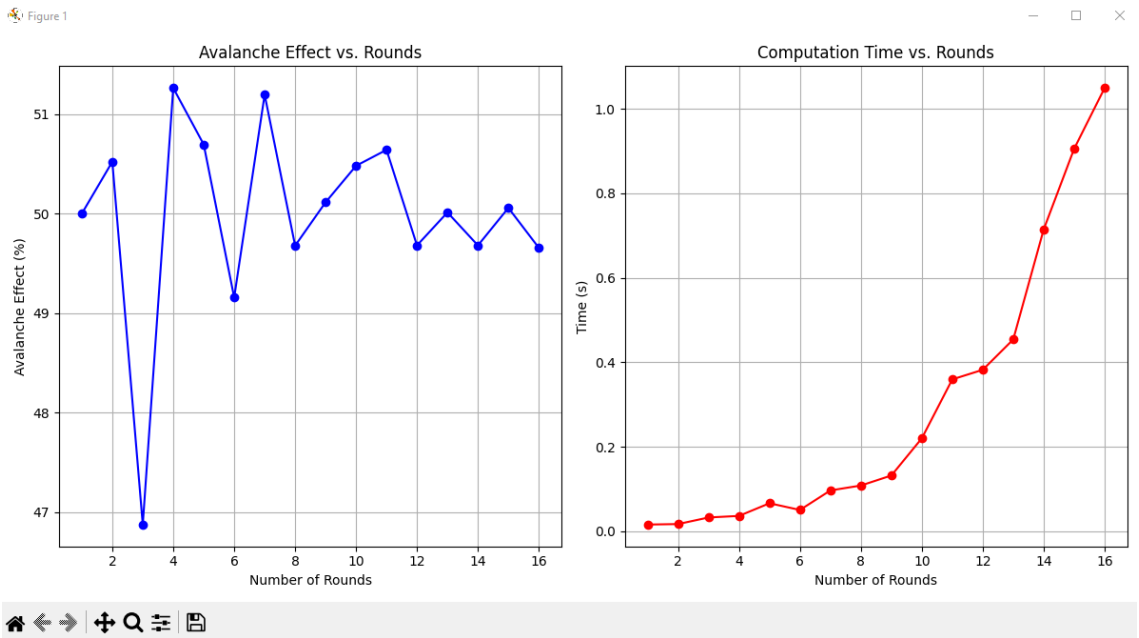
AES offers strong **confusion** and **diffusion**, where each round of encryption significantly alters the relationship between the plaintext, the key, and the resulting ciphertext. Unlike transposition and substitution ciphers, which perform simple rearrangements and letter shifts, AES operates on binary data at the block level and applies multiple rounds of sophisticated transformations.

By applying CBC mode to AES, we take advantage of both the **strength of AES's block encryption** and the **chaining mechanism** provided by CBC. This ensures that each block of plaintext is not only encrypted using AES's complex internal transformations but also influenced by the data from the previous block, amplifying the **avalanche effect**. In contrast, transposition and substitution ciphers, even when combined with CBC, do not provide the same level of diffusion or protection against cryptanalysis because their operations are much simpler and less secure. AES is designed to resist a wide range of attacks, making it far more resilient than classical ciphers.

When testing its effectiveness, I observed that the avalanche effect has increased significantly.

```
Round 1: Avalanche Effect = 50.00%, Time = 0.0156 seconds
Round 2: Avalanche Effect = 50.52%, Time = 0.0168 seconds
Round 3: Avalanche Effect = 46.88%, Time = 0.0324 seconds
Round 4: Avalanche Effect = 51.26%, Time = 0.0361 seconds
Round 5: Avalanche Effect = 50.69%, Time = 0.0661 seconds
Round 6: Avalanche Effect = 49.16%, Time = 0.0502 seconds
Round 7: Avalanche Effect = 51.19%, Time = 0.0965 seconds
Round 8: Avalanche Effect = 49.68%, Time = 0.1081 seconds
Round 9: Avalanche Effect = 50.11%, Time = 0.1318 seconds
Round 10: Avalanche Effect = 50.48%, Time = 0.2198 seconds
Round 11: Avalanche Effect = 50.64%, Time = 0.3594 seconds
Round 12: Avalanche Effect = 49.68%, Time = 0.3821 seconds
Round 13: Avalanche Effect = 50.01%, Time = 0.4539 seconds
Round 14: Avalanche Effect = 49.68%, Time = 0.7145 seconds
Round 15: Avalanche Effect = 50.06%, Time = 0.9056 seconds
Round 16: Avalanche Effect = 49.66%, Time = 1.0494 seconds
```

By conducting a graphical analysis, we can also observe that the computation time increases as the number of rounds increases.



In this way, we can conclude that AES in CBC mode significantly improves the effectiveness and security of simple encryption methods like transposition and Caesar cipher. Its ability to produce a stronger avalanche effect ensures that even minor changes in the input drastically alter the output, making it a much more robust and secure encryption mechanism. This highlights its superiority for protecting sensitive data compared to basic cryptographic techniques.

## - Implementation

When encrypting data with AES in CBC mode, the resulting ciphertext is binary and may contain non-printable characters. This can create challenges when handling and comparing encrypted data, especially if you want to analyse how changes propagate through multiple encryption rounds.

To fix this, I encode the binary ciphertext into a Base64 string. These strings are easier to compare and analyse. When checking the avalanche effect, having ciphertext consistently encoded allows for straightforward binary-to-text conversion, making it simpler to compare the effects of modifications in the plaintext.

First, I developed the **aes_encrypt** function, which relies on the use of the **crypto** library. This function initializes a new AES cipher object with the specified key, CBC (Cipher Block Chaining) mode, and initialization vector (IV). Then, the pad function adds extra bytes to the plaintext to ensure it meets the requirement that the plaintext needs to be a multiple of the block size (16 bytes for AES).

The padded plaintext is processed by the AES cipher in CBC mode, producing ciphertext. The result is a binary string representing the encrypted data. And finally, after encryption, the ciphertext is binary and may contain non-printable characters. To ensure the ciphertext is in a consistent and readable format, it is encoded using Base64. The decode('utf-8') method converts this Base64-encoded byte string into a standard UTF-8 string for easier handling and transmission.

```
# AES encryption with CBC mode
def aes_encrypt(plaintext, key, iv):
    cipher = AES.new(key, AES.MODE_CBC, iv)
    padded_plaintext = pad(plaintext.encode(), AES.block_size)
    ciphertext = cipher.encrypt(padded_plaintext)
    return base64.b64encode(ciphertext).decode('utf-8')
```

In addition, I had to develop another function, **text_to_binary2**, to properly transform the Base64 encoded text into bytes and finally into a binary string.

```
def text_to_binary2(text):
    # Convert base64 encoded text to bytes, then to binary string
    return ''.join(format(byte, '08b') for byte in base64.b64decode(text))
```

Finally, I developed the **repeated_encryption_analysis4** function, which, like the previous functions, collects data on the avalanche effect. In this case, the data comes from the encryption process that performs AES encryption in CBC mode, based on the previously developed functions.

```python
# AES encryption analysis with CBC
def repeated_encryption_analysis4(original_plaintext, modified_plaintext, key, rounds=16):
    avalanche_results = []
    prev_original_text = original_plaintext
    prev_modified_text = modified_plaintext
    for i in range(1, rounds + 1):
        start_time = time.time()
        iv = get_random_bytes(16) # Update the Random Initialization Vector (IV) with each iteration
        # Encrypt original and modified plaintext
        original_ciphertext = aes_encrypt(prev_original_text, key, iv)
        modified_ciphertext = aes_encrypt(prev_modified_text, key, iv)
        # Convert to binary
        original_binary = text_to_binary2(original_ciphertext)
        modified_binary = text_to_binary2(modified_ciphertext)
        # Calculate avalanche effect
        avalanche_effect = calculate_avalanche_effect(original_binary, modified_binary)
        computation_time = time.time() - start_time
        # Reload
        prev_original_text = original_ciphertext
        prev_modified_text = modified_ciphertext

        # Store results for each round
        avalanche_results.append((i, avalanche_effect, computation_time))
        print(f"Round {i}: Avalanche Effect = {avalanche_effect:.2f}%, Time = {computation_time:.4f} seconds")

    return avalanche_results
```

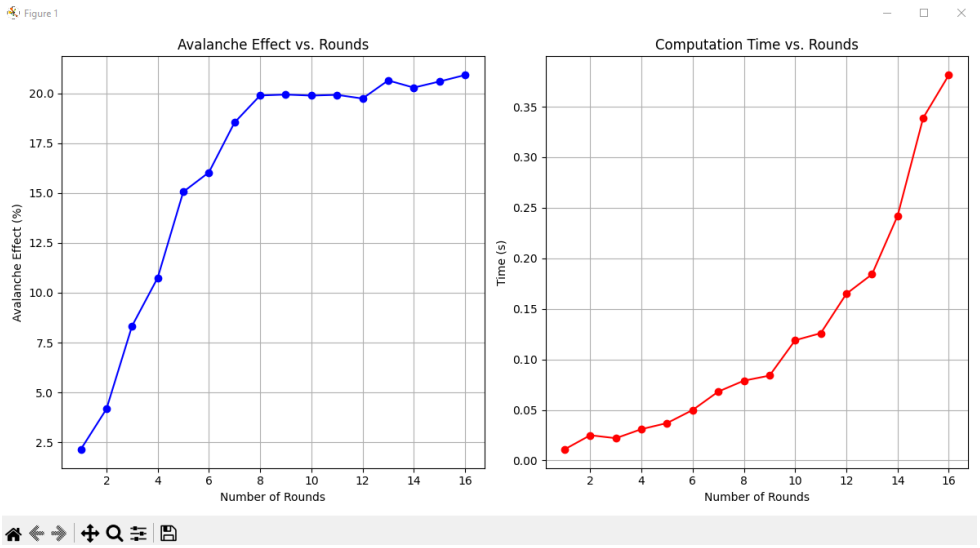
# 3. Discussion

The results obtained in this project highlight the significant differences between classical and modern cryptographic techniques, specifically in their ability to provide security through **diffusion** and **confusion**. Classical ciphers, such as transposition and substitution, while historically important, show clear vulnerabilities in modern contexts. Even though substitution methods like **Vigenère** somewhat improve the avalanche effect, these methods exhibit limited resistance to common cryptanalysis techniques such as **frequency analysis** and **pattern recognition**. Additionally, the lack of diffusion in transposition ciphers, where character positions change but their values do not, further exposes their weaknesses, particularly when small modifications in plaintext produce minimal changes in ciphertext.

```
Round 1: Avalanche Effect = 0.45%, Time = 0.0083 seconds
Round 2: Avalanche Effect = 0.45%, Time = 0.0055 seconds
Round 3: Avalanche Effect = 0.45%, Time = 0.0121 seconds
Round 4: Avalanche Effect = 0.45%, Time = 0.0078 seconds
Round 5: Avalanche Effect = 0.68%, Time = 0.0138 seconds
Round 6: Avalanche Effect = 0.45%, Time = 0.0066 seconds
Round 7: Avalanche Effect = 0.45%, Time = 0.0148 seconds
Round 8: Avalanche Effect = 0.45%, Time = 0.0119 seconds
Round 9: Avalanche Effect = 0.68%, Time = 0.0079 seconds
Round 10: Avalanche Effect = 0.68%, Time = 0.0153 seconds
Round 11: Avalanche Effect = 0.68%, Time = 0.0097 seconds
Round 12: Avalanche Effect = 0.45%, Time = 0.0155 seconds
Round 13: Avalanche Effect = 0.45%, Time = 0.0084 seconds
Round 14: Avalanche Effect = 0.45%, Time = 0.0149 seconds
Round 15: Avalanche Effect = 0.68%, Time = 0.0152 seconds
Round 16: Avalanche Effect = 0.68%, Time = 0.0149 seconds
```

*Transposition and Caesar cipher do not produce avalanche effect*

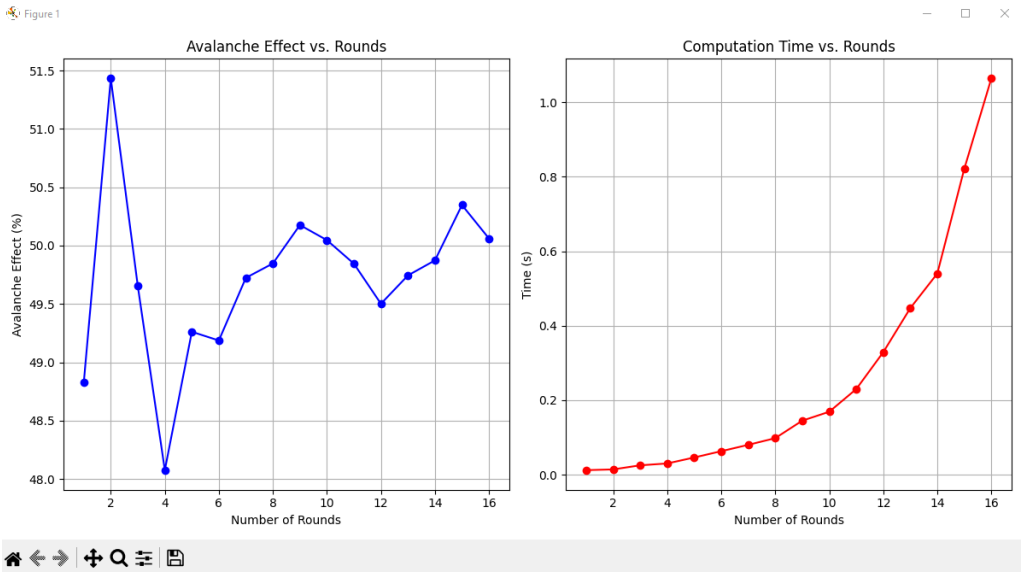
Integrating **CBC mode** with simple encryption methods like **transposition and substitution** significantly enhances their security by introducing block chaining. This method increases the avalanche effect, thereby improving resistance to cryptanalysis and making the encryption more secure against attacks.




Despite being a significant improvement, computational time is also affected accordingly. Additionally, the avalanche effect increases up to a maximum of around 22%, indicating room for further enhancement.

The implementation of **AES in CBC mode** demonstrated a marked improvement in both security and effectiveness. AES, with its complex key structure and multiple rounds of substitution and permutation, provides far greater resistance to cryptographic attacks. This feature significantly boosts the **avalanche effect**, where a minor change in the plaintext results in a completely different ciphertext.

The results clearly show that AES in CBC mode produces a much higher avalanche effect compared to classical methods, confirming its superiority in ensuring data security.




*AES on CBC mode considerably increases avalanche effect*

Compared to the previous method, AES achieves a 50% efficiency with nearly the same computational time.

# 4. Conclusion

In conclusion, the integration of AES in CBC mode significantly enhances both the security and the avalanche effect, compared to simpler encryption methods like transposition and Caesar cipher (even in CBC mode). While these basic methods do not inherently produce a strong avalanche effect, adding AES with CBC ensures that even small changes in the plaintext result in substantial differences in the ciphertext, thereby strengthening the encryption. This combination leads to more robust data protection and improved resistance to cryptographic attacks.

# References


Lectures 03, 04, 07 of the course material.

- Transposition and Caesar Cipher:

https://en.wikipedia.org/wiki/Transposition_cipher

https://en.wikipedia.org/wiki/Caesar_cipher

- Vigènere Cipher:

https://en.wikipedia.org/wiki/Vigen%C3%A8re_cipher

https://www.dcode.fr/cifrado-vigenere

- AES and CBC mode:

https://www.highgo.ca/2019/08/08/the-difference-in-five-modes-in-the-aes-encryption-algorithm/

https://onboardbase.com/blog/aes-encryption-decryption/