CRYPTOGRAPHY ANALYSIS AND IMPLEMENTATION

NAME: ADHITHYA S D
REGNO: 20BCI0130
EMAIL: adhithya.sd2020@vitstudent.ac.in

Cryptography is the practice and study of techniques used to secure communication and information from unauthorised access or tampering. It involves the use of mathematical algorithms and principles to transform plaintext (unencrypted data) into ciphertext (encrypted data) in such a way that only authorised individuals or entities can decipher the information and convert it back to its original form.

The main objectives of cryptography are confidentiality, integrity, authentication, and non-repudiation. Confidentiality ensures that only authorised parties can access and understand the information, integrity ensures that the information remains unaltered and intact during transmission or storage, authentication verifies the identity of the sender or receiver, and non-repudiation prevents the sender from denying their involvement in the communication.

Cryptography relies on various cryptographic algorithms, such as symmetric key algorithms and asymmetric key algorithms. Symmetric key algorithms use the same key for both encryption and decryption, while asymmetric key algorithms utilise a pair of mathematically related keys: a public key for encryption and a private key for decryption.

1) SYMMETRIC KEY ALGORITHM:

A symmetric key algorithm is a type of cryptographic algorithm that uses the same key for both encryption and decryption of data. In other words, the sender and the receiver of the encrypted information share a common secret key. The security of a symmetric key algorithm relies heavily on the secrecy and confidentiality of the shared key. If an unauthorised party gains access to the key, they can decrypt the ciphertext and access the original information.

AES ALGORITHM:

AES (Advanced Encryption Standard) is a widely used symmetric key algorithm for encrypting and decrypting electronic data. It was established by the U.S. National Institute of Standards and Technology (NIST) in 2001 as a replacement for the ageing Data Encryption Standard (DES). AES is a block cipher, which means it operates on fixed-size blocks of data.

How the algorithm works:

AES operates on 128-bit blocks of data and supports three key sizes: 128-bit, 192-bit, and 256-bit. The algorithm consists of several rounds of substitution, permutation, and mixing operations, making it highly resistant to cryptanalysis attacks. The number of rounds performed depends on the key size: 10 rounds for 128-bit keys, 12 rounds for 192-bit keys, and 14 rounds for 256-bit keys.

During each round, the data undergoes a series of transformations, including substitution using a substitution box (S-box), permutation through a shift rows operation, mixing through a mix columns operation, and XORing with a round key derived from the original encryption key. These operations scramble the data, introducing confusion and diffusion, which enhances the security of the encrypted information.

Key strengths and advantages:

- Security: AES is considered highly secure and has withstood extensive cryptanalysis by the global cryptographic community. Its design and implementation have been rigorously analysed, making it a trusted choice for securing sensitive data.
- Efficiency: AES is computationally efficient and can be implemented in hardware or software with reasonable performance, even on resource-constrained devices.
- Versatility: AES supports multiple key sizes, allowing users to choose the appropriate level of security based on their specific requirements.
- Standardisation: AES has been adopted as a global standard, which promotes interoperability and ensures that implementations from different vendors can communicate securely.

Known vulnerabilities or weaknesses:

No practical vulnerabilities have been discovered in AES. However, any cryptographic algorithm is subject to potential attacks in the future as technology advances and new attack techniques are developed. Users should always ensure they are using the latest versions and best practices for implementation and key management.

Real-world examples of AES usage:

AES is extensively used in various applications and industries,

- Secure communication: AES is commonly used to encrypt network traffic, such as in virtual private networks (VPNs) and secure socket layer (SSL) protocols for secure web browsing.

- Data storage: AES is employed to encrypt data at rest, such as on hard drives, USB drives, and cloud storage services, ensuring that sensitive information remains confidential even if the storage medium is compromised.
- Wireless security: AES is the encryption standard for securing Wi-Fi networks. It is used in the Wi-Fi Protected Access (WPA2) protocol, which is widely implemented in routers and access points to protect wireless communications.
- Financial transactions: AES is utilized in secure online banking and payment systems to protect sensitive financial data during transmission.
- Government and military applications: AES is approved for use by government agencies and the military for securing classified information.

2) ASYMMETRIC KEY ALGORITHM:

An asymmetric key algorithm, also known as public-key cryptography, is a cryptographic system that uses a pair of mathematically related keys for encryption and decryption. Unlike symmetric key algorithms, where the same key is used for both operations, asymmetric key algorithms use two distinct keys: a public key and a private key. The security of an asymmetric key algorithm lies in the mathematical relationship between the public and private keys. The encryption process performed with the public key can only be reversed using the corresponding private key, which is kept secret. This property enables secure communication and authentication.

RSA ALGORITHM:

RSA (Rivest-Shamir-Adleman) is a widely used asymmetric key algorithm for encryption, decryption, and digital signatures. It was invented by Ron Rivest, Adi Shamir, and Leonard Adleman in 1977 and remains one of the most popular and trusted public-key encryption algorithms.

How the algorithm works:

The RSA algorithm involves the following steps:

- Key Generation: The user generates a key pair consisting of a public key and a private key. The keys are mathematically related but computationally impractical to derive one from the other. The public key is shared with others, while the private key is kept secret.

- Encryption: To encrypt a message using RSA, the sender converts the plaintext message into a numeric representation and applies the recipient's public key. The encryption process involves modular exponentiation, where the plaintext is raised to the power of the recipient's public key, and the result is reduced modulo the recipient's public modulus.

- Decryption: The recipient uses their private key to decrypt the encrypted message. They apply modular exponentiation to the ciphertext, raising it to the power of their private key, and reduce the result modulo their private modulus.

- Digital Signatures: RSA can also be used for digital signatures. The sender uses their private key to sign a message by applying modular exponentiation, and the recipient can verify the signature using the sender's public key.

Key strengths and advantages of RSA:

- Security: RSA is based on the difficulty of factoring large integers into their prime factors. The security of RSA relies on the mathematical problem of prime factorization, which is computationally infeasible for large numbers.

- Asymmetric Cryptography: RSA allows for secure communication and authentication without the need for a shared secret key. The public key can be freely distributed, while the private key remains confidential.

- Versatility: RSA can be used for encryption, decryption, and digital signatures, making it a versatile algorithm for a wide range of cryptographic operations.

- Widely Accepted: RSA has been extensively analyzed and standardized, making it widely accepted and implemented in various systems and applications.

Known vulnerabilities or weaknesses:

- Key Length: The strength of RSA depends on the size of the key used. As computing power increases, longer key lengths are required to maintain the same level of security. Weaknesses can arise if shorter key lengths are used or if implementation flaws exist.

- Timing Attacks: Timing attacks can potentially exploit the variations in execution time during the RSA algorithm's modular exponentiation operations, revealing information about the private key. Countermeasures, such as constant-time implementations, can mitigate this vulnerability.

Real-world examples of RSA usage:

- Secure Communication: RSA is widely used in secure email protocols (e.g., PGP/GPG), secure web browsing (TLS/SSL), Virtual Private Networks (VPNs), and secure messaging applications.

- Digital Signatures: RSA is utilized for generating and verifying digital signatures in various applications, including software updates, certificate authorities, and secure document signing.

- Key Exchange: RSA is used in key exchange protocols, such as the Diffie-Hellman key exchange, to establish secure shared secret keys between parties.

- Secure Shell (SSH): RSA is commonly used for secure remote login and file transfer in SSH protocols.

- Encryption of Data at Rest: RSA is often employed in encrypting symmetric keys used for data encryption algorithms like AES. This allows secure storage and transmission of symmetric keys.

RSA remains a fundamental and widely adopted asymmetric key algorithm due to its security, versatility, and established standardization. However, it is crucial to use appropriate key lengths and follow best practices to ensure its effectiveness in the face of evolving computational capabilities and potential vulnerabilities.

3) HASH FUNCTIONS:

A hash function is a mathematical function that takes an input (or "message") and produces a fixed-size string of characters, which is typically a hash value or digest. The primary purpose of a hash function is to efficiently transform input data into a unique representation, called a hash code or hash value. Hash functions are commonly used in various fields, including cryptography, data structures, and data integrity verification. It's important to note that while hash functions provide data integrity and other useful properties, they are not encryption mechanisms. Hash functions are one-way functions, meaning they are designed for data integrity and verification rather than data confidentiality. Commonly used hash functions include MD5 (Message Digest Algorithm 5), SHA-1 (Secure Hash Algorithm 1), SHA-256, and SHA-3.

SHA-256:

SHA-256 (Secure Hash Algorithm 256-bit) is a widely used cryptographic hash function. It belongs to the SHA-2 (Secure Hash Algorithm 2) family, which includes other variants like SHA-224, SHA-256, SHA-384, and SHA-512. SHA-256 is designed to produce a fixed-size 256-bit hash value, providing a high level of data integrity and security.

How the algorithm works:

The SHA-256 algorithm operates on a block of input data and performs the following steps:

- Padding: The input message is padded to meet the specific block size requirements of SHA-256. Padding ensures that the input length is a multiple of the block size.

- Message Expansion: The padded message is divided into blocks, and each block undergoes message expansion. The algorithm processes each block iteratively, updating the internal state based on the previous block's hash value.

- Compression: The message expansion is followed by a compression function, which combines the internal state with the current block's data. This process involves several bitwise operations, modular addition, and logical functions.

- Final Hash: After processing all the blocks, the resulting internal state is combined to produce the final hash value, which is a fixed-size 256-bit string.

Key strengths and advantages of SHA-256:

- Security: SHA-256 is designed to provide a high level of security against collision and pre-image attacks. The 256-bit hash size makes it computationally infeasible to find two different inputs that produce the same hash value.

- Data Integrity: SHA-256 is widely used for data integrity verification. By comparing the hash value of the original data with the calculated hash value, one can determine if the data has been tampered with or modified.

- Efficiency: While SHA-256 provides a strong level of security, it is still computationally efficient. It can process large amounts of data relatively quickly, making it suitable for various applications.

- Standardization: SHA-256 is a standardized algorithm that has undergone extensive analysis and review by the cryptographic community. Its well-established status and widespread adoption contribute to its trustworthiness.

Known vulnerabilities or weaknesses:

No practical vulnerabilities have been identified in SHA-256 that compromise its security. Also, we have to note that SHA-256 is a hash function and not a full-fledged encryption mechanism, so it doesn't provide data confidentiality.

Real-world examples of SHA-256 usage:

- Cryptocurrency: SHA-256 is extensively used in cryptocurrencies like Bitcoin. It is employed in the process of mining, where it serves as the basis for generating a hash value that meets certain criteria.

- Digital Signatures: SHA-256 is utilized in digital signature schemes, where it generates a hash value of the document or message that is then encrypted with a private key to create a digital signature.

- Password Storage: SHA-256, combined with additional techniques like salting, is used to securely store passwords in databases or authentication systems. The hash of a user's password is stored instead of the actual password itself.

- Certificate Authorities: SHA-256 is used to generate and verify digital certificates, ensuring the integrity and authenticity of the certificate information.

- Secure File Transfer: SHA-256 can be used to verify the integrity of files during transfer or storage, ensuring that the received files match the original.

SHA-256 is a widely adopted and trusted cryptographic hash function that offers strong security and data integrity. Its usage spans various domains, including cryptocurrencies, digital signatures, password storage, and secure file transfer, among others.

Here is a simple python code to demonstrate SHA-256:

Code:

```
import hashlib

message = input("Enter a string to hash: ")
hash_val = hashlib.sha256(message.encode("UTF-8")).hexdigest()
print(hash_val)
```

Output:

```
Enter a string to hash: hello
2cf24dba5fb0a30e26e83b2ac5b9e29e1b161e5c1fa7425e73043362938b9824
```

```
Enter a string to hash: smartbridge
5161892fe5a293ea2b60786dfc454e120f16c987c8dce932beab7ddde9bcd60b
```

```
Enter a string to hash: Smartbridge
4dab5304cb3f2e5bf09165a5aa8a6884a6e3c12d15038ca23483a72ad4f52c99
```

IMPLEMENTATION:

For implementation I'm choosing RSA Algorithm. So I will implement RSA Algorithm using python.

Scenario/Problem:

The scenario aims to demonstrate the implementation of the RSA encryption algorithm. RSA is a widely used cryptographic algorithm for secure communication and data encryption. It involves generating public and private keys based on the mathematical properties of large prime numbers.

Step-by-step implementation:

- Import the math module to use mathematical functions.
- Define a function modinv(a, m) to calculate the modular inverse of a modulo m. It uses a brute-force approach to find a value x such that (a * x) % m == 1. If no modular inverse exists, it returns -1.
- Prompt the user to enter two prime numbers p and q. These prime numbers are used to generate the public and private keys.
- Calculate the value of n by multiplying p and q. n is part of both the public and private keys.
- Calculate the totient of n (represented as totient) using (p - 1) * (q - 1). The totient is required for key generation.
- Iterate over possible values of a from 2 to totient - 1. For each a, check if it is coprime with totient (i.e., their greatest common divisor is 1) using math.gcd(a, totient) == 1. If a is coprime, print the corresponding encryption key (a, n) and decryption key (modinv(a, totient), n).

- Prompt the user to choose a value for the encryption key e from the list of possible values shown in the previous step.

- Prompt the user to enter the message they want to encrypt. The message is converted to uppercase for simplicity.

- Encrypt the message by iterating over each character ch in the message. Convert the character to its corresponding ASCII value using ord(ch). Apply the encryption formula (((ord(ch) - 65) ** e) % n) to obtain the encrypted value x. Print each encrypted value x.

- Calculate the decryption key d by finding the modular inverse of e modulo totient using the modinv function.

- Initialize an empty string decrypt to store the decrypted message.

- Decrypt each character in the encrypted message by iterating over each character ch in the message. Convert the character to its corresponding ASCII value using ord(ch). Apply the decryption formula (((x ** d) % n) % n) to obtain the decrypted value y. Convert y back to its corresponding ASCII character using chr(y % 26 + 65). Print the intermediate values x, y, and the decrypted character.

- Append the decrypted character to the decrypt string.

- Finally, print the decrypted message stored in the decrypt string.

Code:

```
import math

def modinv(a,m):
    for x in range(1,m):
        if(((a%m) * (x%m)) % m == 1):
            return x
    return -1



p = int(input("Enter the value of p(must be a prime number): "))
q = int(input("Enter the value of q(must be a prime number): "))
n = p*q
print("n = ",n)
totient = (p-1)*(q-1)
print("Totient = ",totient)
```

```
print("\nAll possible Public keys and Private keys\n")
for a in range(2,totient):
    if(math.gcd(a,totient) == 1):
        print("Encryption key(e,n) = ",(a,n),"  <---->  Decryption key(d,n) =
",(modinv(a,totient),n))

e = int(input("Choose a value for e from the above for encryption: "));

message = input("Enter the message: ").upper()


print("\nThe Encrypted message in a sequence of numbers: ")
for ch in message:
    x = (((ord(ch)-65)**e) % n)
    print(x)


d = modinv(e,totient)
decrypt = ""
print("\nDecryption:")
for ch in message:
     x = (((ord(ch)-65)**e) % n)
     y = ((x**d) % n) % n
     print(x,"-->",y,"-->",chr(y%26 +65))
     decrypt += chr(y%26 +65)
print("\nThe Decrypted message is: ",decrypt)
```

Output:

To test the implementation, we follow these steps:

- Provide two prime numbers for p and q when prompted.
- The program will generate a list of possible encryption and decryption keys. Choose a value for e from the list.
- Enter a message to encrypt.
- The program will print the encrypted values of each character.
- The program will then decrypt the message and print the intermediate steps along with the final decrypted message.

- Verify that the decrypted message matches the original input message, indicating the successful implementation of RSA encryption and decryption.

```
Enter the value of p(must be a prime number): 11

Enter the value of q(must be a prime number): 13
n =  143
Totient =  120

All possible Public keys and Private keys

Encryption key(e,n) =  (7, 143)   <---->  Decryption key(d,n) =  (103, 143)
Encryption key(e,n) =  (11, 143)   <---->  Decryption key(d,n) =  (11, 143)
Encryption key(e,n) =  (13, 143)   <---->  Decryption key(d,n) =  (37, 143)
Encryption key(e,n) =  (17, 143)   <---->  Decryption key(d,n) =  (113, 143)
Encryption key(e,n) =  (19, 143)   <---->  Decryption key(d,n) =  (19, 143)
Encryption key(e,n) =  (23, 143)   <---->  Decryption key(d,n) =  (47, 143)
Encryption key(e,n) =  (29, 143)   <---->  Decryption key(d,n) =  (29, 143)
Encryption key(e,n) =  (31, 143)   <---->  Decryption key(d,n) =  (31, 143)
Encryption key(e,n) =  (37, 143)   <---->  Decryption key(d,n) =  (13, 143)
Encryption key(e,n) =  (41, 143)   <---->  Decryption key(d,n) =  (41, 143)
Encryption key(e,n) =  (43, 143)   <---->  Decryption key(d,n) =  (67, 143)
Encryption key(e,n) =  (47, 143)   <---->  Decryption key(d,n) =  (23, 143)
Encryption key(e,n) =  (49, 143)   <---->  Decryption key(d,n) =  (49, 143)
Encryption key(e,n) =  (53, 143)   <---->  Decryption key(d,n) =  (77, 143)
Encryption key(e,n) =  (59, 143)   <---->  Decryption key(d,n) =  (59, 143)
Encryption key(e,n) =  (61, 143)   <---->  Decryption key(d,n) =  (61, 143)
Encryption key(e,n) =  (67, 143)   <---->  Decryption key(d,n) =  (43, 143)
Encryption key(e,n) =  (71, 143)   <---->  Decryption key(d,n) =  (71, 143)
Encryption key(e,n) =  (73, 143)   <---->  Decryption key(d,n) =  (97, 143)
Encryption key(e,n) =  (77, 143)   <---->  Decryption key(d,n) =  (53, 143)
Encryption key(e,n) =  (79, 143)   <---->  Decryption key(d,n) =  (79, 143)
Encryption key(e,n) =  (83, 143)   <---->  Decryption key(d,n) =  (107, 143)
Encryption key(e,n) =  (89, 143)   <---->  Decryption key(d,n) =  (89, 143)
Encryption key(e,n) =  (91, 143)   <---->  Decryption key(d,n) =  (91, 143)
Encryption key(e,n) =  (97, 143)   <---->  Decryption key(d,n) =  (73, 143)
Encryption key(e,n) =  (101, 143)   <---->  Decryption key(d,n) =  (101, 143)
```

```
Encryption key(e,n) =  (97, 143)   <---->  Decryption key(d,n) =  (73, 143)
Encryption key(e,n) =  (101, 143)   <---->  Decryption key(d,n) =  (101, 143)
Encryption key(e,n) =  (103, 143)   <---->  Decryption key(d,n) =  (7, 143)
Encryption key(e,n) =  (107, 143)   <---->  Decryption key(d,n) =  (83, 143)
Encryption key(e,n) =  (109, 143)   <---->  Decryption key(d,n) =  (109, 143)
Encryption key(e,n) =  (113, 143)   <---->  Decryption key(d,n) =  (17, 143)
Encryption key(e,n) =  (119, 143)   <---->  Decryption key(d,n) =  (119, 143)

Choose a value for e from the above for encryption: 107

Enter the message: hello

The Encrypted message in a sequence of numbers:
28
49
110
110
53

Decryption:
28 --> 7 --> H
49 --> 4 --> E
110 --> 11 --> L
110 --> 11 --> L
53 --> 14 --> O

The Decrypted message is:  HELLO
```

SECURITY ANALYSIS:

Potential threats or vulnerabilities:

a. Brute Force Attack: The implementation of the modular inverse function (modinv) using a brute force approach makes it susceptible to brute force attacks. An attacker could potentially iterate through all possible values of x to find the modular inverse, although the time complexity would depend on the size of the modulus m.

b. Prime Number Selection: The security of RSA relies on the selection of large prime numbers p and q. If weak or easily factorable primes are chosen, the security of the encryption can be compromised.

c. Side-channel Attacks: The code provided does not address potential side-channel attacks, such as timing attacks or power analysis attacks. These attacks exploit information leaked through side channels like execution time or power consumption to infer sensitive information.

Countermeasures or best practices:

a. Enhanced Key Generation: Implement a more robust key generation mechanism that includes techniques like prime number generation algorithms (e.g., Miller-Rabin primality test), prime number length requirements, and secure random number generation.

b. Modular Inverse Optimization: Instead of using a brute force approach for finding the modular inverse, utilize more efficient algorithms like the extended Euclidean algorithm or the binary extended Euclidean algorithm.

c. Side-channel Attack Mitigation: Implement countermeasures against side-channel attacks, such as randomizing execution times, introducing dummy computations, or using constant-time algorithms.

Limitations and trade-offs:

a. Limited Input Validation: The code does not perform extensive input validation. It assumes that the user will input prime numbers and valid choices for encryption keys. Adding input validation and error handling is important to ensure the code behaves as expected and to prevent potential vulnerabilities.

b. Key Management: The code does not address key management aspects, such as secure storage of private keys, key rotation, or key exchange protocols. These considerations are crucial for the overall security of an RSA system.

c. Encryption Only: The provided code focuses on encryption and decryption of a single message. In practice, secure communication often involves additional layers, such as hybrid encryption, integrity checks, and authentication mechanisms. These aspects are not covered in the given implementation.

CONCLUSION:

Cryptography plays a vital role in cybersecurity and ethical hacking by providing secure communication, data integrity, and authentication. AES, RSA, and SHA-256 are widely adopted cryptographic algorithms with proven security properties. While these algorithms have their strengths, they also have vulnerabilities that need to be mitigated through proper implementation and adherence to best practices. The practical implementation of AES demonstrated the encryption and decryption of a file, highlighting the importance of key management and protection against potential attacks. Overall, cryptography remains a crucial tool in safeguarding sensitive information and ensuring the security of digital systems.