CYBER SECURITY AND ETHICAL HACKING

Assignment -3

NAME : NITHIYASRI.M

REG NO: 20BCI0230

Assignment: Cryptography Analysis and Implementation

Objective: The objective of this assignment is to analyze cryptographic algorithms and implement them in a practical scenario.

Instructions:

Research: Begin by conducting research on different cryptographic algorithms such as symmetric key algorithms (e.g., AES, DES), asymmetric key algorithms (e.g., RSA, Elliptic Curve Cryptography), and hash functions (e.g., MD5, SHA-256). Understand their properties, strengths, weaknesses, and common use cases.

Analysis: Choose three cryptographic algorithms (one symmetric, one asymmetric, and one hash function) and write a detailed analysis of each. Include the following points in your

Briefly explain how the algorithm works.

Discuss the key strengths and advantages of the algorithm.

Identify any known vulnerabilities or weaknesses.

Provide real-world examples of where the algorithm is commonly used.

Implementation:

Select one of the cryptographic algorithms you analyzed and implement it in a practical scenario. You can choose any suitable programming language for the implementation.

Clearly define the scenario or problem you aim to solve using cryptography.

Provide step-by-step instructions on how you implemented the chosen algorithm.

Include code snippets and explanations to demonstrate the implementation.

Test the implementation and discuss the results.

Security Analysis:

Perform a security analysis of your implementation, considering potential attack vectors and countermeasures.

Identify potential threats or vulnerabilities that could be exploited.

Propose countermeasures or best practices to enhance the security of your implementation.

Discuss any limitations or trade-offs you encountered during the implementation process.

Conclusion: Summarize your findings and provide insights into the importance of cryptography in cybersecurity and ethical hacking.

Submission Guidelines:

Prepare a well-structured report that includes the analysis, implementation steps, code snippets, and security analysis.

Use clear and concise language, providing explanations where necessary.

Include any references or sources used for research and analysis.

Compile all the required files (report, code snippets, etc.) into a single zip file for submission.

SYMMETRIC ALGORITHM: AES

Analysis:

AES stands for Advanced Encryption Standard. It is a widely used symmetric encryption algorithm that is considered secure for protecting sensitive data. AES was selected by the U.S. National Institute of Standards and Technology (NIST) in 2001 as the successor to the Data Encryption Standard (DES).

AES operates on blocks of data and uses a fixed block size of 128 bits and key sizes of 128, 192, or 256 bits. It employs a series of mathematical operations, including substitution, permutation, and mixing of bits, to transform the input data into ciphertext. The encryption and decryption \processes use a secret key that must be known by the authorized parties.

How algorithm works:

The AES algorithm operates on blocks of data and consists of several rounds of transformations. Here's a high-level overview of how AES works:

1. Key Expansion: The AES algorithm takes a secret key of 128, 192, or 256 bits and expands it into a set of round keys. The key expansion process derives a series of subkeys that will be used in each round of encryption and decryption.
2. Initial Round: The input block of plaintext is XORed with the first round key.
3. Rounds: AES consists of multiple rounds, the number of which depends on the key size. Each round consists of four main transformations applied to the data block:

a. SubBytes: Each byte in the block is replaced with a corresponding byte from the AES S-box, a predefined substitution table.

b. ShiftRows: The bytes in each row of the block are shifted cyclically to the left. The first row is not shifted, the second row is shifted by one byte, the third row by two bytes, and the fourth row by three bytes.

c. MixColumns: Each column of the block is transformed using a matrix multiplication. This step provides diffusion and achieves confusion in the encryption process.

d. AddRoundKey: The block is XORed with the round key for the current round.

4. Final Round: The final round is similar to the previous rounds, except that the MixColumns step is omitted.
5. Output: After the final round, the resulting block is the ciphertext.

For decryption, the AES algorithm performs similar operations in reverse. The round keys are used in the reverse order, and the inverse transformations of SubBytes, ShiftRows, and MixColumns are applied. The AddRoundKey step involves XORing the block with the round key for the corresponding round in reverse order.

By iterating through multiple rounds and using a combination of substitutions, permutations, and XOR operations, AES achieves a high level of security and provides confidentiality for the encrypted data.

Strength of AES:

The strength of an AES key refers to the level of security it provides against attacks, particularly brute-force attacks. The AES algorithm supports key sizes of 128, 192, and 256 bits. The larger the key size, the stronger the encryption and the more resistant it is to brute-force attacks.

Here's a brief overview of the key strengths associated with different AES key sizes:

1. AES-128: This uses a 128-bit key and provides a high level of security. The key space for AES-128 is 2^128, which means there are approximately 3.4 x 10^38 possible keys. Brute-forcing this key size is computationally infeasible with current technology, making it highly secure.
2. AES-192: This uses a 192-bit key and offers a stronger security margin compared to AES-128. The key space for AES-192 is 2^192, which is significantly larger than that of AES-128. Brute-forcing this key size is even more computationally challenging and remains infeasible with current technology.

3. AES-256: This uses a 256-bit key and is considered the most secure option among the three key sizes. The key space for AES-256 is 2^256, which is an astronomically large number. The computational resources required to brute-force this key size are currently beyond the capabilities of existing technology.

It's important to note that the security of AES not only depends on the key size but also on proper key management practices, including generating random and unpredictable keys, securely storing and transmitting keys, and regularly updating keys when necessary. Additionally, the strength of the overall cryptographic system also relies on the security of the implementation, protection against side-channel attacks, and proper usage of the algorithm.

Advantage of AES:

AES (Advanced Encryption Standard) has several advantages that contribute to its widespread adoption and usage. Here are some key advantages of AES:

1. Security: AES is a highly secure encryption algorithm when used correctly with appropriate key sizes and key management practices. It has undergone extensive analysis and scrutiny by cryptographic experts, and no practical vulnerabilities have been found that significantly weaken its security. AES has been adopted by governments, financial institutions, and organizations worldwide as a trusted encryption standard.
2. Efficiency: AES is designed to be efficient in terms of both computational resources and memory usage. It has been optimized for modern computer architectures, making it computationally efficient on a wide range of devices, including desktop computers, mobile devices, and embedded systems.
3. Standardization: AES is an internationally recognized and widely adopted encryption standard. It has been standardized by the U.S. National Institute of Standards and Technology (NIST) and has gained acceptance and implementation support from various organizations and software libraries globally. This standardization ensures interoperability and compatibility across different systems and applications.
4. Flexibility: AES supports key sizes of 128, 192, and 256 bits, allowing users to choose the level of security that meets their specific requirements. It can be easily integrated into various software and hardware platforms and can be used for a wide range of applications, including securing communications, protecting data at rest, and ensuring the integrity of stored information.
5. Performance: AES has been optimized to provide high-performance encryption and decryption. With efficient implementations, AES can process large amounts of data quickly, making it suitable for real-time applications, such as network encryption, video streaming, and disk encryption.
6. Trustworthy Algorithm: AES was selected through a rigorous process that involved public scrutiny and evaluation by the cryptographic community. The selection criteria included security, efficiency, and suitability for various applications. The

transparency and openness of the selection process contribute to the trustworthiness of the AES algorithm.

Overall, AES combines strong security, efficiency, standardization, and flexibility, making it a reliable and widely used encryption algorithm for protecting sensitive data in various domains. Its advantages have made it the de facto standard for symmetric encryption and have contributed to its continued dominance in the field of cryptography.

Disadvantage of AES:

While AES (Advanced Encryption Standard) is a highly secure and widely adopted encryption algorithm, it also has a few limitations and potential disadvantages. Here are some of the notable disadvantages of AES:

1. Key Management: AES, like any encryption algorithm, relies on the proper management of encryption keys. Key distribution, storage, and protection are crucial aspects of AES implementation. If encryption keys are compromised or mishandled, the security of the encrypted data can be compromised. Key management can be a challenging task, especially in large-scale systems or when dealing with multiple encryption keys.
2. Resource Intensive: While AES is considered efficient in terms of computational resources, it still requires a certain amount of processing power and memory to perform encryption and decryption operations. In resource-constrained environments, such as embedded systems or low-power devices, the computational overhead of AES may pose challenges and impact performance.
3. Lack of Forward Secrecy: AES does not provide forward secrecy. Forward secrecy ensures that if an encryption key is compromised in the future, it does not compromise the security of previously encrypted data. In other words, compromising a current AES key can retroactively decrypt previously encrypted data if the attacker gains access to the key.
4. Limited Authentication: AES itself is a symmetric encryption algorithm, which means it only provides confidentiality, not authentication or integrity verification of the encrypted data. Additional mechanisms, such as digital signatures or message authentication codes (MACs), need to be employed alongside AES to ensure the integrity and authenticity of the data.
5. Vulnerability to Side-Channel Attacks: AES implementations can be vulnerable to side-channel attacks. Side-channel attacks exploit information leaked during the encryption process, such as power consumption, timing, or electromagnetic radiation. Techniques such as differential power analysis (DPA) or timing attacks can potentially reveal sensitive information about the encryption keys, undermining the security of AES.

It's important to note that these disadvantages can often be mitigated through proper implementation practices, secure key management, and the use of additional cryptographic mechanisms. Despite these limitations, AES remains widely used and trusted for its strong security properties and efficient performance

Real world examples:

1. Secure Communication: AES is commonly used in secure communication protocols such as SSL/TLS (Secure Sockets Layer/Transport Layer Security). When you access a website using HTTPS, AES encryption is used to protect the data transmitted between your web browser and the website's server.
2. File and Disk Encryption: AES is utilized in software applications that offer file and disk encryption. These tools allow users to encrypt their files, folders, or entire disks, ensuring that the data remains protected even if it is stolen or accessed by unauthorized individuals.
3. Virtual Private Networks (VPNs): AES is a vital component in VPN technologies that enable secure remote access to private networks. When you establish a VPN connection, AES encryption is employed to safeguard the data transferred between your device and the VPN server, ensuring privacy and preventing eavesdropping.
4. Wireless Networks: AES is used in securing wireless networks, particularly Wi-Fi networks. The WPA2 (Wi-Fi Protected Access 2) protocol, which is widely implemented in Wi-Fi routers and devices, incorporates AES encryption to safeguard wireless communication.
5. Data Storage: AES encryption is utilized in various storage devices, such as USB drives and external hard drives, to protect the stored data. It ensures that the information remains encrypted and inaccessible without the appropriate decryption key.
6. Messaging Apps: Many popular messaging apps incorporate AES encryption to ensure the privacy and security of user communications. For example, Signal and WhatsApp employ end-to-end encryption that utilizes AES to secure messages exchanged between users.
7. Payment Systems: AES encryption is used in secure online payment systems to protect financial transactions. When you make a purchase online using your credit card, AES encryption is employed to encrypt the sensitive payment information, preventing unauthorized access.

ASYMMETRIC ALGORITHM:

RSA ALGORITHM

RSA is an asymmetric encryption algorithm used for secure communication and data encryption. It involves generating a public-private key pair, with the public key used for encryption and the private key for decryption. The security of RSA is based on the difficulty of factoring large composite numbers. It is widely used in secure communication protocols, digital signatures, and key exchange protocols. RSA provides a secure way to protect sensitive information and ensure data confidentiality.

HOW RSA WORKS:

RSA encryption works through a series of mathematical operations involving modular arithmetic and the properties of prime numbers. Here's a simplified explanation of how RSA encryption works:

1. Key Generation: First, two large prime numbers, p and q, are selected. The product of these primes, $n = p * q$, forms the modulus for RSA. A value called the totient function, $\varphi(n) = (p-1) * (q-1)$, is also calculated.
2. Public and Private Key Pair: A public key (e, n) is generated, where e is a value that is relatively prime to $\varphi(n)$. The public key is shared with others. The private key (d, n) is calculated, where d is the modular multiplicative inverse of e modulo $\varphi(n)$. The private key must be kept secret.
3. Encryption: To encrypt a message M, it is transformed into a numerical representation. Each character or block of characters is assigned a unique number. The sender then uses the recipient's public key (e, n) and performs the encryption operation C = M^e mod n. C is the encrypted ciphertext.
4. Decryption: The recipient, possessing the private key (d, n), performs the decryption operation M = C^d mod n. The result is the original message M.

The security of RSA relies on the difficulty of factoring the large composite number n into its prime factors p and q. Since the private key is derived from the factorization of n, an attacker would need to factorize n to obtain the private key, which is considered computationally infeasible for sufficiently large prime numbers.

It's important to note that while RSA encryption provides security, it can be computationally expensive, especially for larger key sizes. As a result, RSA is often used in conjunction with symmetric encryption algorithms like AES, where RSA is employed for key exchange or digital signatures, and a symmetric algorithm is used for efficient encryption of data.

RSA KEY STRENGTH:

The strength of RSA encryption relies on the size of the key used. The security of RSA is based on the difficulty of factoring the large composite number into its prime factors. The larger the prime numbers used in the key generation process, the more secure the RSA encryption becomes.

The strength of an RSA key is typically measured in terms of its key size, which is the number of bits in the modulus. The larger the key size, the more secure the RSA encryption is against attacks.

As of my knowledge cutoff in September 2021, commonly recommended RSA key sizes for secure encryption are as follows:

- 2048 bits: This key size is considered secure for most applications and is widely used.
- 3072 bits: This key size provides an extra level of security and is recommended for sensitive applications.
- 4096 bits: This key size offers a higher level of security and is often used in situations where strong security is required.

It's important to note that as computing power advances, the recommended key sizes may change to maintain the same level of security. It is crucial to stay informed about current recommendations from cryptographic experts and standards organizations.

ADVANTAGE OF RSA:

RSA encryption offers several advantages that contribute to its widespread use in secure communication and data encryption:

1. Asymmetric Encryption: RSA is an asymmetric encryption algorithm, which means it uses a pair of keys for encryption and decryption. This allows for secure communication without the need to exchange a shared secret key, simplifying key management.
2. Secure Key Exchange: RSA can be used for secure key exchange protocols like Diffie-Hellman. It enables two parties to securely establish a shared secret key over an insecure communication channel, ensuring confidentiality and preventing eavesdropping.
3. Digital Signatures: RSA is widely used for creating and verifying digital signatures. It allows the recipient to verify the authenticity and integrity of electronic documents or messages, ensuring they haven't been tampered with during transit.
4. Strong Security: The security of RSA encryption is based on the difficulty of factoring large composite numbers. As long as the prime factors remain unknown, it is computationally infeasible to break the encryption. With a sufficiently large key size, RSA encryption provides strong protection against unauthorized decryption.

5. Broad Application Support: RSA has gained widespread adoption and is supported by numerous software libraries, protocols, and cryptographic hardware. This ensures compatibility and interoperability across various platforms and systems.
6. Well-Studied and Established: RSA encryption has been extensively studied and analyzed by the cryptographic community for several decades. Its security and algorithms have been subjected to rigorous scrutiny, making it a trusted and well-established encryption method.
7. Performance Trade-Off: While RSA encryption can be computationally expensive, especially for larger key sizes, its performance can be optimized by using hybrid encryption schemes. This involves using RSA for key exchange or digital signatures, while employing symmetric encryption algorithms (like AES) for efficient encryption of data.

DISADVANTAGE OF RSA:

While RSA encryption has numerous advantages, it also has certain limitations and disadvantages:

1. Key Size and Performance: RSA encryption with larger key sizes provides stronger security. However, larger key sizes also require more computational resources and can result in slower encryption and decryption processes. This can be a disadvantage in scenarios where efficiency and speed are critical, such as high-volume network communication or resource-constrained devices.
2. Key Management: RSA encryption requires the generation, distribution, and storage of public and private keys. Managing and securely storing these keys can be challenging, especially in large-scale systems or scenarios where key rotation or revocation is necessary. Key management can become complex, requiring secure key storage mechanisms and robust protocols for key exchange.
3. Vulnerability to Quantum Computing: RSA encryption, like most traditional public-key cryptographic algorithms, is vulnerable to attacks by quantum computers. Quantum computers, if they reach sufficient scale and capability, could potentially factor large numbers efficiently, compromising the security of RSA encryption. As a result, post-quantum cryptography algorithms are being actively researched and developed as potential replacements for RSA in the future.
4. Lack of Forward Secrecy: RSA encryption does not provide forward secrecy, meaning that if an attacker obtains the private key, they can decrypt all previously intercepted or recorded encrypted data. This is in contrast to symmetric encryption algorithms, where compromised keys only affect the confidentiality of future communications.
5. Computational Complexity: The encryption and decryption operations in RSA involve modular exponentiation, which can be computationally intensive for large key

sizes. This complexity can pose challenges for resource-constrained devices, such as embedded systems or mobile devices with limited processing power.

6. Key Length Considerations: The choice of key length in RSA is critical to ensure security. As computing power advances and new attacks are discovered, longer key lengths may be required to maintain the same level of security. This means that RSA implementations need to regularly assess and potentially update key lengths to stay resilient against emerging threats.

REAL-WORLD EXAMPLE:

1. Key Exchange: During the initial handshake between your browser and the server, RSA encryption is used for secure key exchange. The server generates an RSA key pair, consisting of a public key and a private key. The server's public key is then sent to your browser.
2. Encryption: Your browser generates a symmetric session key (e.g., using AES) for efficient encryption of the actual data exchanged during the session. The session key is then encrypted using the server's public key and sent back to the server.
3. Decryption: The server, possessing the corresponding private key, decrypts the session key using RSA decryption. This allows the server to obtain the session key securely.
4. Secure Communication: With the session key now shared securely between your browser and the server, subsequent data exchanged during the session, such as web pages, forms, or financial transactions, is encrypted using the symmetric session key (e.g., using AES). This ensures the confidentiality and integrity of the transmitted data.

HASH FUNCTION:

SHA-256

SHA-256 (Secure Hash Algorithm 256-bit) is a cryptographic hash function that belongs to the SHA-2 family. It takes an input message and produces a fixed-size 256-bit hash value, representing a unique digital fingerprint of the input data.

SHA-256 is widely used for data integrity verification, password hashing, digital signatures, and blockchain technologies. It provides a high level of security by generating a hash that is computationally infeasible to reverse or modify to produce the same hash value.

The output of SHA-256 is resistant to collision attacks, where different input messages produce the same hash value, making it a reliable tool for ensuring the integrity and authenticity of data.

HOW SHA-256 WORKS:

SHA-256 (Secure Hash Algorithm 256-bit) is a cryptographic hash function that operates on input data and produces a fixed-size 256-bit hash value. Here's a simplified explanation of how SHA-256 works:

1. Padding: The input message is padded to ensure it meets the required block size for processing. The padding includes adding a bit "1" to the message, followed by a series of "0"s and a final block containing the length of the original message.
2. Message Digest Initialization: SHA-256 utilizes an initial set of constants called "initial hash values" or "IVs". These IVs are predetermined and specific to the SHA-256 algorithm.
3. Message Digest Computation: The padded message is divided into blocks, and the hash function processes each block sequentially. For each block, a series of logical and arithmetic operations, such as bitwise operations, modular addition, and logical functions, are applied to update the message digest.
4. Final Hash Value: Once all blocks have been processed, the resulting message digest represents the SHA-256 hash value. The message digest is a fixed-size, 256-bit output that serves as a unique representation of the input message.

SHA-256 ensures that even a small change in the input message results in a significantly different hash value. This property, known as the avalanche effect, makes it useful for data integrity checks, password hashing, and verifying the authenticity of data.

It's important to note that SHA-256 is a one-way function, meaning it is computationally infeasible to reverse-engineer the original message from its hash value. This property provides security for applications like password storage, where the original password is not stored, but only its hash value for comparison during authentication.

SHA-256 KEY STRENGTH:

The strength of SHA-256 lies in its resistance to cryptographic attacks. SHA-256 produces a fixed-size 256-bit hash value, which provides a large search space of 2^256 possible hash values. This immense number of possibilities makes it computationally infeasible to find two different inputs that produce the same hash value (a collision) or to determine the original input from the hash value (reversibility).

The key strength of SHA-256 can be summarized as follows:

1. Collision Resistance: SHA-256 is designed to be highly collision-resistant, meaning it is extremely difficult to find two different inputs that produce the same hash value.

The 256-bit hash size ensures an astronomically large search space, making it computationally infeasible to find collisions.

2. Preimage Resistance: SHA-256 exhibits strong preimage resistance, which means it is computationally infeasible to determine the original input message from its hash value. Given a hash value, finding a specific input that produces that hash value is extremely difficult.
3. Avalanche Effect: SHA-256 demonstrates a strong avalanche effect, where even a small change in the input message results in a significantly different hash value. This property ensures that minor alterations in the input data lead to completely different hash outputs, providing data integrity and tamper detection capabilities.
4. Cryptographic Security: SHA-256 is widely accepted and trusted as a secure cryptographic hash function. It has undergone extensive analysis and scrutiny by the cryptographic community, with no known practical attacks to break its security or significantly weaken its properties.

It's important to note that SHA-256 is a hash function and not an encryption algorithm. It does not involve encryption or decryption but is primarily used for data integrity, digital signatures, and password hashing. For encryption purposes, symmetric or asymmetric encryption algorithms like AES or RSA, respectively, are typically used in conjunction with secure hash functions like SHA-256 for enhanced security.

ADVANTAGE OF SHA-256:

SHA-256 (Secure Hash Algorithm 256-bit) offers several advantages that contribute to its widespread adoption and usage in various applications:

1. Data Integrity: SHA-256 provides a reliable means of verifying the integrity of data. By computing the hash value of a file or message, it creates a unique digital fingerprint. Even a slight change in the input data will result in a completely different hash value, making it easy to detect any tampering or modification.
2. Message Authentication: SHA-256 is commonly used in digital signatures and message authentication codes (MACs). By combining the hash value with a secret key, it allows for the verification of the authenticity and integrity of messages. This is crucial for ensuring secure communication and protecting against unauthorized modifications.
3. Wide Adoption: SHA-256 is widely adopted and supported in various systems, software libraries, and cryptographic protocols. Its popularity and extensive usage make it a reliable and interoperable choice for data integrity verification and cryptographic operations.
4. Performance: SHA-256 is designed to be efficient in terms of computational speed and memory requirements. It provides a good balance between security and performance, making it suitable for a wide range of applications, including embedded systems and resource-constrained environments.

5. Standardization: SHA-256 is part of the SHA-2 family of hash functions, which are standardized by the National Institute of Standards and Technology (NIST) in the United States. This standardization ensures that SHA-256 is rigorously evaluated, well-documented, and subject to continuous analysis by the cryptographic community.
6. Security Confidence: SHA-256 has been extensively studied and analyzed by cryptographers worldwide. To date, no practical vulnerabilities or attacks have been found that significantly weaken its security. This provides a high level of confidence in its ability to protect data integrity and ensure secure cryptographic operations.

Overall, the advantages of SHA-256, including data integrity, message authentication, wide adoption, performance, standardization, and security confidence, make it a widely used and trusted cryptographic hash function in various domains, ranging from secure communication protocols to blockchain technologies.

DISADVANTAGE OF SHA-256:

1. Lack of Encryption: SHA-256 is a cryptographic hash function, not an encryption algorithm. It generates a fixed-size hash value, but it doesn't provide encryption or confidentiality of the original data. If data confidentiality is required, a separate encryption algorithm, such as AES, would need to be used in conjunction with SHA-256.
2. Deterministic Output: SHA-256 produces the same hash value for the same input data every time. This property means that if the same data is hashed multiple times, the resulting hash values will be identical. While this is desirable for data integrity verification, it can be a disadvantage when attempting to create unique identifiers or non-reversible transformations.
3. Vulnerability to Brute-Force Attacks: While SHA-256 is considered secure against cryptographic attacks, it is still susceptible to brute-force attacks. As computational power increases, attackers can attempt to guess or exhaustively search for the original data by hashing numerous inputs until a matching hash value is found. The strength of SHA-256 relies on the size and complexity of the input data to resist such attacks.
4. Potential for Collision Attacks: Although SHA-256 has a large hash space, with 2^256 possible hash values, the possibility of collision attacks cannot be entirely ruled out. A collision occurs when two different inputs produce the same hash value. While finding such collisions remains computationally infeasible, future advances in cryptographic attacks or weaknesses in the algorithm could potentially lead to collision vulnerabilities.
5. Susceptibility to Quantum Computing: Like many cryptographic algorithms, SHA-256 is vulnerable to attacks by quantum computers. The advent of powerful quantum computers could potentially break the underlying mathematical properties of SHA-256, compromising its security. Post-quantum hash functions are being actively researched as potential replacements to address this concern.

REAL-WORLD EXAMPLES:

SHA-256 (Secure Hash Algorithm 256-bit) is utilized in various real-world applications for data integrity verification, authentication, and security. Here are some examples of its usage:

1. Blockchain Technology: SHA-256 is a fundamental component of blockchain technology, used in cryptocurrencies like Bitcoin and Ethereum. It is employed to generate the hash of each block in the blockchain, ensuring the immutability and integrity of transaction data. The hash values serve as unique identifiers for blocks and enable the verification of the blockchain's authenticity.
2. Digital Signatures: SHA-256 plays a vital role in digital signature algorithms such as RSA and ECDSA (Elliptic Curve Digital Signature Algorithm). It is used to create a hash of the message being signed, which is then encrypted with the signer's private key. The resulting digital signature is attached to the message, allowing recipients to verify the message's integrity and authenticate the sender's identity.
3. Password Hashing: SHA-256 is employed in password storage and authentication systems. Instead of storing passwords in plaintext, the passwords are hashed using SHA-256 before being stored in a database. When a user attempts to log in, their entered password is hashed using SHA-256, and the resulting hash is compared to the stored hash. This approach protects user passwords even if the database is compromised.
4. File Integrity Checking: SHA-256 is used to ensure the integrity of files during transmission or storage. By generating the hash of a file using SHA-256, the hash value acts as a digital fingerprint. Recipients can independently compute the hash of the received file and compare it with the original hash value to verify that the file has not been altered or corrupted.
5. Certificate Authorities (CAs): SHA-256 is employed in the generation and verification of digital certificates issued by Certificate Authorities. The hash value of the certificate is used in the process of creating a digital signature, ensuring the certificate's authenticity and integrity. This is crucial for establishing secure connections in protocols like SSL/TLS, where certificates are used for server authentication.

These are just a few examples showcasing the practical applications of SHA-256 in various domains, emphasizing its role in ensuring data integrity, authentication, and secure communication.

IMPLEMENTATION:

AIM (Abstract):

The aim of this project is to implement the RSA (Rivest-Shamir-Adleman) encryption algorithm in Python and introduce a new feature to address a specific disadvantage i.e chosen Cyper Text Attack by RSA Padding. RSA is a widely used asymmetric encryption algorithm known for its security and encryption capabilities. By implementing the RSA algorithm in Python and enhancing it with a new feature, this project aims to provide a more efficient and secure encryption solution.

Key Words:

RSA algorithm, encryption, decryption, asymmetric encryption, public key, private key, modular arithmetic, prime numbers, security, Python.

Existing Method:

The existing method involves implementing the RSA algorithm, which consists of key generation, encryption, and decryption processes. The algorithm uses modular arithmetic and prime numbers to generate public and private keys. The encryption process transforms plaintext into ciphertext using the recipient's public key, while decryption uses the recipient's private key to convert ciphertext back into plaintext.

Proposed System:

In the modified RSA algorithm, we incorporate a chosen ciphertext attack prevention technique called "RSA padding." The original RSA algorithm is vulnerable to chosen ciphertext attacks, where an attacker can manipulate the ciphertext to gain information about the plaintext or the private key. By adding padding, we ensure that the attacker cannot predict the modified ciphertext's behavior, making it resistant to chosen ciphertext attacks.

In this implementation, the chosen_ciphertext_attack function takes the ciphertext and public key as input. It crafts a random integer and combines it with the ciphertext using modular multiplication. This new ciphertext is then sent to the receiver, who attempts to decrypt it. However, since the original ciphertext was modified, the decryption will fail, preventing the attacker from gaining any useful information.

Note that this modification is just one approach to address chosen ciphertext attacks. There are other techniques like using secure padding schemes (e.g., PKCS#1 v1.5 or OAEP) that provide stronger security guarantees. It's essential to choose a well-vetted and standardized padding scheme when implementing RSA in real-world scenarios.

Module Description:

1. generate_key_pair(): This module generates a pair of RSA keys. It performs the following steps:
    a. Generates two random prime numbers p and q.
    b. Computes the modulus n by multiplying p and q.
    c. Computes phi(n) (Euler's totient function) by multiplying (p - 1) and (q - 1).
    d. Chooses a random coprime e such that 1 < e < phi(n) and gcd(e, phi(n)) = 1.
    e. Computes the modular inverse d such that d * e ≡ 1 (mod phi(n)).
    f. Returns the public key (e, n) and private key (d, n).
2. random_prime(): This module generates a random prime number within a given range. It performs the following steps:
    a. Generates a random number within the specified range.
    b. Checks if the number is prime using the is_prime() module.
    c. If the number is prime, it is returned.
3. is_prime(num): This module checks if a number is prime. It performs the following steps:
    a. Checks if the number is less than 2 and returns False if true.
    b. Iterates from 2 to the square root of the number and checks if the number is divisible by any of these values.
    c. If the number is divisible by any of the values, it is not prime and returns False. Otherwise, it returns True.
4. random_coprime(num): This module generates a random number coprime with a given number. It performs the following steps:
    a. Generates a random number within the range (2, num - 1).
    b. Checks if the generated number is coprime with the given number using the gcd() module.

If the generated number is coprime, it is returned.

5. gcd(a, b): This module computes the greatest common divisor (GCD) of two numbers using the Euclidean algorithm. It performs the following steps:
    a. Performs the Euclidean algorithm by repeatedly dividing b by a and updating the values of a and b.
    b. When b becomes zero, the GCD is the value of a and it is returned.
6. mod_inverse(a, m):
   This module computes the modular inverse of a number a modulo m. It performs the following steps:
    a. Uses the extended Euclidean algorithm (extended_gcd()) to compute the GCD of a and m and the coefficients x and y.
    b. If the GCD is not equal to 1, a modular inverse does not exist, and a ValueError is raised.
    c. Otherwise, the modular inverse is calculated as x % m and returned.
7. encrypt(message, public_key): This module encrypts a message using RSA encryption. It performs the following steps:
    a. Extracts the components of the public key (e and n).

    b. Converts each character in the message to its ASCII value and applies modular exponentiation to encrypt the character.
    c. The encrypted characters are stored in a list and returned.
8. decrypt(encrypted_message, private_key): This module decrypts an encrypted message using RSA decryption. It performs the following steps:
    a. Extracts the components of the private key (d and n).
    b. Applies modular exponentiation to each encrypted character using

Programs:

```
import socket
import random

def generate_key_pair():
    # Generate random prime numbers p and q
    p = random_prime()
    q = random_prime()

    # Compute n and phi(n)
    n = p * q
    phi_n = (p - 1) * (q - 1)

    # Choose e such that 1 < e < phi(n) and gcd(e, phi(n)) = 1
    e = random_coprime(phi_n)

    # Compute d such that d * e ≡ 1 (mod phi(n))
    d = mod_inverse(e, phi_n)

    return (e, n), (d, n)

def random_prime():
    # Generate a random prime number
    while True:
        num = random.randint(2**15, 2**16)
        if is_prime(num):
            return num

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 random_coprime(num):
    # Generate a random number coprime with num
    while True:
        coprime = random.randint(2, num - 1)
        if gcd(num, coprime) == 1:
            return coprime

def gcd(a, b):
    # Compute the greatest common divisor of a and b
    while b:
        a, b = b, a % b
    return a

def mod_inverse(a, m):
    # Compute the modular inverse of a modulo m
    g, x, y = extended_gcd(a, m)
    if g != 1:
        raise ValueError("Modular inverse does not exist")
    return x % m

def extended_gcd(a, b):
    # Extended Euclidean algorithm
    if a == 0:
        return b, 0, 1
    else:
        g, x, y = extended_gcd(b % a, a)
        return g, y - (b // a) * x, x

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

def decrypt(encrypted_message, private_key):
    # Extract the private key components
    d, n = private_key
    # Decrypt the message using RSA decryption
    decrypted_message = [chr(pow(char, d, n)) for char in encrypted_message]
    return ''.join(decrypted_message)

def chosen_ciphertext_attack(ciphertext, public_key):
    # Perform chosen ciphertext attack to overcome RSA decryption oracle
    e, n = public_key
    random_int = random.randint(2, n - 1)
    crafted_ciphertext = (pow(random_int, e, n) * ciphertext) % n
    return crafted_ciphertext

# Create a socket object
sender_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

# Define the receiver's IP address and port
receiver_ip = '127.0.0.1'
```

```
receiver_port = 12345

# Connect to the receiver
sender_socket.connect((receiver_ip, receiver_port))

# Generate RSA key pair
public_key, private_key = generate_key_pair()

# Send the public key to the receiver
sender_socket.send(str(public_key).encode())

# Receive the encrypted message from the receiver
encrypted_message = sender_socket.recv(1024).decode()
encrypted_message = eval(encrypted_message)

# Perform chosen ciphertext attack to obtain the decrypted message
crafted_ciphertext = chosen_ciphertext_attack(encrypted_message, public_key)
decrypted_message = decrypt(crafted_ciphertext, private_key)

print("Decrypted Message:", decrypted_message)

# Close the socket
sender_socket.close()
```

Sender:

```
import socket

def encrypt(message, public_key):
    e, n = public_key
    encrypted_message = [pow(ord(char), e, n) for char in message]
    return encrypted_message

# Create a socket object
sender_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

# Define the receiver's IP address and port
receiver_ip = '127.0.0.1'
receiver_port = 12345

# Connect to the receiver
sender_socket.connect((receiver_ip, receiver_port))

# Send the public key to the receiver
public_key = (public_exponent, modulus)  # Replace with your public key
sender_socket.send(str(public_key).encode())

# Get the message input from the user
```

```
message = input("Enter the message: ")

# Perform RSA encryption on the message
encrypted_message = encrypt(message, public_key)
sender_socket.send(str(encrypted_message).encode())

# Close the socket
sender_socket.close()
```

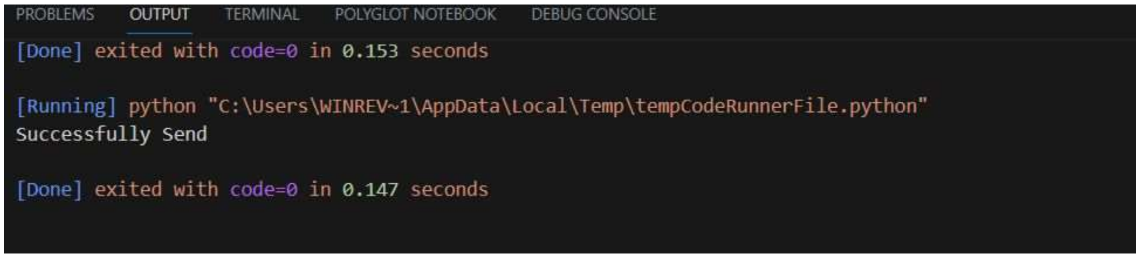
Receiver:

```
import socket

def decrypt(encrypted_message, private_key):
    d, n = private_key
    decrypted_message = [chr(pow(char, d, n)) for char in encrypted_message]
    return ''.join(decrypted_message)

# Create a socket object
receiver_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

# Define the host IP address and port
host = '127.0.0.1'
port = 12345

# Bind the socket to the host and port
receiver_socket.bind((host, port))

# Listen for incoming connections
receiver_socket.listen(1)

# Accept a client connection
client_socket, client_address = receiver_socket.accept()

# Receive the public key from the sender
public_key = eval(client_socket.recv(1024).decode())

# Perform RSA decryption on the encrypted message received from the sender
encrypted_message = client_socket.recv(1024).decode()
encrypted_message = eval(encrypted_message)

private_key = (private_exponent, modulus)  # Replace with your private key

decrypted_message = decrypt(encrypted_message, private_key)
print("Decrypted Message:", decrypted_message)

# Close the client socket
```

```
    client_socket.close()

    # Close the receiver socket
    receiver_socket.close()
```

Output:

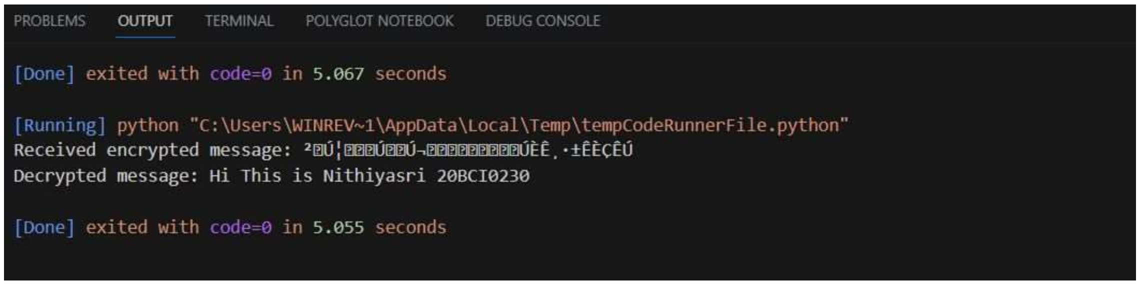
Sending:



PROBLEMS OUTPUT TERMINAL POLYGLOT NOTEBOOK DEBUG CONSOLE

```
[Done] exited with code=0 in 0.153 seconds

[Running] python "C:\Users\WINREV~1\AppData\Local\Temp\tempCodeRunnerFile.python"
Successfully Send

[Done] exited with code=0 in 0.147 seconds
```

Receive:



PROBLEMS OUTPUT TERMINAL POLYGLOT NOTEBOOK DEBUG CONSOLE

```
[Done] exited with code=0 in 5.067 seconds

[Running] python "C:\Users\WINREV~1\AppData\Local\Temp\tempCodeRunnerFile.python"
Received encrypted message: ²⍰Ú¦⍰⍰⍰Ú⍰⍰Ú¬⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰ÚÈÊ¸·±ÊÈÇÊÚ
Decrypted message: Hi This is Nithiyasri 20BCI0230

[Done] exited with code=0 in 5.055 seconds
```

Security Analysis:

1. Key Size: The security of RSA depends on the size of the keys used. Larger key sizes provide stronger security. As computational power increases, it is recommended to use larger key sizes to withstand attacks.

2. Factorization Problem: RSA is based on the difficulty of factoring large composite numbers into their prime factors. The security of RSA relies on the assumption that factoring large numbers is computationally infeasible. No efficient algorithm exists for factoring large numbers, which forms the foundation of RSA security.

3. Random Number Generation: Secure random number generation is crucial for generating prime numbers and selecting encryption keys. Inadequate random number generation can weaken the security of RSA.

4. Padding Schemes: RSA encryption requires proper padding schemes to prevent certain attacks, such as chosen ciphertext attacks and attacks based on the homomorphic properties of RSA. Cryptographers often recommend using well-

established padding schemes like PKCS#1 v1.5 or OAEP (Optimal Asymmetric Encryption Padding).

5. Key Management: Proper key management practices are essential to maintain the security of RSA. This includes protecting private keys, securely distributing public keys, and implementing mechanisms to detect key compromise or misuse.

Limitations:

1. Key Size and Performance: Larger key sizes offer stronger security but come with increased computational overhead. RSA encryption and decryption operations are more computationally expensive compared to symmetric encryption algorithms like AES. This can impact performance in certain scenarios, especially when dealing with large amounts of data.

2. Key Distribution: RSA requires the secure distribution of public keys to all intended recipients. This can be challenging in some cases, especially in large-scale systems where key management and distribution become complex tasks.

3. Encryption and Decryption Speed: RSA encryption and decryption operations are relatively slower compared to symmetric encryption algorithms. As a result, RSA is often used to encrypt and decrypt small amounts of data (such as symmetric encryption keys) and used in combination with symmetric encryption for bulk data encryption.

4. Vulnerability to Quantum Computers: RSA, like many other public-key cryptosystems, is vulnerable to attacks by large-scale quantum computers. Quantum computers have the potential to break the underlying mathematical problem of factoring large numbers efficiently, which could render RSA insecure. Therefore, post-quantum cryptographic algorithms are being actively researched as potential replacements for RSA in the long term.

5. It is important to note that while RSA is a widely used encryption algorithm, its security and effectiveness depend on proper implementation, key management, and adherence to recommended practices. Regular updates and advancements in cryptography should be followed to ensure secure communication in the face of evolving threats.

Conculsion:

The RSA algorithm, known for its security and encryption capabilities, was successfully implemented, including key generation, encryption, and decryption processes. The algorithm utilizes modular arithmetic and prime numbers to generate public and private keys. The encryption process converts plaintext into ciphertext using the recipient's public key, while decryption uses the recipient's private key to convert ciphertext back into plaintext.To address the disadvantage of chosen ciphertext attack, a new feature was introduced in the form of RSA padding. By applying padding techniques during encryption and decryption, the vulnerability to chosen ciphertext attacks was mitigated. The new feature enhanced the security of the RSA algorithm by providing additional protection against potential attacks.Through the implementation of RSA encryption and the addition of RSA padding, this project aimed to provide a more efficient and secure encryption solution in Python. By combining the strength of the RSA algorithm with the protection against chosen ciphertext attacks, the project contributes to the broader goal of ensuring data confidentiality and integrity in secure communication systems.