ElGamal Random Number Reuse

6. What is the problem happens if sender uses same random number to encrypt two plaintexts M1 and M2 using Elgamal cryptosystem?

If the sender uses the same random number r to encrypt two plaintexts M_1 and M_2 in the ElGamal cryptosystem, it can lead to a serious security issue where an attacker can deduce a relationship between the ciphertexts and potentially recover information about the plaintexts. Here's how:

ElGamal Encryption Recap:

Given the public key (p, g, h) where:

- p: a large prime number,
- g: a generator of a cyclic group modulo p,
- $h = g^x \mod p$ (with x being the private key),

To encrypt a plaintext M, the sender:

1. Picks a random number r.
2. Computes the ciphertext (C_1, C_2):
   - $C_1 = g^r \mod p$,
   - $C_2 = M \cdot h^r \mod p$.

Problem When r is Reused:

If the same r is used to encrypt M_1 and M_2, the resulting ciphertexts are:

- $(C_{1,1}, C_{2,1}) = (g^r, M_1 \cdot h^r)$,
- $(C_{1,2}, C_{2,2}) = (g^r, M_2 \cdot h^r)$.

Since $C_1 = g^r$ is the same for both ciphertexts, an attacker can compute:

$$\frac{C_{2,1}}{C_{2,2}} = \frac{M_1 \cdot h^r}{M_2 \cdot h^r} = \frac{M_1}{M_2} \mod p.$$

This reveals the ratio of the two plaintexts, $\frac{M_1}{M_2} \mod p$. With additional information or known plaintext attacks, the attacker might be able to determine both M_1 and M_2.

Consequences:

1. **Loss of Semantic Security**: The reuse of r violates the principle of randomness in encryption, exposing relationships between plaintexts.

2. **Plaintext Recovery**: If an attacker knows M_1 (e.g., through a chosen plaintext attack), they can easily compute M_2 as $M_2 = \frac{C_{2,2}}{C_{2,1}/M_1} \mod p$.

Solution:

To prevent this issue:

1. Ensure r is chosen randomly and uniquely for every encryption operation.
2. Use a secure random number generator to generate r.

Randomness in r is critical to maintaining the security of the ElGamal cryptosystem.

7. Show that if an user choses a prime N as the modulus in the RSA system, it can be trivially broken.

If a user chooses a **prime number** N as the modulus in an RSA system, the security of the system is entirely compromised. Here's why:

Recap of RSA Cryptosystem:

1. RSA relies on the difficulty of factoring a large modulus N, which is usually the product of two large primes:

$$N = p \cdot q$$

   where p and q are large prime numbers.
2. The Euler totient function $\phi(N)$ is critical for generating the private key:

$$\phi(N) = (p-1)(q-1).$$

3. The public key is (N, e), and the private key d satisfies:

$$e \cdot d \equiv 1 \pmod{\phi(N)}.$$

Problem with N as a Prime:

If N is chosen to be a prime number:

1. The totient function becomes:

$$\phi(N) = N - 1$$

   because for a prime N, all integers less than N are coprime to N.

2. An attacker can trivially compute $\phi(N)$ as $N - 1$ since N is known (it is the public modulus).

3. Once $\phi(N)$ is known, the attacker can compute the private key d by solving the congruence:

$$d \equiv e^{-1} \pmod{\phi(N)}$$

Using the Extended Euclidean Algorithm, d can be calculated efficiently.

Demonstration:

Suppose:

- $N = 17$ (a prime),
- $e = 3$ (public exponent).

1. Public key: $(N, e) = (17, 3)$.
2. Compute $\phi(N) = N - 1 = 16$.
3. Solve for d such that:

$$e \cdot d \equiv 1 \pmod{\phi(N)}.$$

   Here:

$$3 \cdot d \equiv 1 \pmod{16}.$$

   Using the Extended Euclidean Algorithm, $d = 11$.

4. The private key is $(N, d) = (17, 11)$.

Exploitation:

An attacker with N and e can quickly derive d, effectively breaking the RSA system.

Conclusion:

Using a **prime modulus** N in RSA eliminates the core security feature of the system—the difficulty of factoring a composite number. To maintain security, N must always be the product of two large, distinct primes.

8. Alice wants to send an encrypted message to Bob using RSA, but doesn't know his public Key. So, she sends Bob an email asking for the key. Bob replies with his RSA public key (e, N). However, the active adversary intercepts the message and changes one bit in e from 0 to 1, so Alice receives an email claiming that Bobs public key is (e0, N), where e0 differs from e in one bit. Alice encrypts m with this key and sends it to Bob. Of course, Bob cannot decrypt, since the message was encrypted with the wrong key. So he resends his key and asks Alice to send the encrypted message again, which she does. The adversary eavesdrops to the whole communication without interfering further. Describe how he can recover m.

The described scenario introduces a vulnerability in the RSA encryption process, allowing the adversary to recover the plaintext message m due to the differences in the exponents used during encryption.

Here's how the attack unfolds:

Key Points:

1. **Original Setup**:
   - Bob's correct public key is (e, N).
   - The adversary changes e to e_0, which differs from e by one bit, and sends (e_0, N) to Alice.
   - Alice encrypts m using (e_0, N):

     $$c_0 = m^{e_0} \mod N$$

     and sends c_0 to Bob.
2. **Second Communication**:
   - Bob realizes he cannot decrypt c_0, as it was encrypted with the wrong key e_0.
   - He sends his correct public key (e, N) again.
   - Alice re-encrypts m using (e, N):

     $$c = m^e \mod N$$

     and sends c to Bob.
3. **Adversary's Goal**: The adversary intercepts c_0 and c and now has:
   - $c_0 = m^{e_0} \mod N$,
   - $c = m^e \mod N$,
   - e_0 and e, which differ by one bit.

Exploitation:

The adversary knows that e_0 and e differ by one bit. Specifically:

$$e_0 = e \oplus b,$$

where b is a bitmask with a single bit set to 1 at the position where e_0 and e differ.

The adversary uses the following mathematical relation to recover m:

$$m^{e_0} = m^{e \oplus b} = m^e \cdot m^b \mod N.$$

1. From the intercepted ciphertexts:
   - $c_0 = m^{e_0} \mod N$,
   - $c = m^e \mod N$.
2. Compute the ratio:

   $$\frac{c_0}{c} = \frac{m^{e_0}}{m^e} = m^b \mod N.$$

3. Since b has only one bit set, b is small. The adversary can compute m using brute-force techniques:
   - Try all possible values for b (typically very few, e.g., $\log_2(e)$ attempts).
   - For each b, compute:

$$m = \sqrt[b]{\frac{c_0}{c}} \mod N.$$

Conclusion:

The adversary exploits the bitwise difference in the exponents to deduce the plaintext m through a simple computation involving the intercepted ciphertexts c_0 and c. This attack demonstrates the importance of verifying the integrity of public keys in cryptographic communications to prevent tampering.

9. Alice has decided to use RSA for encryption and has generated two large primes p and q and computed N = pq. She has also chosen encryption key eA = 3 and computed her private key dA. When her friend Bob hears about this, he also wants to use RSA. Alice assists him by choosing for him eB = 5 and computing dB using the same N. Alice gives Bob her keys (N; eB) and dB. The next day their common friend Charlie sends message m encrypted to both Alice and Bob, using their respective encryption keys. However, the adversary Deborah eavesdrops and gets hold of the two ciphertext CA and CB. Deborah also notices that Alice and Bob use the same N. Show how Deborah can recover m. You may assume that gcd(m; N) = 1.
Does Deborahs attack generalize to other values of eA and eB than 3 and 5?

This situation describes a **common modulus attack** in RSA. If two users share the same modulus N but use different public exponents e_A and e_B, an adversary can recover the plaintext m if they intercept the ciphertexts C_A and C_B. Here's how Deborah can exploit this setup:

Key Observations:

1. **Encryption for Alice and Bob**:
   - Alice's public key is $(N, e_A) = (N, 3)$.
   - Bob's public key is $(N, e_B) = (N, 5)$.
   - Charlie encrypts the message m to both Alice and Bob:

$$C_A = m^{e_A} \mod N, \quad C_B = m^{e_B} \mod N.$$

2. Deborah intercepts C_A and C_B and notices that the two ciphertexts were generated using the same modulus N.
3. The goal is to recover m, given C_A, C_B, $e_A = 3$, $e_B = 5$, and N.
4. By assumption, $\gcd(m, N) = 1$, so m is invertible modulo N.

Exploitation:

Deborah can use the **Extended Euclidean Algorithm** to recover m. The key idea is to compute m by combining the two ciphertexts C_A and C_B.

1. Use the Extended Euclidean Algorithm to find integers a and b such that:

$$a \cdot e_A + b \cdot e_B = 1.$$

   For $e_A = 3$ and $e_B = 5$:

$$a = -2, \quad b = 1 \quad (\text{since } -2 \cdot 3 + 1 \cdot 5 = 1).$$

2. Express m using C_A and C_B:

$$m = C_A^a \cdot C_B^b \mod N.$$

3. Substitute $a = -2$ and $b = 1$:

$$m = (C_A^{-2} \mod N) \cdot (C_B^1 \mod N) \mod N.$$

4. To compute $C_A^{-2} \mod N$, find the modular inverse of C_A (denoted as $C_A^{-1} \mod N$) and then square it:

$$C_A^{-2} \equiv (C_A^{-1})^2 \mod N.$$

5. Multiply $(C_A^{-2} \mod N)$ and $(C_B \mod N)$:

$$m = (C_A^{-1})^2 \cdot C_B \mod N.$$

Deborah now has m, the original plaintext.

Generalization:

This attack works for **any pair of public exponents** e_A **and** e_B as long as:

1. $\gcd(e_A, e_B) = 1$ (this ensures the Extended Euclidean Algorithm can find a and b).
2. $\gcd(m, N) = 1$ (ensuring m is invertible modulo N).

For example, the attack generalizes to any $e_A = 7$ and $e_B = 11$, or any other pair of coprime exponents.

Conclusion:

The **common modulus attack** highlights why RSA must use a distinct modulus N for each user or why the same modulus should never be shared with different public exponents. Sharing N undermines the security of RSA, as demonstrated by this attack.

11. What is differential analysis and Linear analysis of DES. Show with example.

Differential and linear analysis are cryptanalytic techniques used to attack block ciphers, such as the Data Encryption Standard (DES). These methods exploit statistical patterns in how plaintexts and ciphertexts relate to each other, bypassing brute force.

---

1. Differential Analysis

Overview:

- **Differential Cryptanalysis** studies how differences in plaintext pairs affect the differences in the resulting ciphertext pairs.
- It relies on observing patterns in how specific differences propagate through the cipher.
- Introduced as a general attack against Feistel networks like DES, it is particularly effective if the cipher exhibits a non-uniform distribution of output differences for certain input differences.

Steps:

1. **Choose a Pair of Plaintexts**: Select two plaintexts P_1 and P_2 such that their XOR difference is:

$$\Delta P = P_1 \oplus P_2$$

2. **Analyze the Cipher**: Examine how this plaintext difference propagates through each round of the DES encryption.
3. **Identify Key Dependencies**: Observe pairs of ciphertexts C_1 and C_2, resulting in a specific XOR difference $\Delta C = C_1 \oplus C_2$, and correlate it to the round keys.
4. **Deduce Round Keys**: Use statistical biases to guess portions of the key.

Example:

Suppose:

- Plaintext pairs $P_1 = 0x01010101$ and $P_2 = 0x02020202$,
- XOR difference $\Delta P = 0x03030303$.

Track this difference across DES rounds and analyze the output differences ΔC to identify potential key candidates.

---

2. Linear Analysis

Overview:

- **Linear Cryptanalysis** finds approximate linear relationships between plaintext, ciphertext, and the key bits.
- It focuses on the **linear approximations** of the cipher's components, such as S-boxes and XOR operations.

Steps:

1. **Linear Approximations**: Find linear expressions that relate the input and output bits of S-boxes or other components, such as:

$$P[i] \oplus C[j] \oplus K[k] = 0$$

where $P[i]$, $C[j]$, and $K[k]$ are specific bits of plaintext, ciphertext, and key.

2. **Compute Bias**: For a good approximation, count how often the above relationship holds across multiple encryptions. The more it deviates from 50%, the better the approximation.
3. **Attack the Key**: Use known plaintext-ciphertext pairs to deduce parts of the key that align with the observed biases.

Example:

1. Suppose the S-box transformation has an approximation:

$$x_1 \oplus x_2 \oplus y_1 \approx k \quad \text{(with bias 0.1)},$$

   where x_1, x_2 are input bits, y_1 is an output bit, and k is part of the key.
2. Collect many plaintext-ciphertext pairs to identify which k satisfies the approximation most often.

Comparison

Aspect	Differential Cryptanalysis	Linear Cryptanalysis
Basis	Tracks differences in input/output pairs.	Tracks approximate linear relationships.
Attack Model	Requires chosen plaintext pairs.	Requires known plaintext-ciphertext pairs.
Focus	Dependency of XOR differences.	Statistical biases in bit correlations.
Effectiveness	Effective on Feistel networks like DES.	Requires many pairs but is versatile.

Practical Example on DES

1. **Differential Analysis**:
   - Use chosen plaintext pairs like $\Delta P = 0x40080000$, track changes through S-boxes.
   - Analyze the XOR difference ΔC in ciphertext to deduce parts of the subkey for a round.
2. **Linear Analysis**:
   - Use approximations such as $P[3] \oplus C[5] \oplus K[2] = 0$.
   - Check bias across 2^n known pairs, where n is the number of key bits to attack.

Both methods require substantial computational resources but highlight the statistical vulnerabilities of DES, which led to the adoption of stronger ciphers like AES.

13. A and B are the only two stations on an Ethernet. Each has a steady queue of frames to send. Both A and B attempt to transmit a frame, collide, and A wins the first backoff race. At the end of this successful transmission by A, both A and B attempt to transmit and collide. What is the probability that A wins the second backoff race?

To determine the probability that A wins the second backoff race, we need to understand the **binary exponential backoff** mechanism used in Ethernet's CSMA/CD protocol.

Binary Exponential Backoff

1. **After a Collision**: Each station involved in the collision chooses a random backoff time uniformly from a range that depends on the number of collisions it has experienced:
   - After the first collision: Range is $[0, 1]$ (2 slots).
   - After the second collision: Range is $[0, 3]$ (4 slots).
   - After the third collision: Range is $[0, 7]$ (8 slots), and so on.
2. **Winning the Backoff Race**: A station wins if it selects a smaller backoff time than its competitor.

Given Situation

1. Both A and B have experienced **one collision** each (the initial collision).
2. For the second transmission attempt, both choose their backoff times from the range $[0, 1]$.

Probability Analysis

- The backoff times are chosen randomly and independently.
- Possible outcomes for A's and B's backoff times:
  - $(0, 0)$: Neither wins (collision).
  - $(0, 1)$: A wins.
  - $(1, 0)$: B wins.
  - $(1, 1)$: Neither wins (collision).

Thus, the probability $P(A \text{ wins})$ is:

$$P(A \text{ wins}) = \frac{\text{Number of favorable outcomes for } A}{\text{Total number of outcomes}}$$

- Total outcomes = 4 ($(0, 0), (0, 1), (1, 0), (1, 1)$).
- Favorable outcomes for A: $(0, 1)$.

$$P(A \text{ wins}) = \frac{1}{4}.$$

Conclusion

The probability that A wins the second backoff race is $\boxed{\frac{1}{4}}$.

14. Calculate the efficiency of Slotted Aloha protocol. Explain Hidden Terminal Problem. How that is controlled using RTS-CTS scheme.

1. Efficiency of Slotted ALOHA Protocol

Slotted ALOHA Basics

- In Slotted ALOHA, time is divided into discrete slots, and nodes transmit only at the beginning of a time slot.
- If two or more nodes transmit in the same slot, a collision occurs, and none of the transmissions are successful.

Efficiency Derivation

Let G be the average number of packets transmitted per time slot (including retransmissions).

1. **Probability of a Successful Transmission**:
   - For a transmission to be successful, exactly one node must transmit in a given slot.
   - Probability of no transmissions: e^{-G} (Poisson distribution).
   - Probability of one transmission: $G \cdot e^{-G}$.

   Therefore, the probability of a successful transmission is:

   $$P_{\text{success}} = G \cdot e^{-G}.$$

2. **Efficiency**:
   - Efficiency is the fraction of time slots used for successful transmissions.
   - Maximum efficiency occurs at $G = 1$, where:

   $$\text{Efficiency} = G \cdot e^{-G} = 1 \cdot e^{-1} \approx 0.3679.$$

Conclusion:

The maximum efficiency of Slotted ALOHA is approximately **36.8%**.

---

2. Hidden Terminal Problem

Definition:

The hidden terminal problem occurs in wireless networks when two devices, A and C, are outside each other's range but both are within range of a third device, B. If A and C transmit simultaneously, their signals collide at B, but neither A nor C can detect this collision because they cannot sense each other's transmissions.

Example:

- Device A wants to transmit to B, and device C also wants to transmit to B.
- Since A and C are not aware of each other's presence, they may transmit at the same time, causing interference at B.

---

3. RTS-CTS Scheme to Solve Hidden Terminal Problem

The **Request to Send (RTS) - Clear to Send (CTS)** mechanism is part of the IEEE 802.11 wireless protocol to avoid collisions caused by hidden terminals.

Steps:

1. **RTS (Request to Send)**:
   - The sender (e.g., A) sends a short RTS frame to the receiver (e.g., B) to request permission to transmit.
2. **CTS (Clear to Send)**:
   - If the receiver B is ready to receive, it replies with a CTS frame, granting permission to A to transmit.
3. **Transmission**:
   - After receiving the CTS, A starts transmitting the data.
   - Other devices that hear the CTS (e.g., C) defer their transmissions for the duration specified in the CTS.

Benefits:

- Devices like C, which overhear B's CTS, avoid transmitting, preventing collisions.
- The hidden terminal problem is mitigated as C is informed indirectly about A's transmission.

Summary Table

Aspect	Slotted ALOHA	Hidden Terminal Problem	RTS-CTS Solution
Efficiency	Max 36.8%	Leads to collisions at a common receiver.	Prevents collisions using RTS-CTS frames.
Key Challenge	Collisions when multiple nodes transmit.	Collisions from transmitters unaware of each other.	Requires additional control overhead.

15. In MACAW protocol, SIFS<PIFS<DIFS. What is the reason behind this ordering?

The ordering of **SIFS < PIFS < DIFS** in the **MACAW (Multiple Access with Collision Avoidance for Wireless)** protocol is designed to prioritize certain types of transmissions in a wireless network and ensure fairness and efficiency in communication. Here's the reasoning behind this ordering:

---

1. Explanation of Timing Intervals

1. **SIFS (Short Interframe Space):**
   - **Purpose:** Used for high-priority transmissions like acknowledgments (ACK), RTS (Request to Send), CTS (Clear to Send), and data frame fragments.
   - **Reason for Short Interval:**
     - To minimize the delay in time-critical transmissions and ensure they have the highest priority.
     - Ensures a quick response to ongoing communication to maintain efficient use of the channel.

2. **PIFS (Point Coordination Function Interframe Space):**
   - **Purpose:** Used by the Point Coordinator in a centralized network (e.g., for polling in PCF mode).
   - **Reason for Intermediate Interval:**
     - Gives slightly lower priority than SIFS but higher than regular contention-based transmissions.
     - Enables the point coordinator to manage traffic in the network while minimizing interference with ongoing data exchanges.
3. **DIFS (Distributed Coordination Function Interframe Space):**
   - **Purpose:** Used for regular data frame transmissions in the contention-based Distributed Coordination Function (DCF) mode.
   - **Reason for Longest Interval:**
     - Provides the lowest priority, ensuring that high-priority operations like acknowledgments and point-coordinator functions are not disrupted.
     - Prevents collisions by allowing other priority frames to access the channel first.

---

2. Why the Order SIFS < PIFS < DIFS?

Efficiency and Priority Management

- **SIFS < PIFS:**
  - Ensures that acknowledgment frames (ACK), RTS/CTS, and fragments are sent without unnecessary delays, preventing time-sensitive communication from being interrupted by point coordination traffic.
  - Avoids collisions during high-priority exchanges.
- **PIFS < DIFS:**
  - Ensures that the Point Coordinator can manage traffic efficiently, even in the presence of regular contention-based transmissions.
  - The additional gap in DIFS allows the point coordinator to gain access to the medium before regular stations in the network.

---

3. Practical Impact

- **SIFS:** Facilitates rapid and reliable communication within an ongoing exchange.
- **PIFS:** Allows centralized control (e.g., polling) to manage network traffic efficiently.
- **DIFS:** Ensures fairness among competing devices by allowing contention-based access only after higher-priority operations are completed.

Summary Table

Interval	Priority	Purpose	Reason for Ordering
SIFS	Highest	ACK, RTS/CTS, frame fragments	Ensures rapid response and minimal delay in critical transmissions.
PIFS	Medium (PCF mode)	Point Coordinator polling	Allows centralized management without disrupting ongoing communication.
DIFS	Lowest (DCF mode)	Contention-based data transmissions	Provides fairness by deferring to high-priority traffic first.

By adhering to this order, MACAW ensures a balance between **efficiency**, **fairness**, and **reliability** in wireless networks.

16. Suppose an active adversary wishes to decrypt a particular message c = me (mod n) intended for Alice. Assume Alice will decrypt arbitrary cipher-text for adversary other than c itself. Find the procedure adversary will follow to reveal the plaintext message m corresponding to c from the results provided by Alice.

The problem describes a **Chosen Ciphertext Attack (CCA)** scenario on RSA, where the adversary exploits Alice's willingness to decrypt arbitrary ciphertexts (except for c) to deduce the plaintext message m corresponding to $c = m^e \mod n$. Here's how the adversary can proceed:

Step-by-Step Procedure

1. Goal:

Recover m, the plaintext of $c = m^e \mod n$, without directly asking Alice to decrypt c.

2. Key Idea:

Leverage the RSA property:

$$\text{If } c' = c \cdot r^e \mod n, \text{ then } c' \text{ decrypts to } m \cdot r \mod n.$$

Here, r is a random value chosen by the adversary.

3. Procedure:

1. **Choose a Random r:**
   - Select a random integer r such that $\gcd(r, n) = 1$ (ensuring r is invertible modulo n).
2. **Modify Ciphertext c:**
   - Compute a new ciphertext c':

   $$c' = c \cdot r^e \mod n.$$

3. **Query Alice with c':**
   - Ask Alice to decrypt c' (since $c' \neq c$, Alice will respond).
   - Let m' be the plaintext returned by Alice:

   $$m' = m \cdot r \mod n.$$

4. **Recover m:**
   - Compute the modular inverse of $r \mod n$, denoted as r^{-1}.
   - Multiply m' by r^{-1} to retrieve m:

$$m = m' \cdot r^{-1} \mod n.$$

Why This Works

The modified ciphertext $c' = c \cdot r^e \mod n$ decrypts to $m \cdot r \mod n$ because of the properties of RSA decryption:

- Decryption involves computing $(c')^d \mod n$, where d is the private key.
- Substituting c':

$$(c')^d = (c \cdot r^e)^d \mod n = (m^e)^d \cdot (r^e)^d \mod n = m \cdot r \mod n.$$

Thus, the adversary reconstructs m using $m' = m \cdot r \mod n$.

Summary

The adversary can recover the plaintext m by:

1. Generating a random r.
2. Crafting $c' = c \cdot r^e \mod n$.
3. Asking Alice to decrypt c'.
4. Using the decrypted result m' to compute $m = m' \cdot r^{-1} \mod n$.

This attack leverages the mathematical structure of RSA and the adversary's ability to query decryptions of chosen ciphertexts other than c.

17. 3 is a popular choice for public key exponent in RSA – Can you reason about it.

The public key exponent $e = 3$ is a popular choice in RSA for several practical and mathematical reasons. Here's an explanation:

---

1. Efficiency in Encryption

- RSA encryption involves computing $c = m^e \mod n$, where e is the public key exponent.
- When $e = 3$, the encryption process is significantly faster because exponentiation with a small exponent requires fewer multiplications. For example:

$$m^3 = m \cdot m \cdot m.$$

- This efficiency is particularly beneficial for systems that perform many encryptions, such as web servers handling numerous HTTPS connections.

---

2. Smaller Public Key

- The public key e is part of the pair (e, n). Using $e = 3$ keeps the public key compact and easy to store or transmit.

---

3. Security Considerations

Although $e = 3$ is small, RSA remains secure under proper conditions:

1. **Large n:** The modulus $n = p \cdot q$ must be sufficiently large (e.g., 2048 bits) to ensure the security of RSA.
2. **Padding Schemes:** Proper padding (e.g., OAEP in modern RSA implementations) is used to avoid vulnerabilities like the "small plaintext attack" when $e = 3$.
   - Without padding, if $m^3 < n$, $c = m^3 \mod n$ does not wrap around, and an attacker could compute m by taking the cube root of c.

---

4. Avoiding $e = 1$ and Other Small Values

- $e = 1$ is not used because $c = m^1 \mod n = m$, meaning no encryption is applied.
- $e = 2$ is avoided because it does not work with the RSA requirement that e and $\phi(n)$ (Euler's totient function of n) must be coprime. For $e = 2$, e would not be coprime to $\phi(n)$ if n includes even numbers.

---

5. Balance Between Speed and Security

- Larger values of e, such as $e = 65537$ (another popular choice), offer stronger security against certain attacks but result in slower encryption due to more multiplications.
- $e = 3$ strikes a balance, offering good performance while remaining secure under proper conditions.

---

6. Historical Context

- $e = 3$ was one of the earliest choices for RSA public keys and has been widely adopted for its simplicity and speed.
- However, due to some vulnerabilities in improperly implemented padding schemes, $e = 65537$ (a Fermat prime) has become more common in modern RSA.

Conclusion

Choosing $e = 3$ provides:

1. Faster encryption due to reduced computational complexity.
2. Simplicity in implementation.
3. Compact public keys.

However, its security relies heavily on using proper padding schemes and ensuring n is large enough. For these reasons, while $e = 3$ is still used in some applications, larger values like $e = 65537$ are preferred for enhanced security in modern systems.

18. If a firewall was deployed at IIT Bhubaneswar, would it make the following system less vulnerable to attacks, more vulnerable to attacks, or the same?
- IIT's academic building, in the context of remote buffer overflow attacks.
- IIT's web servers, in the context of DDoS attacks.

Deploying a firewall at IIT Bhubaneswar would have different effects on the system's vulnerability depending on the type of attack being considered. Let's analyze both scenarios:

---

1. IIT's Academic Building in the Context of Remote Buffer Overflow Attacks

- **Buffer overflow attacks** typically involve an attacker sending more data to a buffer than it can handle, causing the program to overwrite adjacent memory and potentially execute arbitrary code. These attacks can occur when external, potentially malicious entities interact with vulnerable software running on a system, such as servers or workstations.
- **Impact of a Firewall:**
  - **Less Vulnerable:** A firewall can help reduce the exposure of internal systems to external attackers by blocking unauthorized access to critical ports or services. If the academic building's systems are behind a firewall, the firewall can block direct external access to vulnerable applications or servers, mitigating remote buffer overflow attempts.
  - **Contextual Factors:** The effectiveness of the firewall depends on whether it is configured to inspect and filter traffic for malicious payloads or suspicious patterns that could indicate an attack. Firewalls can prevent incoming connections to vulnerable services and thus reduce the risk of remote buffer overflow attacks.

Conclusion: The firewall would make the academic building **less vulnerable** to remote buffer overflow attacks by blocking unauthorized access to vulnerable systems.

---

2. IIT's Web Servers in the Context of DDoS (Distributed Denial of Service) Attacks

- **DDoS attacks** involve overwhelming a target system (such as a web server) with a massive volume of traffic, rendering it unavailable to legitimate users. DDoS attacks generally target the availability of the system, rather than exploiting specific vulnerabilities.
- **Impact of a Firewall:**
  - **More Vulnerable or Same:** While a firewall can help block some types of traffic, traditional firewalls are not particularly effective at mitigating large-scale DDoS attacks, especially if the attack involves a high volume of traffic coming from multiple sources (distributed).
  - **DDoS Mitigation Techniques:** Specialized solutions, such as **intrusion detection systems (IDS)**, **intrusion prevention systems (IPS)**, **rate limiting**, **traffic filtering**, and cloud-based DDoS protection services, are generally more effective for mitigating DDoS attacks. Firewalls may help with basic filtering, but they typically aren't designed to handle the massive traffic volumes associated with DDoS attacks.

Conclusion: A firewall would likely make the web servers **more vulnerable** to DDoS attacks or at best have **no significant effect** since traditional firewalls are not optimized for handling the volume and nature of DDoS traffic.

Summary

- **IIT's Academic Building (Remote Buffer Overflow Attacks):** The firewall would likely **make the system less vulnerable** to attacks by blocking unauthorized access to potentially vulnerable systems.
- **IIT's Web Servers (DDoS Attacks):** The firewall would likely **have little to no effect** or **make the system more vulnerable** to DDoS attacks, as firewalls are typically not designed to mitigate such high-volume attacks.

Show mathematically: given a RSA public key (N, e) and the factorization of N, one can derive the secret key d.

In RSA, the public key consists of (N, e), where:

- $N = p \times q$, the product of two large prime numbers p and q.
- e is the public exponent, chosen such that $\gcd(e, \phi(N)) = 1$, where $\phi(N) = (p-1)(q-1)$ is Euler's totient function.

The secret key d is the modular inverse of e modulo $\phi(N)$, i.e., d satisfies:

$$e \cdot d \equiv 1 \mod \phi(N)$$

Given the factorization of N, i.e., knowing p and q, you can derive the secret key d through the following steps:

Step 1: Calculate $\phi(N)$

Since $N = p \times q$, we can calculate $\phi(N)$ using the formula:

$$\phi(N) = (p-1)(q-1)$$

Step 2: Compute d, the Modular Inverse of e Modulo $\phi(N)$

To find d, we need to compute the modular inverse of e modulo $\phi(N)$, which means finding d such that:

$$e \cdot d \equiv 1 \mod \phi(N)$$

This can be done using the **Extended Euclidean Algorithm**, which solves the equation $e \cdot d - \phi(N) \cdot k = 1$ for d and some integer k.

Extended Euclidean Algorithm:

1. Apply the Euclidean algorithm to find the greatest common divisor (gcd) of e and $\phi(N)$:
   - Start by dividing $\phi(N)$ by e, and continue dividing the remainder into the previous divisor.
   - This process will give us the greatest common divisor $\gcd(e, \phi(N))$, which we know is 1 since e and $\phi(N)$ are coprime.
2. Use the Extended Euclidean algorithm to express 1 as a linear combination of e and $\phi(N)$:
   - The final step of the Euclidean algorithm will give us coefficients d and k such that:

   $$e \cdot d + \phi(N) \cdot k = 1$$

   - Here, d is the modular inverse of e modulo $\phi(N)$, i.e., $d \equiv e^{-1} \mod \phi(N)$.

Step 3: Final Result

Once d is computed, we have the secret key, which allows for decryption in RSA. Thus:

$$d \equiv e^{-1} \mod \phi(N)$$

Summary of the Steps:

1. Compute $\phi(N) = (p-1)(q-1)$.
2. Use the Extended Euclidean Algorithm to compute d such that $e \cdot d \equiv 1 \mod \phi(N)$.
3. The resulting d is the secret key.

This shows mathematically how knowing the factorization of N allows you to derive the private key d in RSA.

21. What RC4 key value will leave S unchanged during initialization? That is, after the initial permutation of S, the entries of S will be equal to the values from 0 through 255 in ascending order.

In RC4, the key scheduling algorithm (KSA) initializes a state array S of 256 bytes, which contains values from 0 to 255. The goal of the KSA is to permute this array based on the key provided. The question asks which key will leave the array S unchanged during initialization, meaning after the permutation step, S should remain in its original order: $S[0] = 0, S[1] = 1, \ldots, S[255] = 255$.

RC4 Key Scheduling Algorithm (KSA) Overview:

The KSA uses the following steps:

1. **Initialization**:

   $$S[i] = i \quad \text{for} \quad i = 0, 1, \ldots, 255$$

   Initially, S is in the order $[0, 1, 2, \ldots, 255]$.

2. **Permutation**: The array is then permuted based on the key. The permutation is controlled by the key, which is repeated as necessary to match the length of S. For each index i in S, the value of $S[i]$ is swapped with another value determined by the key.

   Specifically:

$$j = (j + S[i] + K[i \mod \text{key length}]) \mod 256$$

   where K is the key, j is an index, and i iterates over all the values of S.

Condition for Unchanged S:

For S to remain unchanged, the permutation process must not alter the initial values of S. That is, for every i, the swap operation should effectively have no effect, implying that:

$$S[i] = i \quad \text{for all} \quad i$$

To achieve this, we need the key to ensure that the index j calculated in the permutation step never causes a swap of $S[i]$ with any other value.

Key Choice:

The only key that will leave S unchanged is one where the key scheduling algorithm effectively does **no swaps** during its execution. This happens when:

$$K[i \mod \text{key length}] = 0 \quad \text{for all} \quad i$$

This means that the key must be a sequence of zeros. If the key is all zeros, the calculation for j simplifies to:

$$j = (j + S[i] + 0) \mod 256 = j$$

Thus, the value of j will always equal its previous value, and no swap will occur because the array S will not change.

Conclusion:

The key that will leave the state array S unchanged during initialization in RC4 is a **key that is all zeros** (i.e., $K = \{0, 0, 0, \ldots\}$).

In other words, the key should be of any length, but all of its values should be 0. This ensures that the permutation step does not alter the initial state of S, and it remains in the ordered state $[0, 1, 2, \ldots, 255]$.

22. Suppose the round trip propagation delay for a 8 Mbps Ethernet having 48-bit jamming signal is 45 μs. What is the minimum frame size?

To determine the minimum frame size for an Ethernet with a round-trip propagation delay and a jamming signal, we need to account for the **slot time** and the need for collision detection. Ethernet systems use a minimum frame size to ensure that the sender is able to detect a collision before it finishes transmitting the frame.

Given:

- **Propagation delay (round trip)**: $45\,\mu s$
- **Transmission speed**: $8\,\text{Mbps}$
- **Length of the jamming signal**: $48\,\text{bits}$

Step 1: Calculate the propagation delay in one direction (one-way delay):

The **round-trip propagation delay** is the time it takes for a signal to go from the sender to the receiver and then back. Since the round-trip delay is given as $45\,\mu s$, the **one-way propagation delay** is:

$$\text{One-way delay} = \frac{\text{Round-trip delay}}{2} = \frac{45\,\mu s}{2} = 22.5\,\mu s$$

Step 2: Calculate the transmission time for 48 bits (jamming signal):

The jamming signal consists of 48 bits, and the transmission speed is $8\,\text{Mbps}$ (8 million bits per second). To calculate the time to send a 48-bit jamming signal:

$$\text{Transmission time for jamming signal} = \frac{\text{Jamming signal length (in bits)}}{\text{Transmission rate}} = \frac{48\,\text{bits}}{8 \times 10^6\,\text{bits per sec}$$

Step 3: Calculate the minimum frame size:

In Ethernet, the **minimum frame size** is determined by the following criteria:

1. The sender must be able to detect a collision before it finishes transmitting the frame.
2. The frame must be long enough to ensure that during the transmission, if a collision occurs, it is detected by the sender.

The total time during which the sender must be transmitting the frame (to ensure collision detection) is at least equal to the round-trip propagation delay. Therefore, the frame must be transmitted long enough to cover both the propagation delay and the jamming signal.

Thus, the **minimum transmission time** is:

$$\text{Minimum transmission time} = \text{Round-trip propagation delay} + \text{Jamming signal transmission ti}$$

This is:

$$\text{Minimum transmission time} = 45\,\mu s + 6\,\mu s = 51\,\mu s$$

Now, using the transmission speed of $8\,\text{Mbps}$, we calculate the **minimum frame size**:

$$\text{Minimum frame size} = \text{Transmission rate} \times \text{Minimum transmission time}$$

$$\text{Minimum frame size} = 8 \times 10^6\,\text{bps} \times 51 \times 10^{-6}\,\text{s} = 408\,\text{bits}$$

Step 4: Convert bits to bytes:

Since Ethernet frames are typically measured in bytes, we convert the minimum frame size from bits to bytes:

$$\text{Minimum frame size (in bytes)} = \frac{408 \text{ bits}}{8 \text{ bits per byte}} = 51 \text{ bytes}$$

Conclusion:

The **minimum frame size** for the given Ethernet configuration is **51 bytes**.

23. In RC4, if S[2] = 0 and S[1] =/2, then what is the probability of 2nd output be 0. How this can be exploited by adversaries?

In the RC4 encryption algorithm, the output is generated by applying the **Keystream** generated from the state array S through the **PRGA (Pseudo-Random Generation Algorithm)**. The output depends on the internal state of the array S, which is permuted during the **KSA (Key Scheduling Algorithm)** phase.

Scenario Analysis:

Given:

- $S[2] = 0$
- $S[1] \neq 2$

We are interested in the probability that the **2nd output byte** from RC4's output will be 0, and how this scenario could be exploited by adversaries.

Step 1: Understanding RC4 Output Generation:

RC4's PRGA works as follows:

1. The variables i and j are incremented and updated based on the values in the state array S.
2. The output byte is generated by XORing $S[i]$ and $S[j]$, which are selected based on the current values of i and j.
3. The output byte O is given by:

$$O = S[(S[i] + S[j]) \mod 256]$$

4. This process continues to generate a sequence of output bytes.

Step 2: Identifying the Probability of the 2nd Output Being 0:

In RC4, the output byte at each step depends on the values in $S[i]$ and $S[j]$, which are modified by the PRGA. The specific question asks about the **2nd output byte** and the condition where $S[2] = 0$ and

$S[1] \neq 2.$

The 2nd output byte occurs after the first two steps in the PRGA:

1. In the first step, the algorithm increments i and j, and calculates the first output byte.
2. In the second step, the next values of i and j are used to compute the next output byte, which is the **second byte** in the keystream.

The second output byte is determined by the values of $S[i]$ and $S[j]$ at that step. If $S[2] = 0$, the second output byte will be determined by the XOR of $S[i]$ and $S[j]$.

For the second output to be 0, we must have:

$$S[i] = S[j]$$

since:

$$S[i] \oplus S[j] = 0 \quad \text{implies} \quad S[i] = S[j]$$

If $S[2] = 0$, and $S[1] \neq 2$, it's likely that the second output byte could be 0, depending on the specific values of i and j. However, the exact probability depends on the state of the entire array S and the relationship between i and j.

Step 3: Exploiting This by Adversaries:

If the second output byte is 0 with a significant probability under certain conditions, this can be exploited by an adversary in the following ways:

1. **Key Recovery Attacks**: By analyzing the keystream for patterns or repeated outputs (e.g., frequent zeros), adversaries can potentially make inferences about the internal state of the key stream and, in some cases, deduce parts of the secret key. The ability to predict outputs based on known state conditions (such as $S[2] = 0$) weakens the security of RC4.
2. **Statistical Bias**: If certain byte values, such as 0, appear with higher probability than expected in the output, this creates a **bias** in the keystream. An adversary can exploit this bias by analyzing the frequency of keystream bytes and comparing them against expected uniform distributions. This kind of statistical attack can be used to reverse-engineer the keystream or part of the secret key.
3. **Known Plaintext Attacks**: If the adversary knows or can guess part of the plaintext and has access to the ciphertext, they can use this knowledge to search for patterns in the keystream. If the adversary detects a recurring keystream byte (such as 0), this could provide clues about the internal state of the cipher, leading to potential vulnerabilities in key recovery.
4. **Chosen Plaintext Attacks**: In some cases, if an adversary can encrypt arbitrary plaintexts and analyze the keystream output, they may notice predictable results based on conditions like $S[2] = 0$. This predictability can be exploited to make educated guesses about the keystream or even the key itself.

Conclusion:

The probability that the second output byte is 0 in RC4 depends on the state of the array S, and in the scenario where $S[2] = 0$ and $S[1] \neq 2$, the second output byte might indeed have a bias towards 0.

This bias can be exploited by adversaries through statistical analysis or known plaintext attacks, undermining the security of the RC4 stream cipher. This is why RC4 is considered cryptographically weak and vulnerable to various attacks.

24. In DES, Find the output of the initial permutation box when the input is given in hexadecimal as: 0x0002 0000 0000 0001

To find the output of the **Initial Permutation (IP)** in DES, we need to perform the permutation according to the DES IP table. The input is given in **hexadecimal** format: `0x0002 0000 0000 0001`.

Step 1: Convert the hexadecimal input into binary:

We begin by converting the hexadecimal input into a 64-bit binary string.

$$0x0002000000000001 = 0000\ 0000\ 0000\ 0010\ 0000\ 0000\ 0000\ 0001$$

This corresponds to the following 64-bit binary string:

$$0000000000000010000000000000000000000000000000000000000000000001$$

Step 2: Find the Initial Permutation (IP) table:

The IP table is a predefined permutation, which reorders the bits of the input. The positions in the IP table correspond to specific bit positions in the original 64-bit input.

Here is the **IP table** (each number indicates the bit position):

```makefile
IP: 58 50 42 34 26 18 10 2
    60 52 44 36 28 20 12 4
    62 54 46 38 30 22 14 6
    64 56 48 40 32 24 16 8
    57 49 41 33 25 17 9 1
    59 51 43 35 27 19 11 3
    61 53 45 37 29 21 13 5
    63 55 47 39 31 23 15 7
```

This means that the 58th bit of the input will go to the 1st position of the output, the 50th bit will go to the 2nd position, and so on.

Step 3: Apply the IP to the binary input:

Let's permute the input binary string based on the IP table. The binary input is:

$$0000000000000010000000000000000000000000000000000000000000000001$$

We now permute the bits according to the IP table. The bit positions in the table start from 1, so we index the input from the left (1 to 64).

For example:

- The 58th bit (from the input) goes to the 1st position of the output.
- The 50th bit goes to the 2nd position, and so on.

After applying the IP, the output in binary will be:

$$0010000000000001000000000000000100000000000000000000000000000000$$

Step 4: Convert the binary result to hexadecimal:

Now, we convert the resulting 64-bit binary string back into hexadecimal format:

$$0010\ 0000\ 0000\ 0001\ 0000\ 0000\ 0000\ 0000 = 0x2001000000000000$$

Final Answer:

The output of the Initial Permutation (IP) when the input is `0x0002 0000 0000 0001` is:

$$0x2001000000000000$$

25. LFSR based stream ciphers are prone to reconstruction attack. Prove/disprove it.

Reconstruction Attacks in LFSR-based Stream Ciphers

Linear Feedback Shift Register (LFSR)-based stream ciphers are known to be vulnerable to certain types of attacks, including **reconstruction attacks**. To understand why this is the case, let's break down the concepts and reasoning involved.

LFSR Overview

An **LFSR** is a shift register in which the input to the register is a linear function of the current state of the register. In the case of an LFSR-based stream cipher, the internal state of the LFSR is used to generate a sequence of bits (the keystream), which is then XORed with the plaintext to produce ciphertext.

The LFSR mechanism:

- A **feedback polynomial** defines which bits from the state register are XORed to form the new input bit.
- The output bit of the LFSR (or the tapped bits) is used for encryption (keystream).

Reconstruction Attack Overview

In a **reconstruction attack**, an attacker attempts to determine the internal state of the LFSR, given enough output bits from the cipher. The goal is to reconstruct the LFSR's initial state (or the secret key in the case of stream ciphers), which can then be used to decrypt the ciphertext or generate the same keystream.

Proving LFSR Vulnerability to Reconstruction Attacks

1. Key Observation: LFSR is Linear

The most critical property of LFSRs is that they are **linear** systems. The output bits generated by the LFSR are a linear function of the internal state. This property makes the system susceptible to attacks using **linear algebra** techniques.

In particular, an attacker who can observe a sufficient number of output bits from the LFSR can solve for the internal state of the LFSR using a method called **Berlekamp-Massey Algorithm** or **Gaussian elimination**.

2. Berlekamp-Massey Algorithm

The **Berlekamp-Massey Algorithm** is a well-known algorithm that can efficiently reconstruct the minimal polynomial (i.e., the feedback polynomial) of an LFSR given a sequence of output bits. Once the feedback polynomial is known, the attacker can determine the initial state of the LFSR, which allows them to predict the subsequent keystream bits.

- This algorithm requires a number of output bits equal to the degree of the LFSR to fully reconstruct the system.
- The more bits the attacker has, the more accurate the reconstruction is. For an LFSR with a length n, $2n$ output bits are typically enough for a successful reconstruction attack.

3. Implication for LFSR-based Ciphers

- If an attacker has access to a significant number of output bits from an LFSR-based stream cipher, they can apply the Berlekamp-Massey algorithm to reconstruct the LFSR's internal state, revealing the key or the keystream.
- This essentially breaks the security of the cipher, as the attacker can now regenerate the keystream used to encrypt the message, allowing for decryption.

Disproving LFSR Vulnerability to Reconstruction Attacks (Under Certain Conditions)

While LFSR-based stream ciphers are generally vulnerable to reconstruction attacks, there are **certain countermeasures** that can make such attacks harder:

1. **Non-Linear Feedback or Combination of LFSRs:** Some modern stream ciphers use combinations of multiple LFSRs or **non-linear feedback** to make the system harder to analyze. For example:
    - **Combined LFSRs**: By combining the output of multiple LFSRs (e.g., using **multiple taps** or **non-linear functions**), the system becomes much more difficult to analyze using simple linear techniques like Berlekamp-Massey.
    - **Non-linear feedback**: Introducing non-linear operations or additional layers of encryption can significantly complicate any attempt to apply linear algebraic techniques for state reconstruction.

2. **Key-Scheduling and Initialization:** In certain ciphers, **key-scheduling algorithms** are designed to provide additional complexity to the LFSR state transitions. In these systems, even if the attacker knows some bits of the keystream, reconstructing the entire internal state may not be feasible without knowledge of the secret key or other additional parameters.
3. **Longer Key Sizes:** Increasing the length of the LFSR (i.e., using a larger state) increases the number of output bits needed for successful reconstruction. For example, with a 128-bit LFSR, an attacker would need to observe at least 128 bits of output to apply the Berlekamp-Massey algorithm successfully. This can make attacks computationally expensive, though not impossible.

Conclusion

- **Proven Vulnerability**: LFSR-based stream ciphers are indeed **prone to reconstruction attacks** because of their linear nature. With enough output bits, an attacker can reconstruct the internal state of the LFSR using linear algebraic methods such as the Berlekamp-Massey algorithm.
- **Mitigation**: While it is possible to make the cipher more resistant to such attacks (e.g., by using multiple LFSRs, non-linear feedback, or longer key lengths), **LFSR-based ciphers without additional security mechanisms are fundamentally weak** and susceptible to reconstruction attacks.

Thus, LFSRs alone, without any additional measures, are vulnerable to reconstruction attacks.

26. State Confusion and Diffusion property of a block cipher. Show any of the two with an example for DES

Confusion and Diffusion in Block Ciphers

The concepts of **confusion** and **diffusion** are fundamental to the security of block ciphers. These properties were introduced by Claude Shannon, and they aim to make the relationship between the ciphertext and the key as complex as possible, making it harder for attackers to deduce the key or plaintext through cryptanalysis.

1. Confusion:

- **Definition**: Confusion refers to the property that the ciphertext should depend on the key in a complex and non-linear manner. In other words, any change in the key should result in a significant and unpredictable change in the ciphertext.
- **Goal**: To obscure the relationship between the key and the ciphertext, making it hard for attackers to reverse-engineer the encryption process (i.e., deduce the key from the ciphertext).

Example in DES: In the DES algorithm, confusion is achieved using the **S-boxes** during the substitution step of each round. The S-boxes introduce non-linearity, making the relationship between the key bits and the ciphertext more complex.

How it works:

- During the encryption process in DES, the input to the **S-box** (a 6-bit block) is mapped to a 4-bit output using a predefined substitution table. This substitution step ensures confusion, as the output bits do not have a simple, direct relationship with the input bits.

For example, if you use an S-box to substitute a 6-bit input, the output is highly non-linear. A small change in the input bits can cause a large change in the output, increasing the difficulty for an attacker trying to determine the key based on the ciphertext.

S-Box Example:

Consider one of the S-boxes in DES:

```makefile
S1 =
  14 4 13 1 2 15 11 8 3 10 6 12 5 9 0 7
  0 15 7 4 14 2 13 1 10 8 9 3 5 11 12 6
  11 8 12 7 1 14 2 13 6 15 0 9 10 4 5 3
  12 1 10 15 9 2 6 8 0 5 14 3 11 4 7 13
```

If a 6-bit input `101010` is fed into this S-box, the output might be `0110`. Changing a bit of the input drastically alters the output.

---

2. Diffusion:

- **Definition**: Diffusion refers to the property that the plaintext should be spread out over the ciphertext, meaning that a change in a single bit of the plaintext should affect many bits of the ciphertext. This ensures that patterns in the plaintext are "dispersed" throughout the ciphertext.
- **Goal**: To make sure that the relationship between the plaintext and ciphertext is not simple and predictable, which helps in resisting attacks like frequency analysis.

Example in DES: In DES, diffusion is achieved through the **permutation** step. Specifically, after each round of substitution and permutation, the bits of the plaintext are spread across the ciphertext.

How it works:

- In DES, the **Initial Permutation (IP)** and the **Final Permutation (FP)** apply a fixed bit permutation to the input and output blocks, respectively.
- Within each round of DES, the output of the **S-box** step is permuted further using a **P4** permutation or other types of bit-level shifts.

During the rounds of DES, the bit positions in the left and right halves of the data block are swapped and mixed using the results from the S-boxes. The output of one round becomes the input to the next round, ensuring that the plaintext's bits are diffused throughout the ciphertext.

Diffusion Example in DES:

In the first round of DES:

- The 64-bit input block is divided into two 32-bit halves (Left and Right).

- The right half of the block goes through an **F-function**, which includes substitution and permutation.
- The results of the F-function are then XORed with the left half of the block, and the two halves are swapped.

This swapping ensures that the bits of the input are mixed, so changing one bit of the plaintext will affect many bits of the ciphertext.

Initial Permutation (IP) Example:

The initial permutation (IP) reorders the bits of the 64-bit input plaintext according to a predefined pattern, which is another form of diffusion.

```makefile
IP: 58 50 42 34 26 18 10 2
    60 52 44 36 28 20 12 4
    62 54 46 38 30 22 14 6
    64 56 48 40 32 24 16 8
    57 49 41 33 25 17 9 1
    59 51 43 35 27 19 11 3
    61 53 45 37 29 21 13 5
    63 55 47 39 31 23 15 7
```

Summary:

- **Confusion** in DES is achieved through non-linear substitution using S-boxes. This ensures that the ciphertext is not easily related to the key.
- **Diffusion** in DES is achieved through bit permutations (IP, FP) and through the swapping of left and right halves during each round of the encryption process. This ensures that changes in the plaintext affect multiple bits of the ciphertext.

Together, confusion and diffusion make it difficult for an attacker to reverse the encryption process or deduce the key.

27. State 2 week keys of DES. Why are they vulnerable?

Weak Keys in DES

In the **Data Encryption Standard (DES)**, a **weak key** is a key that results in poor or insecure encryption. DES uses a 56-bit key for encryption, which is divided into 16 rounds of processing. During the key scheduling phase, the 56-bit key is used to generate 16 subkeys (one for each round). Some keys lead to situations where the encryption is easily reversible or predictable, which significantly weakens the security of the cipher.

Two Weak Keys in DES:

1. **All 1s Key** (`0xFFFFFFFFFFFFFF`)
   - This key has all its bits set to `1`. When this key is used for DES encryption, it results in the same subkey being used in every round of the algorithm. This means that there is no actual mixing of the key during encryption, making the encryption pattern predictable and vulnerable to attacks.
2. **All 0s Key** (`0x00000000000000`)
   - This key has all its bits set to `0`. Like the all 1s key, it causes a trivial key schedule where the subkeys are repeated in every round, making the encryption process weak and easily predictable.

Why Are These Keys Vulnerable?

1. **Predictability**:
   - When weak keys are used, the subkeys generated for the rounds of DES are either all the same (for the all-1s key) or have no variability (for the all-0s key). This makes the encryption highly predictable because the same operation is applied to every round.
2. **Lack of Diffusion and Confusion**:
   - The lack of variation in the subkeys means that the key's effect on the data doesn't change from round to round, which violates the principles of **diffusion** and **confusion** in cryptography. In DES, the goal is to mix the plaintext and key as much as possible over multiple rounds to prevent patterns from emerging in the ciphertext. Weak keys fail to achieve this, which makes it easier for attackers to break the cipher.
3. **Simplified Attacks**:
   - If an attacker knows that the encryption is using a weak key, they can immediately reduce the search space of possible keys and launch a brute-force attack more efficiently. Since there are fewer distinct subkeys in the key schedule for weak keys, the attack is more successful and faster than for a random key.
4. **Key Collisions**:
   - With weak keys, due to the repeated subkeys, the internal state of the DES algorithm becomes predictable after just a few rounds. This allows an adversary to guess or infer parts of the key based on these repetitive patterns.

Conclusion:

Weak keys in DES, such as the all 1s and all 0s keys, are vulnerable because they lead to a predictable key schedule, reducing the strength of the encryption. They undermine the goals of confusion and diffusion in the cipher and make the encryption process more susceptible to cryptanalysis. To mitigate this risk, weak keys should be avoided during key generation in secure systems.

29. How both Authentication and Confidentiality can be achieved using public key (asymmetric key cryptosystems)? Is it good choice to use public key for digital signature?

Achieving Authentication and Confidentiality using Public Key Cryptosystems

In **public key (asymmetric) cryptosystems**, both **authentication** and **confidentiality** can be achieved through different techniques, often using a combination of encryption and digital signatures.

1. Confidentiality (Encryption with Public Key):

- **Confidentiality** ensures that only the intended recipient can read the message. Public key cryptography can achieve confidentiality through **encryption**.
- In public key encryption, each participant has a **public key** and a **private key**:
  - **Public key**: This key is widely distributed and can be used by anyone to encrypt a message intended for the owner of the associated **private key**.
  - **Private key**: The private key is kept secret and is used by the key owner to decrypt the message.

  How confidentiality is achieved:
  - Suppose Alice wants to send a confidential message to Bob. Bob sends Alice his **public key**.
  - Alice encrypts her message using Bob's **public key**.
  - Only Bob, who possesses the corresponding **private key**, can decrypt the message and read it.

  This guarantees confidentiality because only Bob can decrypt the message, and no one else (including an eavesdropper) can read it, even if they have access to the ciphertext.

2. Authentication (Digital Signature):

- **Authentication** ensures the sender's identity and guarantees the integrity of the message. In public key cryptosystems, **digital signatures** are used to achieve authentication.

  How authentication is achieved:
  - Alice wants to authenticate a message she sends to Bob. She first **hashes** the message (creating a fixed-size message digest).
  - Alice then **encrypts the hash** with her **private key** to create a **digital signature**.
  - Alice sends the message along with the digital signature to Bob.
  - Bob can verify the authenticity of the message by performing the following steps:
    1. **Decrypt the digital signature** using Alice's **public key** to retrieve the hash value.
    2. **Hash the received message** to calculate its hash value.
    3. If the hash values match, Bob can be confident that the message was indeed sent by Alice (since only Alice could have signed it with her private key) and that the message was not tampered with.

  This guarantees **authenticity** because only Alice, who has the private key, could have signed the message. It also guarantees **integrity** because any change in the message would result in a mismatch between the hashes.

Is it a Good Choice to Use Public Key for Digital Signatures?

Yes, using **public key cryptography** for **digital signatures** is not only a good choice, but it is also the standard and recommended approach in many cryptographic protocols and systems (such as SSL/TLS, email encryption, and blockchain).

Reasons why public key is used for digital signatures:

1. **Asymmetric Nature:**

- Public key cryptography is inherently designed for scenarios where one party (the signer) needs to sign a document or message and another party (the verifier) needs to verify it. The use of the **private key** for signing ensures that the signature can only come from the legitimate signer (the holder of the private key), while the **public key** allows anyone to verify the signature.

2. **Non-repudiation:**
    - Public key digital signatures provide **non-repudiation**, meaning that once Alice signs a message with her private key, she cannot deny having sent the message. This is important for authentication purposes in various applications such as legal agreements or financial transactions.
3. **Security of the Private Key:**
    - The private key is used only for signing, and its secrecy is paramount. The public key is distributed and can be freely shared with others to verify signatures, making it an effective way to authenticate messages or documents.
4. **Widespread Standardization:**
    - Using public key cryptography for digital signatures is standardized and widely adopted in security protocols, such as **PKI (Public Key Infrastructure)**, **SSL/TLS**, **email encryption standards** like **S/MIME**, and digital currencies (e.g., **Bitcoin**).
5. **Efficiency:**
    - Public key systems like RSA allow for the use of small and fixed-size signatures, making them efficient for authentication. While the process of signing and verification is computationally expensive compared to symmetric cryptography, it is still highly efficient and secure for many real-world applications.

Challenges of Using Public Key Cryptography for Digital Signatures:

- **Performance**: Public key cryptography operations (especially RSA) are slower than symmetric cryptography operations. This makes it less suitable for large amounts of data. However, in practice, digital signatures are applied to message digests (which are much smaller than the full message), so the performance impact is minimized.
- **Key Management**: Public key systems require careful management of public and private keys, which may require the use of a **Certificate Authority (CA)** to issue and verify keys and their associated digital certificates.

Conclusion:

- **Authentication** and **confidentiality** can both be achieved using public key cryptosystems. Confidentiality is achieved through encryption, and authentication is achieved through digital signatures.
- Using public key cryptography for digital signatures is a **good choice** because it offers strong security, non-repudiation, and widespread standardization, although it comes with performance considerations and key management challenges.

30. Consider a cipher that has the three keys, three plaintexts and four ciphertext that are represented using the following encryption table

	m1	m2	m3
k1	c2	c3	c1
k2	c1	c2	c4
k3	c3	c1	c2

Suppose that the plaintexts and keys are used with the following probabilities:
f(m1) = f (m2) = 2/7; f(m3) = 3/7
f(k1) = f (k2) = f(k3) = 1/3.
Does the above cryptosystem have perfect secrecy?

Definition of Perfect Secrecy:

A cryptosystem is said to have **perfect secrecy** if the ciphertext reveals no information about the plaintext. Mathematically, a system has perfect secrecy if for all possible plaintexts $m_1, m_2, \ldots, m_n$ and all ciphertexts $c_1, c_2, \ldots, c_n$, the probability of a ciphertext given a plaintext is independent of the plaintext. That is:

$$P(m_i|c_j) = P(m_i)$$

for all i and j. This implies that the probability of observing a particular ciphertext c_j given a plaintext m_i is the same for all plaintexts, and no additional information can be learned about the plaintext from the ciphertext.

Given Data:

- The **encryption table** maps plaintexts and keys to ciphertexts:

	m_1	m_2	m_3
k_1	c_2	c_3	c_1
k_2	c_1	c_2	c_4
k_3	c_3	c_1	c_2

- The **probabilities of plaintexts**:

$$f(m_1) = \frac{2}{7}, \quad f(m_2) = \frac{2}{7}, \quad f(m_3) = \frac{3}{7}$$

- The **probabilities of keys**:

$$f(k_1) = \frac{1}{3}, \quad f(k_2) = \frac{1}{3}, \quad f(k_3) = \frac{1}{3}$$

Check for Perfect Secrecy:

To check if the cryptosystem has perfect secrecy, we need to verify that the ciphertext distribution for each possible plaintext is independent of the plaintext itself. In other words, for each possible ciphertext c_j, we need to check if:

$$P(c_j) = \sum_{i=1}^{3} P(c_j|m_i)f(m_i)$$

where $P(c_j)$ is the probability of obtaining ciphertext c_j, and $P(c_j|m_i)$ is the conditional probability of obtaining ciphertext c_j given plaintext m_i. We can compute the probabilities for each ciphertext by summing over all possible plaintexts and keys.

Step 1: Compute the Probability $P(c_j)$

First, compute the probability of obtaining each ciphertext c_1, c_2, c_3, c_4 by marginalizing over both plaintexts and keys.

For ciphertext c_1, there are two possible ways to obtain it: m_1 with k_2 and m_2 with k_1.

$$P(c_1) = P(m_1, k_2) + P(m_2, k_1)$$

$$P(c_1) = f(m_1)f(k_2) + f(m_2)f(k_1) = \frac{2}{7} \cdot \frac{1}{3} + \frac{2}{7} \cdot \frac{1}{3} = \frac{4}{21}$$

Similarly, calculate for c_2, c_3, and c_4.

For c_2:

$$P(c_2) = P(m_1, k_1) + P(m_2, k_2) + P(m_3, k_3) = \frac{2}{7} \cdot \frac{1}{3} + \frac{2}{7} \cdot \frac{1}{3} + \frac{3}{7} \cdot \frac{1}{3}$$

$$P(c_2) = \frac{2}{21} + \frac{2}{21} + \frac{3}{21} = \frac{7}{21} = \frac{1}{3}$$

For c_3:

$$P(c_3) = P(m_1, k_2) + P(m_2, k_3) + P(m_3, k_1) = \frac{2}{7} \cdot \frac{1}{3} + \frac{2}{7} \cdot \frac{1}{3} + \frac{3}{7} \cdot \frac{1}{3}$$

$$P(c_3) = \frac{2}{21} + \frac{2}{21} + \frac{3}{21} = \frac{7}{21} = \frac{1}{3}$$

For c_4:

$$P(c_4) = P(m_2, k_2) + P(m_3, k_3) = \frac{2}{7} \cdot \frac{1}{3} + \frac{3}{7} \cdot \frac{1}{3}$$

$$P(c_4) = \frac{2}{21} + \frac{3}{21} = \frac{5}{21}$$

Step 2: Check if $P(c_j|m_i) = P(c_j)$ for Each Ciphertext

To determine if the system has perfect secrecy, we need to check if the probability of each ciphertext conditioned on the plaintext matches the overall probability of the ciphertext.

For each c_j, compute $P(c_j|m_i)$ for each i and verify if:

$$P(c_j) = \sum_{i=1}^{3} P(c_j|m_i)f(m_i)$$

Let's take c_1 as an example:

- For m_1, c_1 can only be produced by k_2, so $P(c_1|m_1) = \frac{1}{3}$.
- For m_2, c_1 can only be produced by k_1, so $P(c_1|m_2) = \frac{1}{3}$.
- For m_3, c_1 cannot be produced at all, so $P(c_1|m_3) = 0$.

Thus,

$$P(c_1) = \frac{1}{3} \cdot \frac{2}{7} + \frac{1}{3} \cdot \frac{2}{7} + 0 = \frac{4}{21}$$

This matches the previously computed $P(c_1) = \frac{4}{21}$. Similarly, check this for c_2, c_3, and c_4.

Conclusion:

The cryptosystem **does not have perfect secrecy** because the probabilities for ciphertexts do not remain consistent across different plaintexts. Therefore, the ciphertext reveals information about the plaintext, violating the condition for perfect secrecy.

35. In the Diffie-Hellman protocol, each participant selects a secret number x and sends the other participant αx mod q for some public number α .What would happen if the participants sent each other xα for some public number instead?

In the Diffie-Hellman protocol, each participant selects a secret number x and sends $\alpha^x \mod q$ (where α is a public base and q is a large prime modulus) to the other participant. This process allows both participants to independently compute a shared secret key based on their own private key and the other participant's public value.

Now, if the participants were to send each other $x\alpha \mod q$ (i.e., multiplying the secret number x by the public number α), instead of sending $\alpha^x \mod q$, the following would happen:

1. Loss of Exponentiation's Security Feature:

The Diffie-Hellman protocol relies on the difficulty of the **discrete logarithm problem** (DLP), which is the challenge of finding x from $\alpha^x \mod q$. This problem is computationally hard, which ensures that even if an adversary intercepts the public messages, they cannot easily deduce the shared secret key.

If the participants sent $x\alpha \mod q$ instead of $\alpha^x \mod q$, the problem becomes trivially solvable. Here's why:

- If each participant sends $x\alpha \mod q$, an adversary who intercepts the message can easily recover x because they already know α and q. The message is simply $x\alpha \mod q$, and given α, the adversary can solve for x using basic algebra (because the operation involved is linear, not an exponentiation).
- For example, if the message is $y = x\alpha \mod q$, the adversary can compute $x = y/\alpha \mod q$, which is straightforward if division is possible modulo q (i.e., α has a modular inverse).

2. Breaking of Key Exchange:

The entire security of the Diffie-Hellman protocol is based on the fact that only the exponentiation process, $\alpha^x \mod q$, is difficult to reverse (due to the discrete logarithm problem). If participants instead send $x\alpha \mod q$, it is no longer difficult for an adversary to deduce the secret values x and y (the secret keys of the participants). This would essentially eliminate the security guarantees provided by Diffie-Hellman.

Since the adversary can easily deduce the secret keys, they could also compute the shared secret key, $\alpha^{xy} \mod q$, and potentially decrypt or manipulate messages between the participants.

3. No Shared Secret Key:

Even if participants were to exchange $x\alpha \mod q$, they wouldn't be able to securely compute a shared secret. The formula for the shared secret in Diffie-Hellman is based on exponentiation and is designed so that each participant can combine their private key with the other participant's public key to compute the same shared secret. However, with the modification to $x\alpha$, there would be no way for both parties to compute a common secret key in a secure manner. Each participant's private key x and y cannot combine as they would in the exponentiation-based Diffie-Hellman to produce the same shared value.

Conclusion:

If participants were to send $x\alpha \mod q$ instead of $\alpha^x \mod q$, the Diffie-Hellman protocol would fail because it would become easy for an adversary to determine the secret keys of the participants and compute the shared secret. The core security of Diffie-Hellman comes from the discrete logarithm problem, which is not present in the case of linear operations like multiplying x by α. Therefore, this change would undermine the confidentiality and security of the key exchange.

39. State the size of LFSR used in content scrambling scheme (CSS) for DVD encryption.

In the Content Scrambling System (CSS) used for DVD encryption, a **16-bit Linear Feedback Shift Register (LFSR)** is used for the encryption process.

The LFSR is part of the algorithm responsible for generating a pseudo-random sequence used to scramble the data on the DVD. This pseudo-random sequence is then XORed with the plaintext data to encrypt it. The encryption process is designed to be difficult to reverse without the correct decryption key, ensuring the protection of copyrighted content on DVDs.

The LFSR's size (16 bits) determines the length of the internal state and the complexity of the generated sequence, making it an essential component of the CSS encryption mechanism.

40. Let K = 111...111 consisting of all 1's be the key for DES cryptosystem. If DESK (x) = y, then what is DESK (y)?

Given the key $K = 111\ldots111$ (all 1's) in the DES (Data Encryption Standard) cryptosystem, let's analyze the situation where $\mathrm{DES}_K(x) = y$. We are asked to determine what $\mathrm{DES}_K(y)$ would be.

Step 1: Key Setup and Initial Observations

In the DES cryptosystem, the encryption and decryption operations are dependent on the key K. When $K = 111\ldots111$, this key will undergo a series of permutations and transformations during the DES process.

Step 2: DES Structure

DES is a symmetric-key block cipher that performs 16 rounds of encryption. It uses a combination of substitution and permutation operations to achieve confusion and diffusion. The key K is divided into 16 subkeys, one for each round. Each round uses a different subkey, and the key schedule is generated by applying a series of shifts and permutations to K.

However, a crucial property of DES is that the encryption and decryption processes are essentially the same, with the only difference being the order of the subkeys: the subkeys are applied in reverse order during decryption.

Step 3: Analysis of the All-1's Key

If the key K consists of all 1's, the key schedule used to generate the subkeys will also follow a predictable structure. The fact that the key consists entirely of 1's means that the bitwise operations involved in the encryption process (XOR, AND, etc.) may simplify in a specific manner due to the uniformity of the key.

This leads to the following key insight: **the DES encryption process with a key of all 1's is reversible without much transformation**. Therefore, applying DES encryption twice with the same key K (i.e., performing $\mathrm{DES}_K(\mathrm{DES}_K(x))$) is effectively equivalent to applying the identity transformation on the input.

Step 4: Identity Property of Double DES with the Key $K = 111\ldots111$

When the key K is all 1's, the DES encryption has a special property: **applying DES twice with the same key results in the original input**. This is because the DES operation with such a key effectively undoes itself due to the symmetry of the transformations (due to the all-1's structure). Specifically, the structure of the round function and the permutations used in DES, when applied twice with the same key, cancel out.

Thus, for any input x:

$$\mathrm{DES}_K(\mathrm{DES}_K(x)) = x$$

Step 5: Conclusion

Therefore, if $\mathrm{DES}_K(x) = y$, then applying DES again on y with the same key will return the original input x:

$$\mathrm{DES}_K(y) = x$$

In summary, $\mathrm{DES}_K(\mathrm{DES}_K(x)) = x$ when $K = 111\ldots111$. This means that the encryption operation with an all-1's key behaves like an involution, effectively reversing itself after two applications.

41. Select True statements
i) Block Ciphers cannot reuse keys
ii) Block cipher and stream cipher behave similar wrt avalanche effect
iii) RC4 is used in SSL/TLS
iv) Stream Ciphers are slower than Block Ciphers
v) Mail/file transfer use block ciphers
vi) Browser/Web Links use block ciphers
vii) Block ciphers use more code than stream cipher

Let's analyze each statement one by one:

i) Block Ciphers cannot reuse keys

- **False**. Block ciphers **can** reuse keys, but reusing keys for multiple encryption operations can lead to vulnerabilities. For example, reusing the same key for encrypting multiple messages without proper initialization vectors (IVs) or other safeguards can expose the system to certain attacks. However, block ciphers themselves do not inherently forbid key reuse.

ii) Block cipher and stream cipher behave similarly with respect to avalanche effect

- **False**. While both block and stream ciphers are designed to achieve diffusion and confusion (as described in Shannon's principles), they behave differently in terms of avalanche effect. The avalanche effect refers to how a small change in the input (e.g., a single bit) significantly changes the output. In block ciphers, the entire block is altered by the change (with proper settings like padding and modes), whereas in stream ciphers, a single bit change can cause a corresponding change in the output stream.

iii) RC4 is used in SSL/TLS

- **True**. **RC4** (Rivest Cipher 4) was historically used in SSL/TLS protocols, although it has been deprecated due to security vulnerabilities (e.g., biases in the keystream and potential for attacks). Still, it was widely used for a time in SSL/TLS.

iv) Stream Ciphers are slower than Block Ciphers

- **False**. **Stream ciphers** are generally faster than block ciphers. Stream ciphers process data bit by bit (or byte by byte), which makes them more efficient for certain applications (e.g., encrypting real-time data streams). Block ciphers, on the other hand, process larger blocks of data (typically 64 or 128 bits) and may involve more computational overhead due to the complexity of block transformations.

v) Mail/file transfer use block ciphers

- **True**. Many **email and file transfer protocols** (e.g., PGP, S/MIME) use block ciphers like AES for securing data, especially when dealing with sensitive files or messages. Block ciphers offer strong security guarantees when used with appropriate modes of operation (e.g., CBC, GCM).

vi) Browser/Web Links use block ciphers

- **True**. **Modern web communication** protocols like HTTPS, which rely on SSL/TLS, often use block ciphers (e.g., AES) for encrypting data between the client and server. While stream ciphers were used historically (e.g., RC4), block ciphers are now the standard in most secure web communication.

vii) Block ciphers use more code than stream cipher

- **True**. Block ciphers generally involve more complex operations (such as multiple rounds of transformations, padding, and modes of operation), which can result in more lines of code compared to stream ciphers. Stream ciphers often have simpler and more efficient implementations, which contribute to their lower computational overhead.

Final Summary of True Statements:

- **iii) RC4 is used in SSL/TLS**
- **v) Mail/file transfer use block ciphers**
- **vi) Browser/Web Links use block ciphers**
- **vii) Block ciphers use more code than stream cipher**

42. Prove that OTP with reusing keys doesn't follow perfect secrecy.

To prove that **One-Time Pad (OTP) with key reuse** does not follow perfect secrecy, we need to recall the definition of **perfect secrecy** and the properties of the OTP encryption scheme.

1. Perfect Secrecy Definition:

A cryptosystem has **perfect secrecy** if, given the ciphertext, the probability of any plaintext message being the correct one is independent of the ciphertext. In other words, for a message m and its corresponding ciphertext c, the following condition must hold:

$$P(m|c) = P(m)$$

This means that the ciphertext c reveals no information about the plaintext m, and every possible plaintext is equally likely, given the ciphertext.

2. One-Time Pad (OTP) Encryption:

The OTP encryption scheme is defined as follows:

- Let m be a plaintext message.
- Let k be a key of the same length as m, chosen randomly.

- The ciphertext c is computed as:

$$c = m \oplus k$$

where $\oplus$ denotes the bitwise XOR operation.

For OTP to achieve **perfect secrecy**, the key k must be truly random, as long as the plaintext message m, and used only **once**. This ensures that for any plaintext m and ciphertext c, the probability distribution over possible plaintexts, given the ciphertext, is uniform (i.e., all plaintexts are equally likely).

3. Key Reuse and Its Consequences:

Now, consider the scenario where **the same key k is reused** for encrypting **multiple messages**. Suppose Alice encrypts two messages m_1 and m_2 with the same key k, resulting in ciphertexts c_1 and c_2 as follows:

$$c_1 = m_1 \oplus k$$

$$c_2 = m_2 \oplus k$$

Now, the adversary, who knows both c_1 and c_2, can easily compute the XOR of the two ciphertexts:

$$c_1 \oplus c_2 = (m_1 \oplus k) \oplus (m_2 \oplus k) = m_1 \oplus m_2$$

This operation eliminates the key k, and the adversary is left with the result of $m_1 \oplus m_2$. From this, the adversary can potentially deduce some information about the relationship between the two messages m_1 and m_2. For example, if the adversary knows that one of the messages m_1 is a common phrase or follows a known format, they may be able to deduce the structure or content of m_2.

This behavior violates **perfect secrecy** because the adversary now has information about the messages that they would not have had if the key had been used only once.

4. The Failure of Perfect Secrecy with Key Reuse:

When the key is reused, the ciphertexts corresponding to different messages can be related through the XOR operation. This relationship introduces **dependencies between ciphertexts** and **leaks information** about the plaintexts. Therefore, the ciphertext no longer hides the plaintexts completely, and the probability distribution over possible plaintexts is no longer uniform. The adversary can exploit this relationship and potentially break the encryption.

5. Conclusion:

By reusing the key k in OTP, the cryptosystem **loses perfect secrecy** because the ciphertexts for different messages can be combined to reveal information about the plaintexts. Therefore, OTP with key reuse does not follow perfect secrecy. The key must be used only once to maintain the perfect secrecy property.