Assignment - 3

Name: Gokul R

Reg. No.: 20MIS0332

Course: Cyber Security and Ethical Hacking

Campus: VIT Vellore

Symmetric: AES

AES (Advanced Encryption Standard) is a symmetric encryption algorithm that operates on fixed-size blocks of data. It uses a substitution-permutation network (SPN) structure, which involves multiple rounds of substitution and permutation operations to provide encryption and decryption.

The key components in AES are the following:

- Block Size: AES operates on blocks of 128 bits (16 bytes). If the input message is not a multiple of 16 bytes, padding is typically applied.
- Key Size: AES supports three key sizes: 128 bits, 192 bits, and 256 bits. The key size determines the number of rounds in the algorithm.
- Substitution Box (S-box): AES uses a nonlinear S-box, which substitutes each byte of the block with another byte based on a predefined lookup table. This step adds confusion to the encryption process.
- Key Expansion: The original encryption key is expanded into a set of round keys. Each round key is derived from the previous round key and undergoes a series of transformations using the S-box, bitwise XOR operations, and circular shifts.

The AES encryption process involves the following steps:

- Initial Round: The first-round key is combined with the input block using bitwise XOR.
- Rounds: AES operates on multiple rounds, the number of which depends on the key size. Each round consists of four main operations: SubBytes, ShiftRows, MixColumns, and AddRoundKey.
    - SubBytes: Each byte of the block is substituted using the S-box lookup table.
    - ShiftRows: The bytes in each row of the block are shifted cyclically to the left. The first row remains unchanged, the second row shifts by one byte, the third row shifts by two bytes, and the fourth row shifts by three bytes.
    - MixColumns: The columns of the block are mixed using a linear transformation. This operation provides diffusion and increases the security of AES.
    - AddRoundKey: The current round key is combined with the intermediate block using bitwise XOR.
- Final Round: The final round excludes the MixColumns operation, and it consists of SubBytes, ShiftRows, and AddRoundKey operations.

Once all the rounds are completed, the resulting block is the encrypted ciphertext.

Decryption using AES follows a similar process, but with the inverse operations of the encryption steps applied in reverse order. This allows the ciphertext to be transformed back into the original plaintext using the same key.

Overall, AES provides a secure and efficient symmetric encryption scheme by utilizing substitution and permutation operations along with key expansion and mixing operations.

Advantages of AES (Advanced Encryption Standard):

- Security: AES is a widely accepted and trusted encryption algorithm that provides a high level of security. It has been extensively studied and subjected to rigorous analysis by the cryptographic community.
- Efficiency: AES is designed to be computationally efficient, especially when implemented in hardware or optimized software. It offers a good balance between security and performance.
- Standardization: AES is a standard encryption algorithm adopted by various organizations, including the United States government. Its widespread use ensures interoperability and compatibility across different systems and platforms.

- Flexibility: AES supports key sizes of 128, 192, and 256 bits, allowing users to choose an appropriate level of security based on their specific needs and requirements.
- Resistance to Attacks: AES has proven resistance against various cryptographic attacks when used correctly. It offers protection against known attacks such as brute force, differential cryptanalysis, and linear cryptanalysis.

Disadvantages/Considerations of AES:

- Key Management: One of the challenges with any encryption algorithm, including AES, is proper key management. Generating and securely storing encryption keys, distributing them to authorized parties, and protecting them from unauthorized access require careful attention.
- Side-Channel Attacks: AES implementations can be vulnerable to side-channel attacks, where an attacker exploits information leaked during the encryption process, such as power consumption, timing, or electromagnetic radiation. Countermeasures need to be taken to mitigate these risks.
- Block Size Limitation: AES operates on fixed-size blocks of 128 bits. While this is suitable for many applications, it can be a limitation if there is a need to encrypt data that exceeds this block size. Modes of operation or additional techniques may be required to encrypt larger data sets.
- Quantum Computing Threat: AES, like other symmetric encryption algorithms, is susceptible to attacks by powerful quantum computers. As quantum computing advances, it poses a potential threat to the security of AES. However, post-quantum cryptographic algorithms are being developed to address this concern.

It's important to note that while AES is widely regarded as a secure and efficient encryption algorithm, the actual security of an AES implementation also depends on factors such as key management, proper usage, and the overall security of the system in which it is implemented.

Real world example: Google Drive, AES-256 used in Gmail.

Asymmetric: RSA

The RSA (Rivest-Shamir-Adleman) algorithm is a widely used asymmetric encryption algorithm that provides secure communication and digital signatures. Here's a high-level explanation of how RSA works:

Key Generation:

- Choose two large prime numbers, p and q.
- Compute n = p * q, which serves as the modulus for both the public and private keys.
- Compute the totient function of n, φ(n) = (p - 1) * (q - 1).
- Select an encryption exponent e, which is a positive integer coprime to φ(n) (i.e., gcd(e, φ(n)) = 1).
- Calculate the decryption exponent d, which is the modular multiplicative inverse of e modulo φ(n) (i.e., e * d ≡ 1 (mod φ(n))).

Key Distribution:

- The public key consists of the pair (e, n) and is shared with anyone who wants to send an encrypted message or verify a digital signature.
- The private key consists of the pair (d, n) and must be kept secret by the owner.

Encryption:

- Convert the plaintext message into a numeric representation, typically using a predefined encoding scheme.
- Apply the encryption operation using the public key: ciphertext = plaintext^e mod n. Here, ^ denotes exponentiation and mod is the modulus operation.
- Transmit the ciphertext to the intended recipient.

Decryption:

- Receive the ciphertext.
- Apply the decryption operation using the private key: plaintext = ciphertext^d mod n.
- Convert the resulting numeric representation back into the original plaintext message using the encoding scheme.

Digital Signature:

- To create a digital signature, the sender computes a cryptographic hash function (e.g., SHA-256) on the message to produce a fixed-length digest.
- The sender then encrypts the digest using their private key: signature = digest^d mod n.
- The signature is sent along with the message.
- The recipient verifies the signature by decrypting it using the sender's public key and recomputing the hash of the received message. If the two digests match, the signature is valid.

Real world Example: Virtual Private Network (VPN), web browsers, multiple network channels

---

Hash: SHA512

SHA-512 (Secure Hash Algorithm 512-bit) is a cryptographic hash function that belongs to the SHA-2 family. It is designed to take an input message and produce a fixed-size output hash value of 512 bits. The process of SHA-512 involves several steps:

- Padding: The input message is padded to ensure its length is a multiple of 1024 bits. The padding includes a 1 bit, followed by a series of 0 bits, and then the length of the original message in binary form.
- Initialization: SHA-512 initializes a 512-bit state (a set of variables) with predetermined constant values. This state will be updated as the algorithm progresses.
- Message Processing: The padded message is processed in blocks of 1024 bits each. For each block, the following steps are performed:
    - Message Expansion: The 1024-bit block is divided into 16 words of 64 bits each.
    - Message Schedule: A message schedule array of 80 64-bit words is created. This array is derived from the previous block and is used to update the state.
    - Compression: The state is updated through a series of rounds. Each round consists of various bitwise operations, such as bitwise rotations, XOR, AND, and OR operations, along with logical functions, such as majority and choice functions. These operations

mix the data in a nonlinear and systematic manner, providing diffusion and increasing the security of the hash function.

- Finalization: After processing all the blocks, the final state is obtained. The 512-bit hash value is derived by concatenating the values of the state variables.
- The resulting hash value represents a unique fingerprint of the input message. Even a small change in the input message will produce a significantly different hash value. This property makes SHA-512 suitable for verifying the integrity of data and ensuring it has not been tampered with.

It's important to note that SHA-512 is a one-way function, meaning that it is computationally infeasible to reverse the process and obtain the original message from the hash value. Additionally, SHA-512 is resistant to various cryptographic attacks when used correctly.

Implementations of SHA-512 are readily available in cryptographic libraries and frameworks, making it relatively straightforward to use in various applications requiring secure hashing.

Advantages of SHA-512:

- Strong Security: SHA-512 is designed to provide a high level of security. It produces a 512-bit hash value, making it resistant to brute-force attacks and collision attacks. It is considered a secure and robust cryptographic hash function.
- Wide Adoption: SHA-512 is widely adopted and supported by cryptographic libraries, frameworks, and systems. It is a recognized standard and used in various applications, including digital signatures, password storage, and data integrity checks.
- Efficiency: While SHA-512 generates a 512-bit hash value, it is still relatively efficient in terms of computational speed and memory usage. It strikes a balance between security and performance, making it suitable for many practical use cases.
- Fixed Output Size: SHA-512 always produces a fixed-size output of 512 bits, regardless of the input size. This allows for consistent handling and storage of hash values.

Disadvantages/Considerations of SHA-512:

- Fixed Output Size: While the fixed output size is an advantage in some cases, it can be a limitation in situations where a shorter hash length is desired. In such cases, truncation or additional hashing techniques may be necessary to obtain a shorter hash.
- Implementation Complexity: Implementing SHA-512 correctly requires attention to detail and adherence to best practices. It involves several steps and cryptographic operations, which can be challenging to implement without proper knowledge and understanding of the algorithm.
- Potential for Misuse: SHA-512, like any hash function, can be misused if not applied correctly. For example, it should not be used as a general-purpose encryption algorithm. It is crucial to use SHA-512 for its intended purpose, such as data integrity checks and digital signatures.
- Vulnerability to Quantum Computing: As with most traditional cryptographic algorithms, SHA-512 is potentially vulnerable to attacks from quantum computers, which are exponentially more powerful than classical computers. While this is a concern for long-term security, post-quantum cryptographic algorithms are being developed to address this challenge.

It's important to note that the advantages and disadvantages of SHA-512 should be considered in the context of specific use cases and requirements. While SHA-512 is widely used and provides strong security, it's essential to stay updated with the latest recommendations and advancements in cryptography to ensure the appropriate selection and use of hash functions.

Implementation:

RSA:

Problem Statement:

Emily and David are two individuals who want to communicate securely over an insecure network. They need a system that allows them to encrypt and decrypt messages using the RSA encryption algorithm. The system should generate the necessary keys, handle encryption and decryption, and ensure the integrity of the messages exchanged.

Additional consideration:

- Generate prime numbers p and q for Emily and David.

- Compute Emily's public and private keys based on p and q.
- Compute David's public and private keys based on p and q.
- Emily inputs a message she wants to send to David.
- Emily encrypts the message using David's public key.
- Emily sends the encrypted message to David.
- David decrypts the encrypted message using his private key.
- David receives the decrypted message and compares it with the original message to verify the integrity of the communication.

Code:

```
import random


def is_prime(num):
    # Check if a number is prime
    if num < 2:
        return False
    for i in range(2, int(num ** 0.5) + 1):
        if num % i == 0:
            return False
    return True


def generate_prime():
    # Generate a large prime number
    while True:
        num = random.randint(2 ** 10, 2 ** 11)
        if is_prime(num):
            return num


def compute_gcd(a, b):
    # Compute the greatest common divisor (GCD) of two numbers
    while b != 0:
        a, b = b, a % b
    return a


def compute_mod_inverse(e, phi_n):
    # Compute the modular multiplicative inverse of a number
    x, y, u, v = 0, 1, 1, 0
    while e != 0:
        q, r = phi_n // e, phi_n % e
        m, n = x - u * q, y - v * q
        phi_n, e, x, y, u, v = e, r, u, v, m, n
    return x % phi_n


def encrypt(message, public_key):
    # Encrypt the message using the public key
    e, n = public_key
    encrypted_message = [pow(ord(char), e, n) for char in message]
    return encrypted_message
```

```
def decrypt(encrypted_message, private_key):
    # Decrypt the encrypted message using the private key
    d, n = private_key
    decrypted_message = [chr(pow(char, d, n)) for char in
encrypted_message]
    return "".join(decrypted_message)


# Step 1: Generate two large prime numbers, p and q
p = generate_prime()
q = generate_prime()

# Step 2: Compute n = p * q
n = p * q

# Step 3: Compute the totient function of n, phi(n) = (p - 1) * (q - 1)
phi_n = (p - 1) * (q - 1)

# Step 4: Select an encryption exponent e, which is a positive integer
coprime to phi(n)
e = random.randint(2, phi_n)
while compute_gcd(e, phi_n) != 1:
    e = random.randint(2, phi_n)

# Step 5: Compute the decryption exponent d, which is the modular
multiplicative inverse of e modulo phi(n)
d = compute_mod_inverse(e, phi_n)

# Public key: (e, n)
public_key = (e, n)

# Private key: (d, n)
private_key = (d, n)

# The message to be encrypted
message = input("Enter the message: ")

# Encryption
encrypted_message = encrypt(message, public_key)
print("Encrypted message:", encrypted_message)

# Decryption
decrypted_message = decrypt(encrypted_message, private_key)
print("Decrypted message:", decrypted_message)
```

Step-by-step explanation of the RSA encryption algorithm process:

- Two large prime numbers, p and q, are generated using the generate_prime() function.
- The modulus n is computed as the product of p and q.
- The totient function phi(n) is computed as (p - 1) * (q - 1).

- An encryption exponent e is randomly selected between 2 and phi(n), ensuring it is coprime with phi(n).
- The decryption exponent d is computed using the compute_mod_inverse() function, which finds the modular multiplicative inverse of e modulo phi(n).
- The public key is represented as (e, n).
- The private key is represented as (d, n).
- The message to be encrypted, message, is defined.
- The encrypt() function takes the message and the public key as input, and encrypts the message by raising each character's Unicode value to the power of e modulo n.
- The encrypted message is stored in encrypted_message.
- The decrypt() function takes the encrypted message and the private key as input, and decrypts the message by raising each encrypted character to the power of d modulo n.
- The decrypted message is stored in decrypted_message.
- The encrypted message and decrypted message are printed to demonstrate the encryption and decryption process.

Security Analysis of RSA Algorithm Implementation:

- **Potential Attack Vectors and Countermeasures:**
  - Brute Force Attack: An attacker tries all possible private keys to decrypt the encrypted message. Countermeasure: Use sufficiently large prime numbers for p and q to make the key space too large for practical brute force attacks.
  - Factorization Attack: An attacker tries to factorize the modulus n to obtain the private key. Countermeasure: Choose large prime numbers p and q, and periodically regenerate new keys to mitigate the risk of factorization attacks.
  - Timing Attacks: An attacker measures the execution time of decryption operations to gather information about the private key. Countermeasure: Implement constant-time decryption algorithms to ensure that the execution time is independent of the decrypted value, making timing attacks harder.
  - Side Channel Attacks: Attackers exploit information leaked through physical characteristics (power consumption, electromagnetic

radiation) during encryption or decryption. Countermeasure: Implement countermeasures like power analysis, electromagnetic shielding, or randomizing the execution time to mitigate side channel attacks.

- Chosen Ciphertext Attacks: An attacker interacts with the encryption and decryption oracle to obtain information about the private key. Countermeasure: Implement proper padding schemes like OAEP (Optimal Asymmetric Encryption Padding) to thwart chosen ciphertext attacks.

- **Enhancing Security:**
  - Use longer key sizes: Increase the size of p and q to enhance the security against brute force and factorization attacks.
  - Regular key regeneration: Periodically regenerate new keys to mitigate the risk of key compromise due to advances in attack methods or compromised factors.
  - Secure key storage: Safely store the private key to prevent unauthorized access. Utilize secure hardware modules or encryption techniques for key protection.
  - Cryptographically secure random number generation: Ensure the use of a secure random number generator for generating prime numbers, as weak random numbers can weaken the security of the RSA algorithm.
  - Implement proper padding schemes: Apply robust padding schemes like OAEP to add randomness and prevent chosen ciphertext attacks.
  - Conduct security testing: Perform thorough security testing, including vulnerability assessments and penetration testing, to identify and mitigate potential vulnerabilities.

- **Limitations and Trade-Offs:**
  - Performance: RSA encryption and decryption operations can be computationally intensive, especially with larger key sizes. This may impact the performance of systems that heavily rely on RSA operations. Consider optimizing the implementation or utilizing hardware acceleration techniques if performance is a concern.
  - Key Management: RSA requires secure key management practices. Safeguarding the private key is crucial, as its compromise can lead

to the decryption of encrypted data. Implement secure key storage and access control mechanisms to mitigate risks.

- Key Size Considerations: While larger key sizes enhance security, they also increase computational overhead. It is essential to strike a balance between security requirements and performance considerations when choosing key sizes.
- Trust in Key Generation: The security of RSA heavily relies on the trustworthiness of the prime number generation process. Ensure the integrity and authenticity of the prime number generation algorithm and the randomness of the generated primes.

Reference:

- https://medium.com/@gowtham180502/implementing-rsa-algorithm-using-python-836f7da2a8e0
- https://www.rsa.com/
- https://cybernews.com/resources/what-is-aes-encryption/