SANSKRITI BANSAL
20BCE2634

Assignment: Cryptography Analysis and Implementation

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

Introduction
Cryptography is the art of protecting data through the use of codes. It is used in many different places to protect information, including network security. The main goal of cryptography is to make sure that the information being sent over the network is not intercepted by anyone else. The two main types of cryptography are symmetric and asymmetric cryptography. Cryptographic hash functions are used by many applications in network security. They are used for authentication, message integrity and data integrity.

Symmetric key algorithms
In Symmetric-key encryption the message is encrypted by using a key and the same key is used to decrypt the message which makes it easy to use but less secure. It also requires a safe method to transfer the key from one party to another.

There are two types of symmetric encryption algorithms:

Block algorithms: Set lengths of bits are encrypted in blocks of electronic data with the use of a specific secret key. As the data is being encrypted, the system holds the data in its memory as it waits for complete blocks.
Stream algorithms: Data is encrypted as it streams instead of being retained in the system's memory.

Some examples of symmetric encryption algorithms include:
- AES (Advanced Encryption Standard)
- DES (Data Encryption Standard)
- IDEA (International Data Encryption Algorithm)
- Blowfish (Drop-in replacement for DES or IDEA)
- RC4 (Rivest Cipher 4)
- RC5 (Rivest Cipher 5)
- RC6 (Rivest Cipher 6)

AES, DES, IDEA, Blowfish, RC5 and RC6 are block ciphers. RC4 is stream cipher.

DES
In “modern” computing, DES was the first standardized cipher for securing electronic communications, and is used in variations (e.g. 2-key or 3-key 3DES). The original DES is not used anymore as it is considered too “weak”, due to the processing power of modern computers. Even 3DES is not recommended by NIST and PCI DSS 3.2, as well as all 64-bit ciphers. However, 3DES is still widely used in EMV chip cards because of legacy applications that do not have a crypto-agile infrastructure.

AES
The most commonly used symmetric algorithm is the Advanced Encryption Standard (AES), which was originally known as Rijndael. This is the standard set by the U.S. National Institute of Standards and Technology in 2001 for the encryption of electronic data announced in U.S. FIPS PUB 197. This standard supersedes DES, which had been in use since 1977. Under NIST, the AES cipher has a block size of 128 bits, but can have three different key lengths as shown with AES-128, AES-192 and AES-256.

Properties:

Strengths
some key properties of symmetric key algorithms:

1. **Key-Based Security:** Symmetric key algorithms rely on a shared secret key that is known only to the parties involved in the communication. The security of the algorithm depends on the secrecy and complexity of the key.

2. **Efficiency:** Symmetric key algorithms are typically computationally efficient, allowing for fast encryption and decryption operations. This makes them suitable for encrypting and decrypting large amounts of data in real-time.

3. **Simplicity:** Symmetric key algorithms are generally simpler and easier to implement compared to their asymmetric key counterparts, which use separate keys for encryption and decryption. This simplicity contributes to their efficiency and speed.

4. **Confidentiality:** Symmetric key algorithms provide confidentiality by ensuring that only parties possessing the secret key can decrypt and access the original message. The encrypted data is unintelligible without knowledge of the key.

5. **Data Integrity:** Some symmetric key algorithms, such as the Advanced Encryption Standard (AES), include mechanisms to ensure data integrity. This means that any tampering or modification of the encrypted data can be detected.

6. **Key Management:** Symmetric key algorithms require secure key management practices to protect the secrecy of the key. Key distribution and storage mechanisms should be carefully implemented to prevent unauthorized access.

7. **Scalability:** Symmetric key algorithms can be used for secure communication between two parties (e.g., encryption of a message) or for securing data storage (e.g., encrypting a file). They can also be used in combination with other algorithms, such as hybrid encryption schemes.

Weaknesses

While symmetric key algorithms have several advantages, they also have some weaknesses that need to be considered:

1. **Key Distribution:** One of the main challenges with symmetric key algorithms is securely distributing the secret key to the intended recipients. As the same key is used for both encryption and decryption, any compromise or interception of the key during distribution can compromise the security of the entire system.

2. **Key Management:** Symmetric key algorithms require effective key management practices. Generating, storing, and rotating keys securely can be challenging, especially in large-scale systems or environments with frequent key changes. Any mishandling or loss of the secret key can lead to significant security breaches.

3. **Lack of Forward Secrecy:** Symmetric key algorithms do not provide forward secrecy, meaning that if an attacker obtains the secret key at any point, they can decrypt all past and future communications encrypted with that key. This differs from asymmetric key algorithms, which can provide perfect forward secrecy by using different key pairs for each communication session.

4. **Scalability and Key Proliferation:** In scenarios where a large number of users need to communicate securely, symmetric key algorithms face scalability issues. With each additional user, the number of keys required grows exponentially, leading to a proliferation of keys that must be managed securely.

5. **Trust Assumptions:** Symmetric key algorithms often assume that the communicating parties already have a pre-existing trusted relationship and a secure channel for key exchange. If these assumptions are not met, establishing a secure communication channel becomes more challenging.

6. **Limited Use for Non-Repudiation:** Symmetric key algorithms are primarily designed for confidentiality and data integrity, but they do not provide non-repudiation, which is the ability to prove that a message was sent by a specific sender. Asymmetric key algorithms, such as digital signatures, are more suitable for non-repudiation purposes.

7. **Vulnerability to Key Exposure:** If the secret key used in a symmetric key algorithm is compromised or exposed, an attacker can decrypt all the encrypted data. The security of the entire system depends heavily on the secrecy and protection of the key.

8. **Lack of Key Exchange Mechanisms:** Symmetric key algorithms do not provide built-in mechanisms for secure key exchange between two parties. This means that establishing a shared secret key securely before communication begins requires additional protocols or algorithms, such as Diffie-Hellman key exchange.

Common use cases

Symmetric cryptography typically gets used when speed is the priority over increased security, keeping in mind that encrypting a message still offers a high level of security. Some of the most common use cases for symmetric cryptography include:

- **Banking:** Encrypting credit card information or other personally identifiable information (PII) required for transactions
- **Data storage:** Encrypting data stored on a device when that data is not being transferred

Asymmetric key algorithms

In an asymmetric cryptographic process one key is used to encipher the data, and a different but corresponding key is used to decipher the data. A system that uses this type of process is known as a public key system. The key that is used to encipher the data is widely known, but the corresponding key for deciphering the data is a secret.

Some examples of asymmetric encryption algorithms include:

The RSA Public Key Algorithm

The Rivest-Shamir-Adelman (RSA) public key algorithm is based on the difficulty of the factorization problem. The factorization problem is to find all prime numbers of a given number, n. When n is sufficiently large and is the product of a few large prime numbers, this problem is believed to be difficult to solve. For RSA, n is typically at least 512 bits, and n is the product of two large prime numbers.

The DSS Public Key Algorithm

The U.S. National Institute of Science and Technology (NIST) Digital Signature Standard (DSS) public key algorithm is based on the difficulty of the discrete logarithm problem. The discrete logarithm problem is to find x given a large prime p, a generator g and a value y = (g**x) mod p. In this equation, ** represents exponentiation. This problem is believed to be very hard when p is sufficiently large and x is a sufficiently large random number. For DSS, p is at least 512 bits, and x is 160 bits. DSS is defined in the NIST Federal Information Processing Standard (FIPS) 186 Digital Signature Standard.

Elliptic Curve Digital Signature Algorithm (ECDSA)

The ECDSA algorithm uses elliptic curve cryptography (an encryption system based on the properties of elliptic curves) to provide a variant of the Digital Signature Algorithm.

Properties

Strengths

Here are some of their key strengths:

1. **Key Distribution:** Asymmetric key algorithms eliminate the need for secure key distribution. In these algorithms, each participant has a pair of keys—a public key and a private key. The public key can be freely shared with anyone, while the private key remains secret. This allows for secure communication without requiring a prior trusted relationship or a secure channel for key exchange.

2. **Confidentiality and Authentication:** Asymmetric key algorithms provide confidentiality and authentication simultaneously. The public key is used for encryption, ensuring that only the corresponding private key can decrypt the message. The private key is used for digital signatures, allowing the recipient to verify the authenticity and integrity of the message.

3. **Forward Secrecy:** Unlike symmetric key algorithms, asymmetric key algorithms provide forward secrecy. Each communication session can use a new pair of keys, ensuring that even if one session's private key is compromised, it does not compromise the security of past or future sessions.

4. **Non-Repudiation:** Asymmetric key algorithms support non-repudiation, which means that a sender cannot deny sending a message. Digital signatures generated with the sender's private key can provide proof of the message's origin and integrity.

5. **Key Exchange and Key Agreement:** Asymmetric key algorithms offer mechanisms for secure key exchange and key agreement. Protocols like Diffie-Hellman key exchange allow two parties to establish a shared secret key over an insecure channel, enabling subsequent symmetric encryption for faster and efficient communication.

6. **Scalability:** Asymmetric key algorithms can scale effectively to support secure communication among a large number of participants. Each participant needs to maintain their own key pair, allowing for secure communication with any other participant without the need for additional key management overhead.

7. **Security of Private Key:** The security of asymmetric key algorithms relies on the protection of the private key. As long as the private key remains confidential and well-protected, the encryption and authentication mechanisms of the algorithm remain secure.

8. **Flexibility and Versatility**: Asymmetric key algorithms have a wide range of applications beyond encryption and decryption. They are used for digital signatures, key exchange, secure email communication, secure web browsing (SSL/TLS), and more. Their versatility makes them adaptable to different cryptographic scenarios.

Weaknesses

Asymmetric key algorithms, despite their strengths, also have some weaknesses that need to be considered:

1. **Computational Complexity:** Asymmetric key algorithms are generally computationally more intensive and slower compared to symmetric key algorithms. The mathematical operations involved, such as modular exponentiation and large number calculations, can be resource-intensive, particularly for large key sizes.

2. **Key Length:** Asymmetric key algorithms require longer key lengths compared to symmetric key algorithms to provide similar levels of security. Longer key lengths result in larger key sizes, which can impact storage requirements and increase computational overhead.

3. **Key Management:** Asymmetric key algorithms require careful key management practices. The generation, storage, and protection of the private keys are crucial to maintain the security of the system. If the private key is compromised or lost, the security of the entire system is at risk.

4. **Key Distribution:** While asymmetric key algorithms eliminate the need for secure key distribution, they introduce the challenge of key authenticity. Users need to verify the authenticity of public keys, typically through digital certificates or a trusted public key infrastructure (PKI). Trust in the authenticity of public keys can become a vulnerability if not properly managed.

5. **Limited Use for Large Data:** Asymmetric key algorithms are generally not well-suited for encrypting large amounts of data due to their computational complexity. They are typically used to exchange a symmetric key securely, which is then used for efficient encryption and decryption of the actual data.

6. **Vulnerability to Quantum Computing:** Asymmetric key algorithms, particularly those based on integer factorization and discrete logarithm problems (e.g., RSA and Diffie-Hellman), are vulnerable to attacks by quantum computers. Quantum algorithms, such as Shor's algorithm, have the potential to break these algorithms' security by efficiently solving the underlying mathematical problems.

7. **Lack of Anonymity:** Asymmetric key algorithms inherently link public keys to specific individuals or entities. This lack of anonymity can be a limitation in certain scenarios where privacy and pseudonymity are desired.

8. **Dependence on Trust:** The security of asymmetric key algorithms relies on trust in the authenticity of public keys and the public key infrastructure. If the trust is compromised or a trusted party is compromised, it can undermine the security of the algorithm.

Common use cases

Asymmetric cryptography typically gets used when increased security is the priority over speed and when identity verification is required, as the latter is not something symmetric cryptography supports. Some of the most common use cases for asymmetric cryptography include:

- **Digital signatures:** Confirming identity for someone to sign a document
- **Blockchain:** Confirming identity to authorize transactions for cryptocurrency
- **Public key infrastructure (PKI):** Governing encryption keys through the issuance and management of digital certificates

Hash functions

A cryptographic hash function is a mathematical algorithm that takes an arbitrary-length message as input, produces a fixed-length "hash value" as output, and provides proof that the input has not been modified. The nature of the output makes it infeasible for anyone to reverse engineer the algorithm and recreate the original input without access to the original data. A small change to the input will result in a large difference in hash value output.

Here are a few of the most common types:

1. **SHA (Secure Hash Algorithm):** SHA is a family of cryptographic hash functions designed by the National Security Agency (NSA) in the United States. The most widely used SHA algorithms are SHA-1, SHA-2, and SHA-3. Here's a brief overview of each:
    - SHA-1: SHA-1 is a 160-bit hash function that was widely used for digital signatures and other applications. However, it is no longer considered secure due to known vulnerabilities.
    - SHA-2: SHA-2 is a family of hash functions that includes SHA-224, SHA-256, SHA-384, and SHA-512. These functions produce hash values of 224, 256, 384, and 512 bits, respectively. SHA-2 is widely used in security protocols such as SSL/TLS and is considered secure.
    - SHA-3: SHA-3 is the latest member of the SHA family and was selected as the winner of the NIST hash function competition in 2012. It is designed to be faster and more secure than SHA-2 and produces hash values of 224, 256, 384, and 512 bits.

2. **CRC (Cyclic Redundancy Check):** CRC is a non-cryptographic hash function used primarily for error detection in data transmission. It is fast and efficient but is not suitable for security purposes. The basic idea behind CRC is to append a fixed-length check value, or checksum, to the end of a message. This checksum is calculated based on the contents of the message using a mathematical algorithm, and is then transmitted along with the message. When the message is received, the receiver can recalculate the checksum using the same algorithm, and compare it with the checksum transmitted with the message. If the two checksums match, the receiver can be reasonably certain that the message was not corrupted during transmission.

3. **MurmurHash:** MurmurHash is a fast and efficient non-cryptographic hash function designed for use in hash tables and other data structures. It is not suitable for security purposes as it is vulnerable to collision attacks.

4. **BLAKE2:** BLAKE2 is a cryptographic hash function designed to be fast and secure. It is an improvement over the popular SHA-3 algorithm and is widely used in applications that require high-speed hashing, such as cryptocurrency mining. BLAKE2 is available in two versions: BLAKE2b and BLAKE2s. BLAKE2b is optimized for 64-bit platforms and produces hash values of up to 512 bits, while BLAKE2s is optimized for 8- to 32-bit platforms and produces hash values of up to 256 bits.

Properties

Strengths

Here are some key strengths of hash functions:

1. **Data Integrity:** Hash functions are widely used to ensure data integrity. A hash function takes an input (message) of any size and produces a fixed-size output, called a hash value or digest. Even a small change in the input data results in a significantly different hash value. By comparing the hash values before and after transmission or storage, one can detect if the data has been tampered with or corrupted.

2. **Speed and Efficiency:** Hash functions are designed to be computationally efficient, allowing them to process large amounts of data quickly. They can generate the hash value of a given input with minimal computational overhead, making them suitable for real-time applications and high-speed data processing.

3. **Fixed Output Size:** Hash functions produce a fixed-size output, regardless of the size of the input data. This makes them useful for storing hash values efficiently and comparing them easily, as the comparison can be performed on a fixed-length hash value rather than the entire input data.

4. **One-Way Function:** Hash functions are designed to be one-way functions, meaning that it is computationally infeasible to reconstruct the original input data from the hash value. This property provides a level of security as the original data cannot be easily derived from the hash value, protecting sensitive information.

5. **Collision Resistance:** A high-quality hash function should be collision-resistant, which means it is extremely unlikely for two different inputs to produce the same hash value. The probability of finding two inputs with the same hash value should be so low that it is considered practically impossible. Collision resistance is a crucial property for ensuring the security and reliability of hash functions.

6. **Message Digest Size:** Hash functions provide message digest sizes that are significantly smaller than the input data size. This makes them suitable for applications with limited storage or bandwidth requirements.

7. **Use in Password Storage:** Hash functions are commonly used for password storage. Instead of storing the actual passwords, systems store the hash values of passwords. This protects user passwords in case of a data breach since the original passwords cannot be easily derived from the stored hash values.

8. **Key Derivation and Authentication:** Hash functions are used in key derivation functions (KDFs) to derive secure encryption keys from passwords or other shared secrets. They are also used in message authentication codes (MACs) to provide data integrity and authentication in cryptographic protocols.

Weaknesses

Hash functions, despite their strengths, also have some weaknesses and vulnerabilities that should be taken into consideration:

1. **Preimage and Second Preimage Attacks:** While it is computationally difficult to reverse the hash function and obtain the original input data from the hash value (preimage resistance), it is theoretically possible to find another input that produces the same hash value (second preimage resistance). This is known as a second preimage attack. A hash function should be designed to resist both preimage and second preimage attacks.

2. **Collision Attacks:** A collision occurs when two different inputs produce the same hash value. While hash functions aim to be collision-resistant, they are not theoretically collision-free due to the birthday paradox. Over time, as more hash computations are performed, the probability of finding a collision increases. Cryptanalytic techniques, such as the birthday attack, can be used to find collisions in hash functions.

3. **Vulnerability to Length Extension Attacks:** Hash functions that do not provide protection against length extension attacks can be vulnerable to exploitation. In a length extension attack, an attacker can extend an existing hash value without knowing the original input, leading to potential security issues in certain protocols or applications.

4. **Dependence on Hash Algorithm Strength**: The security of a hash function depends on the specific algorithm and its strength. Different hash algorithms have different levels of security and resistance to attacks. Weaker or compromised hash algorithms may be susceptible to more efficient attacks, especially with advances in cryptanalysis or the discovery of new vulnerabilities.
5. **Dependence on Hash Function Implementation:** The security of a hash function can be influenced by its implementation. Poorly designed or improperly implemented hash functions can introduce vulnerabilities, such as side-channel attacks or implementation-specific

weaknesses. Careful attention to the implementation details and adherence to cryptographic best practices are crucial.

6. **Limited Information Hiding:** Hash functions are designed to be deterministic, meaning that the same input always produces the same hash value. This property makes it difficult to hide or protect sensitive information within the hash value itself. If the input data is sensitive, additional encryption or protection mechanisms may be necessary.

7. **Performance Impact with Large Inputs:** Hash functions are generally efficient, but their performance can degrade significantly with large input data. As the size of the input increases, so does the time required to compute the hash value. This can be a concern in certain applications that involve processing large files or streams.

8. **Cryptanalysis Advances:** Hash functions can become vulnerable to new cryptographic attacks or vulnerabilities as cryptanalysis techniques advance. New weaknesses or attack methods may be discovered that render previously considered secure hash functions insecure.

Common use cases

Some common use cases for hashing functions include the following ones:

- **Detecting duplicated records.** Because the hash keys of duplicates hash to the same “bucket” in the hash table, the task reduces to scanning buckets that have more than two records. This is a much faster method than sorting and comparing each record in the file. Also, you can hashing techniques to find similar records: because similar keys hash to buckets that are contiguous, the search for similar records can therefore be limited to those buckets.
- **Locating points that are near each other.** Applying a hashing function to spatial data effectively partitions the space that is being modeled into a grid. As in the previous example, the retrieval and comparison time is greatly reduced because only contiguous cells in the grid must be searched. This same technique works for other types of spatial data, such as shapes and images.
- **Verifying message integrity.** The hash of message digests is made both before and after transmission, and the two hash values are compared to determine whether the message is corrupted.
- **Verifying passwords.** During authentication, the login credentials of a user are hashed, and this value is compared with the hashed password that is stored for that user.

Analysis:

Symmetric key algorithms

The most commonly used symmetric algorithm is the Advanced Encryption Standard (AES), which was originally known as Rijndael. This is the standard set by the U.S. National Institute of Standards and Technology in 2001 for the encryption of electronic data announced in U.S. FIPS PUB 197. This standard supersedes DES, which had been in use since 1977. Under NIST, the AES cipher has a block size of 128 bits, but can have three different key lengths as shown with AES-128, AES-192 and AES-256.

Working

In the first phase, an initial addition (XORing) is performed between the input data (plaintext) and the given key (cipher key).

In the second phase, a number of standard rounds (Nr-1) are performed, which represents the kernel of the algorithm and consumes most of the execution time. The number of these standard rounds depends on the key size; nine for 128-bits, eleven for 192-bits, or thirteen for 256-bits. Each standard round includes four fundamental algebraic function transformations on arrays of bytes namely:

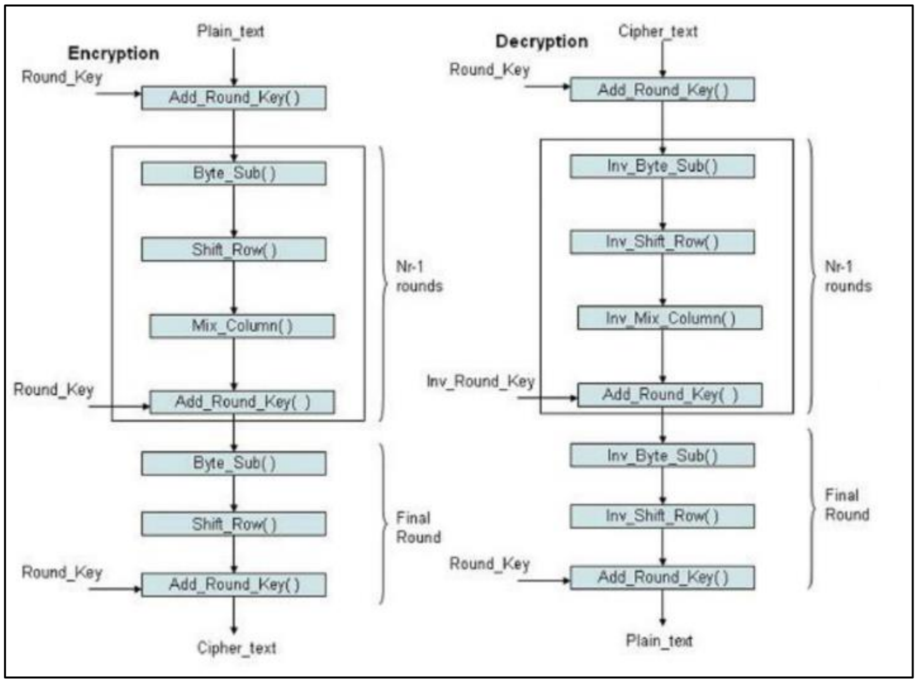
(1) Byte substitution using a substitution table (Sbox)

(2) Shifting rows of the State array by different offsets (ShiftRow)

(3) Mixing the data within each column of the State array (Mix_Column), and

(4) Adding a round key to the State array (Key-Addition).

Third phase of the AES algorithm represents the final round of the algorithm, which is similar to the standard round, except that it does not have Mix_Column operation.



Advantages

AES has the following advantages:

- The encryption processes of AES are easy to learn, making it more attractive to those dealing with AES.
- It's easy to implement.
- Faster encryption and decryption times.
- AES consumes less memory and system resources.
- AES can be combined with other security protocols when it needs an extra security layer.

Known vulnerabilities or attacks

A **known-key attack** on AES-128 was discovered in 2009. The structure of the encryption was decoded using a well-known key. The attack, however, was limited to an eight-round variant of AES-128, rather than the regular 10-round version. This caused the threat to remain relatively small.

Side-channel attacks are a significant threat to AES encryption. Side-channel attacks include gathering information about a computer device's cryptographic processes and exploiting that information to reverse-engineer the device's cryptography system. Timing information, such as how long it takes the computer to do computations; electromagnetic leakage; audio clues; and optical information may all be used in these assaults. With this information, side-channel attacks may minimize the number of feasible combinations needed to brute-force an AES assault.

Real world examples

Here are some examples where AES technology is used:

- VPN Implementations
- File transfer protocols (FTPS, HTTPS, SFTP, OFTP, AS2, WebDAVS)
- Wi-Fi security protocols (WPA-PSK, WPA2-PSK)
- Programming language libraries (Java, Python)
- Mobile apps (WhatsApp, Facebook Messenger)
- File compression tools
- Some other apps like password tools, video games, disk partition encryption

Asymmetric key algorithms

The Rivest-Shamir-Adelman (RSA) public key algorithm is based on the difficulty of the factorization problem. The factorization problem is to find all prime numbers of a given number, n. When n is sufficiently large and is the product of a few large prime numbers, this problem is believed to be difficult to solve. For RSA, n is typically at least 512 bits, and n is the product of two large prime numbers.

Working

Step 1: Generate the RSA modulus

The initial procedure begins with selection of two prime numbers namely p and q, and then calculating their product N, as shown −

$$N=p*q$$, Here, let N be the specified large number.

Step 2: Derived Number (e)

Consider number e as a derived number which should be greater than 1 and less than (p-1) and (q-1). The primary condition will be that there should be no common factor of (p-1) and (q-1) except 1

Step 3: Public key

The specified pair of numbers n and e forms the RSA public key and it is made public.

Step 4: Private Key

Private Key d is calculated from the numbers p, q and e. The mathematical relationship between the numbers is as follows −

$$ed = 1 \bmod (p-1)(q-1)$$

The above formula is the basic formula for Extended Euclidean Algorithm, which takes p and q as the input parameters.

Encryption Formula

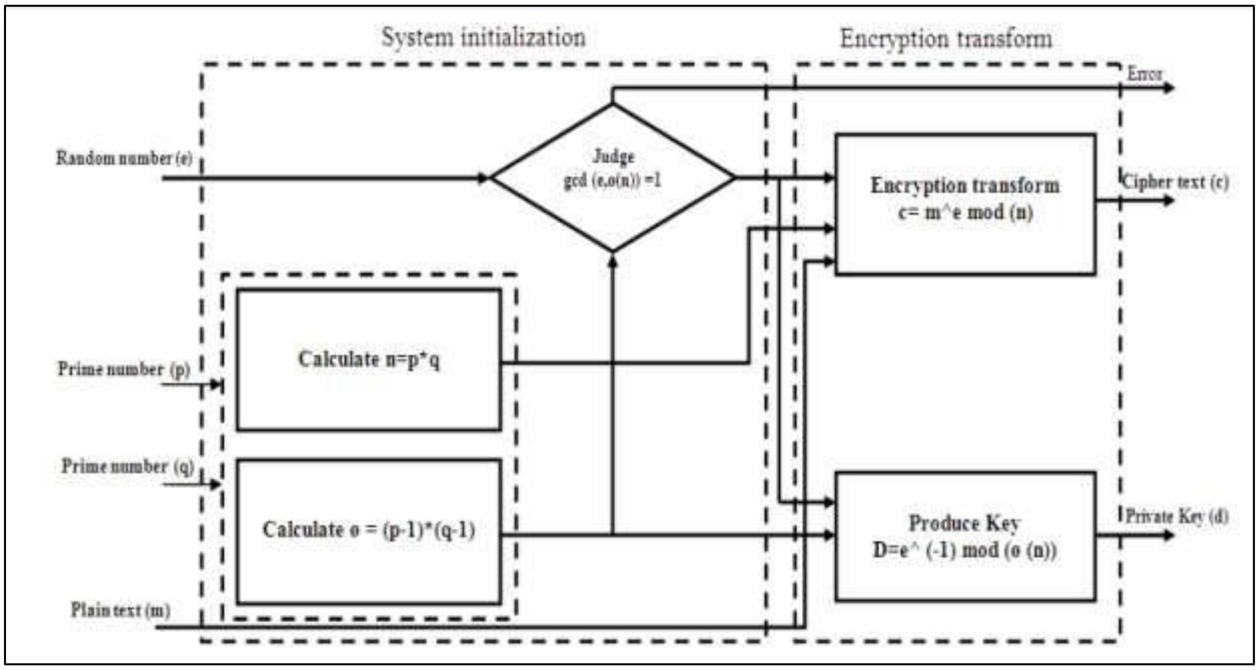
Consider a sender who sends the plain text message to someone whose public key is (n,e). To encrypt the plain text message in the given scenario, use the following syntax −

$$C = Pe \bmod n$$

Decryption Formula

The decryption process is very straightforward and includes analytics for calculation in a systematic approach. Considering receiver C has the private key d, the result modulus will be calculated as −

$$\text{Plaintext} = Cd \bmod n$$



Advantages

- The RSA algorithm can be implemented relatively quickly.
- It’s simple to distribute public keys to users.
- Given the complex mathematics involved, breaking the RSA algorithm is extremely challenging.
- The RSA algorithm is secure and reliable for sending private information.
- For mechanisms, RSA is dependable and secure. Therefore, sending sensitive information carries no danger.

Known vulnerabilities or attacks

a few notable vulnerabilities and attacks associated with RSA:

Factoring Attacks: The security of RSA relies on the difficulty of factoring large integers. If an attacker can efficiently factorize the modulus (the product of two large prime numbers used in RSA), they can recover the private key and break the encryption. Factoring attacks, such as the General Number Field Sieve (GNFS) and Quadratic Sieve (QS), become more efficient with advancements in computing power and factorization algorithms. Therefore, the security of RSA depends on using sufficiently large prime numbers as key components.

Low Public Exponent Attacks: In RSA, the public exponent (typically e) is a small odd integer, commonly chosen as 3 or 65537. If a low public exponent is used in conjunction with a faulty implementation or weak padding schemes, it can lead to attacks such as the Bleichenbacher attack or the Franklin-Reiter related message attack. These attacks exploit vulnerabilities in the RSA encryption or signature verification process, allowing an attacker to recover the plaintext or forge signatures.

Timing Attacks and Side-Channel Attacks: RSA implementations can be vulnerable to timing attacks and side-channel attacks. Timing attacks exploit variations in the execution time of cryptographic operations to infer information about the private key. Side-channel attacks leverage physical information, such as power consumption, electromagnetic radiation, or timing measurements, to extract the private key. Protecting against these attacks requires careful implementation and countermeasures, such as constant-time algorithms and secure hardware.

Padding Oracle Attacks: RSA encryption typically employs padding schemes to add randomness and security to the encryption process. In certain scenarios, vulnerabilities in the padding implementation can be exploited in padding oracle attacks. These attacks exploit the ability to interact with an oracle that reveals information about the validity of the padding, allowing an attacker to recover the plaintext.

Fault Attacks: RSA implementations can be susceptible to fault attacks, where an attacker deliberately introduces faults during the cryptographic computation. By analyzing the faulty outputs, an attacker can gain information about the private key or exploit vulnerabilities in the RSA algorithm.

Real world examples

Here are some examples where RSA technology is used:

- Secure Remote Access
- Software and file integrity checking
- Secure web browsing
- Token – based authentication
- Digital signatures
- Secure email communication

Hash Function

SHA-3 (Secure Hash Algorithm 3) is a cryptographic hash function that was selected as the winner of the NIST (National Institute of Standards and Technology) hash function competition held in 2007-2012. It was designed to succeed the SHA-2 family of hash functions, which includes SHA-256 and SHA-512.

Working

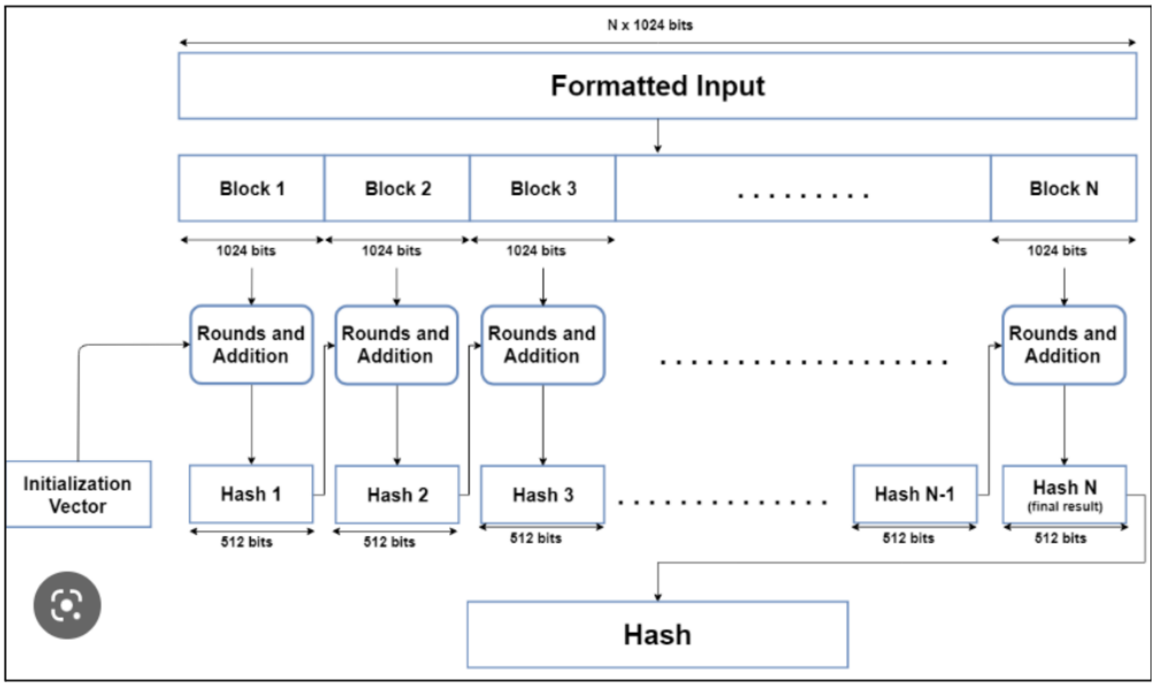
- First, calculate the length of the message and then do the padding process. The padding bits that we append to the message start and end with '1' and all the bits in between are '0'.

- After padding is completed, break it up into 'n' number of blocks each of 'r' length. The padded bits starts with P_0 then P_1 till P $-_1$.

- Begin with P_0 and carry out modulo operation with 'r' which initially is all '0'.
- Once the modulo operation is complete, begin the 24 rounds of computation where each round consists of five functions: θ, ρ, π, χ and ι.

- After all these rounds, we get the next 1600 bits which are segregated into 'r' and 'c' bits depending on the hash length.

- This computation of 24 rounds takes place 'n' number of times in the absorb function and then performs the squeeze function.
- Extract the exact number of bits from 'r' and that is the complete hash value.



Advantages

- SHA-3 is ideal for securing embedded subsystems, sensors, consumer electronic devices, and other systems that use symmetric key-based message authentication codes (MACs).
- The algorithm is also faster than its predecessors, with a reported average speed of 12.5 cycles per byte on an Intel Core 2 processor.
- SHA-3 family of cryptographic hash functions are not vulnerable to the "length extension attack".
- It's large computational algorithm improves on its security

Known vulnerabilities or attacks

- The vulnerability is a buffer overflow that allows attacker-controlled values to be eXclusive-ORed (XORed) into memory (without any restrictions on values to be XORed and even far beyond the location of the original buffer), thereby making many standard protection measures against buffer overflows. According to the researcher who identified the problem, **could use the vulnerability to violate the cryptographic properties** of the hash function and find the first and second preimages, as well as determine collisions.

- Also**, the creation of a prototype exploit is announced,** qu**e allows to achieve code execution when calculating the hash** from a specially designed file. The vulnerability can also potentially be used to attack digital signature verification algorithms using SHA-3 (for example, Ed448).

Real world examples

Here are some examples where AES technology is used:

- Digital signatures
- Data integrity verification
- Password hashing
- Blockchain applications
- Secure Hash-Based Message Authentication Codes (HMAC)
- Data forensics and digital evidence
- Cryptographic key derivation

Implementation: (RSA algorithm)

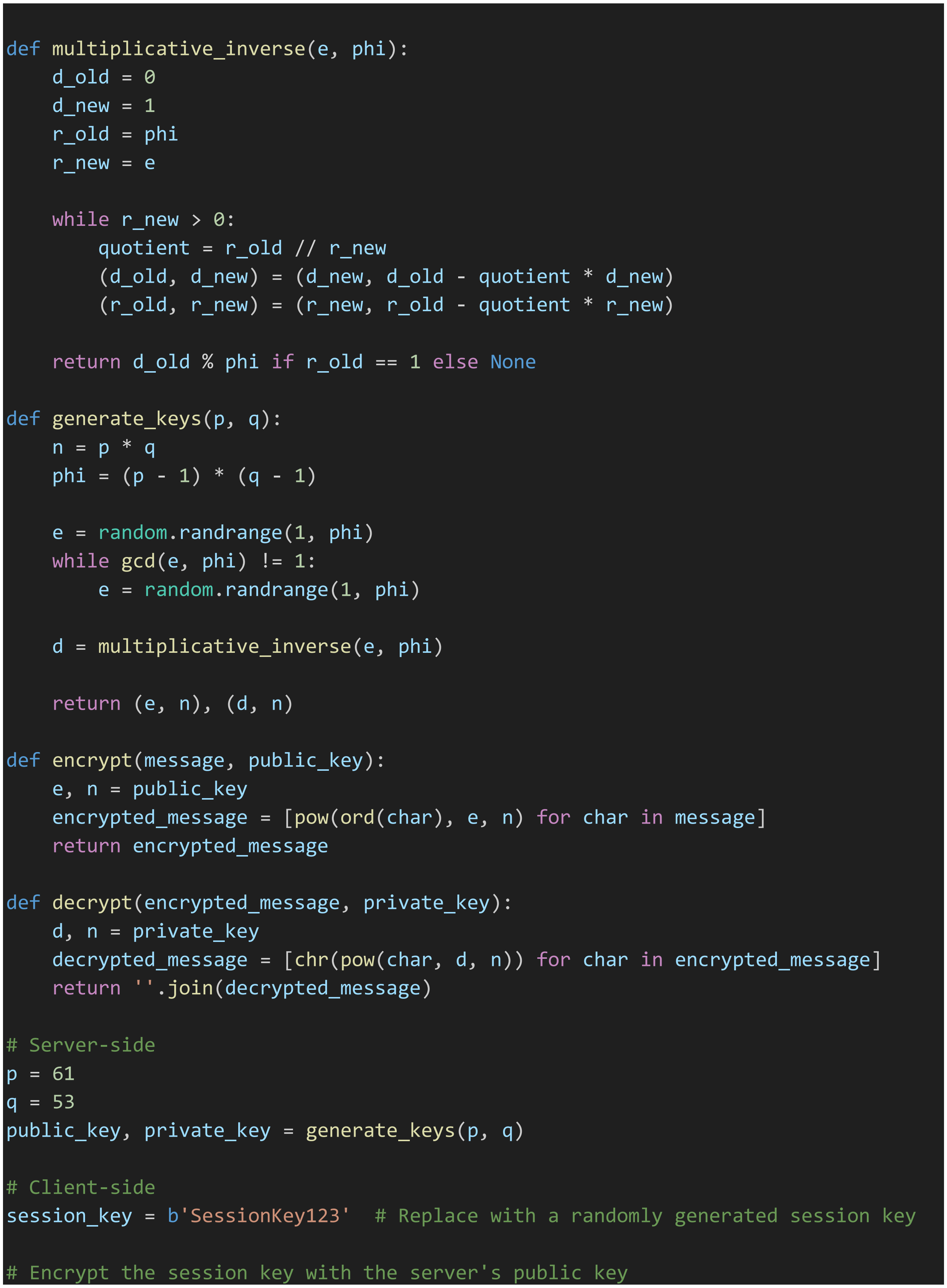
Scenario

One real-life problem that can be solved using the RSA algorithm is secure communication and data exchange between a client and a server over the internet. Here's how RSA can be applied to address this problem:

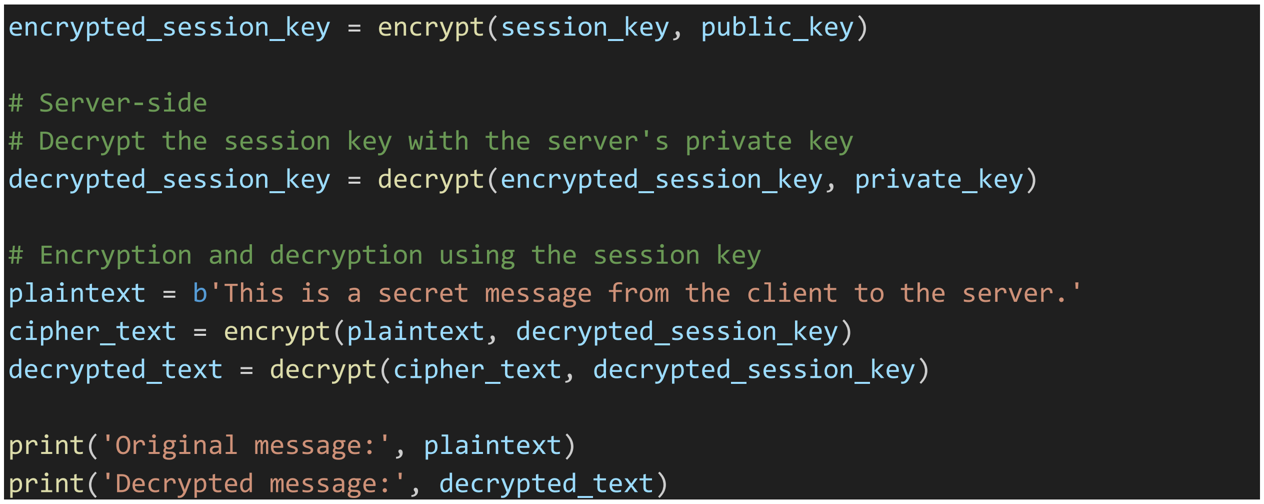
- Key Generation:
  The server generates a pair of RSA keys: a public key and a private key. The public key (e, n) is made available to clients, while the private key (d, n) is kept securely on the server.
- Client Authentication:
  When a client wants to establish a secure connection with the server, the server sends its public key (e, n) to the client. The client verifies the authenticity of the server's public key through digital certificates or other means to ensure it belongs to the intended server.
- Session Key Exchange:
  The client generates a random session key for symmetric encryption (e.g., AES) to achieve faster encryption and decryption compared to RSA. The client encrypts the session key using the server's public key (e, n) obtained in Step 2 and sends it securely to the server.
- Encryption:
  Both the client and server now have the session key. They can use the session key for symmetric encryption to encrypt the actual data being transmitted between them. The encrypted data is sent over the insecure channel.
- Decryption:
  The server receives the encrypted data. It decrypts the data using the session key, which remains confidential since it was only exchanged securely between the client and server. The server can process the decrypted data as required.

By using RSA for client authentication and session key exchange, the server and client can establish a secure communication channel over an insecure network. The symmetric encryption (e.g., AES) using the session key ensures efficient encryption and decryption of the actual data being transmitted. This combination of asymmetric and symmetric encryption provides confidentiality, integrity, and authenticity to the communication, protecting it from eavesdropping and tampering.

```python
import random
import math

def gcd(a, b):
    while b != 0:
        a, b = b, a % b
    return a
```

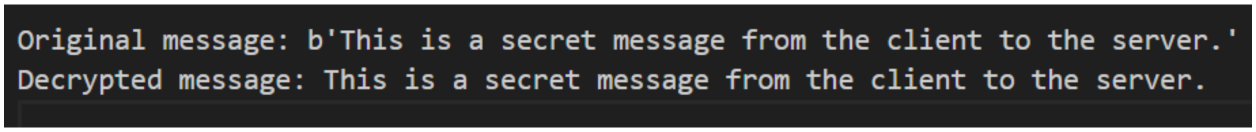


```python
def multiplicative_inverse(e, phi):
    d_old = 0
    d_new = 1
    r_old = phi
    r_new = e

    while r_new > 0:
        quotient = r_old // r_new
        (d_old, d_new) = (d_new, d_old - quotient * d_new)
        (r_old, r_new) = (r_new, r_old - quotient * r_new)

    return d_old % phi if r_old == 1 else None

def generate_keys(p, q):
    n = p * q
    phi = (p - 1) * (q - 1)

    e = random.randrange(1, phi)
    while gcd(e, phi) != 1:
        e = random.randrange(1, phi)

    d = multiplicative_inverse(e, phi)

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

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

# Server-side
p = 61
q = 53
public_key, private_key = generate_keys(p, q)

# Client-side
session_key = b'SessionKey123'  # Replace with a randomly generated session key

# Encrypt the session key with the server's public key
```



```
encrypted_session_key = encrypt(session_key, public_key)

# Server-side
# Decrypt the session key with the server's private key
decrypted_session_key = decrypt(encrypted_session_key, private_key)

# Encryption and decryption using the session key
plaintext = b'This is a secret message from the client to the server.'
cipher_text = encrypt(plaintext, decrypted_session_key)
decrypted_text = decrypt(cipher_text, decrypted_session_key)

print('Original message:', plaintext)
print('Decrypted message:', decrypted_text)
```

output:



```
Original message: b'This is a secret message from the client to the server.'
Decrypted message: This is a secret message from the client to the server.
```

Security Analysis: (attack vectors and countermeasures)

- **Key Generation:**
  The code generates RSA keys by selecting two prime numbers, p and q. The security of RSA relies on the difficulty of factoring large composite numbers, so it is crucial to choose these primes carefully to ensure they are sufficiently large and randomly generated.
  In the provided code, the values of p and q are small prime numbers (61 and 53, respectively) for simplicity. In practice, much larger prime numbers should be used to ensure the security of the RSA keys.

- **Encryption and Decryption:**
  The encryption and decryption operations are implemented using the modular exponentiation (pow) function, which is a fundamental operation in RSA.
  The code converts the plaintext and ciphertext to lists of integers representing the ASCII values of the characters. This approach can be insecure if not properly padded. In practice, a proper padding scheme (such as PKCS#1 v1.5 or OAEP) should be used to avoid vulnerabilities like padding oracle attacks or chosen ciphertext attacks.

- **Randomness:**
  The code uses the random module to generate random values for the session key and the public exponent e. It's important to ensure that the random number generator is cryptographically secure and produces truly random values. In production code, a more

secure random number generator, such as os.urandom() or random.SystemRandom(), should be used.

- **Timing Attacks and Side-Channel Attacks:**
  The code does not address timing attacks or side-channel attacks, which can exploit variations in the execution time or power consumption of cryptographic operations to obtain sensitive information. In a real-world implementation, countermeasures such as constant-time operations and side-channel resistance techniques should be applied.

- **Certificate Verification:**
  The code does not include a mechanism for verifying the authenticity of the public key obtained from the server. In practice, proper certificate validation and trust management should be implemented to ensure that the received public key belongs to the intended server and has not been tampered with.

To address these potential attack vectors, it is recommended to use well-tested and widely accepted cryptographic libraries that provide secure and properly implemented RSA functions, as these libraries have undergone extensive scrutiny and security testing. Additionally, following established best practices for key size, secure random number generation, padding, certificate validation, and side-channel protection is essential for a secure RSA implementation.

However, I could not implement the code using RSA library because of its installation and implementation difficulties.

Conclusion

When it comes to network security, the most important function of a cryptographic functions is to provide integrity for data transmission through a network connection or over the Internet itself. Without this protection, passwords would be vulnerable to interception or modification by third parties which could potentially result in unauthorized access or modification of encrypted files on your computer or device. Therefore, there is a need for architectures that can be proven to be secure and effective.
The term "cybersecurity" refers to the practises, processes, techniques, and resources that are utilised to prevent, detect, and recover from assaults on computer systems, networks, software, and the private information that is stored on them. Ethical hackers deliberately search out systems' vulnerabilities and test whether or not they can be exploited, with the goal of preventing malicious hackers from misusing the system. The practise of probing a system, network, or application for vulnerabilities in the hope that those vulnerabilities might be addressed before an unauthorised hacker or attacker takes use of them is referred to as ethical hacking. As the risk of cyber attacks and the related loss of data continues to rise, businesses are increasingly looking for ways to defend themselves, which has led to a surge in demand for cybersecurity professionals with specialised knowledge.

References

- https://www.linuxadictos.com/en/identificaron-una-vulnerabilidad-en-la-biblioteca-del-algoritmo-sha-3.html
- https://u-next.com/blogs/cyber-security/rsa-algorithm/
- https://www.okta.com/identity-101/rsa-encryption/#:~:text=Where%20Are%20RSA%20Encryption%20Algorithms,of%20messages%20might%20use%20RSA.
- https://www.techtarget.com/searchsecurity/definition/Advanced-Encryption-Standard#:~:text=AES%20is%20implemented%20in%20software,cybersecurity%20and%20electronic%20data%20protection.
- https://www.geeksforgeeks.org/hash-functions-and-list-types-of-hash-functions/
- https://cryptography.io/en/latest/hazmat/primitives/asymmetric/index.html
- https://venafi.com/blog/what-are-best-use-cases-symmetric-vs-asymmetric-encryption/
- https://iq.opengenus.org/hash-functions-examples/