Some Class
Random Examples

Your Name

Contents

Chapter 1

Symmetric Algorithm

Definition 1.0.1: Symmetric Algorithm

A symmetric algorithm, also known as a symmetric-key algorithm or a secret-key algorithm, is a type of cryptographic algorithm used to secure data through encryption and decryption. Unlike asymmetric algorithms (public-key algorithms) which use a pair of keys for encryption and decryption, symmetric algorithms use a single secret key for both processes.

Here's how a symmetric algorithm works:

1.Key Generation:
A secret key is generated by a secure random number generator. This key must be kept confidential between the parties involved in the communication.

2.Encryption:
The plaintext (original) data is combined with the secret key using the encryption algorithm to produce ciphertext (encrypted data). This process is designed to be computationally difficult to reverse engineer without the key.

3.Decryption:
The recipient uses the same secret key and the decryption algorithm to transform the ciphertext back into the original plaintext.

Key features of symmetric algorithms:

Efficiency:
Symmetric algorithms are generally faster and require less computational power than asymmetric algorithms. This makes them suitable for encrypting large amounts of data.

Confidentiality:
Since the same key is used for both encryption and decryption, ensuring the confidentiality of the key is crucial. If the key is compromised, the security of the communication is compromised as well.

Key Distribution:
One challenge with symmetric algorithms is securely distributing the secret key to the parties involved. If an attacker intercepts the key during distribution, they can decrypt the communication.

Scalability:
When multiple parties need to communicate securely, symmetric algorithms can become less practical due to the need for each pair of parties to have a unique secret key.

Common symmetric algorithms include:

Data Encryption Standard (DES):
An older symmetric algorithm that has been largely replaced by AES due to security concerns, but it was historically significant.

Triple DES (3DES):
A variant of DES that applies the DES algorithm three times to each data block, providing increased security.

Advanced Encryption Standard (AES):
A widely used symmetric algorithm, adopted as a standard by the U.S. government, known for its strong security and efficiency.

Blowfish:
A symmetric algorithm designed for efficient software implementation and widely used in various applications.

RC4:
A stream cipher symmetric algorithm used for secure communications and often implemented in software.

Salsa20:
A stream cipher symmetric algorithm known for its speed and security, often used in real-time applications.

It's important to note that while symmetric algorithms are efficient, they require a secure method of key distribution, especially in scenarios where multiple parties need to communicate securely. This is where asymmetric cryptography comes into play, as it helps address the key distribution problem by using public and private key pairs.

1.1 Data Encryption Standard (DES)

The Data Encryption Standard (DES) is a symmetric-key block cipher that played a significant role in the history of cryptography and computer security. Here's an overview of its history:

Origins and Development (Early 1970s):
DES was developed by IBM in the early 1970s for the U.S. National Security Agency (NSA) as a response to a call for encryption standards that could be used to protect sensitive government data. The development of DES was influenced by earlier work on block ciphers and cryptographic techniques.

Standardization (Mid-1970s):
DES was selected as the encryption standard for non-classified government communications in 1973 by the National Institute of Standards and Technology (NIST) (formerly known as the National Bureau of Standards). The process of standardizing DES included public scrutiny and evaluation by the cryptographic community.

Public Release (1975):
DES was published as a Federal Information Processing Standard (FIPS) publication, specifically FIPS PUB 46, in 1975. It became widely used in both government and private sectors for secure communications and data protection.

Popularity and Widespread Use (1970s-1990s):
DES gained popularity as a reliable and efficient encryption algorithm for securing data in various applications, including financial transactions, communication systems, and more. Despite its security limitations, DES remained a widely adopted standard due to its availability and ease of implementation.

Cryptanalysis and Concerns (Late 20th Century):
Over the years, concerns about the security of DES began to arise. Researchers found potential vulnerabilities and weaknesses in the algorithm. Advances in computing power, particularly the development of specialized hardware and techniques, made certain attacks against DES more feasible.

Challenges and Replacements (Late 1990s - Early 2000s):
As DES began to show signs of weakness against modern cryptanalytic techniques, efforts to replace it gained momentum. Triple DES (3DES), a variant of DES that applies the algorithm three times in a row, was introduced as a temporary solution to improve security. It provided a higher level of encryption but was computationally more intensive.

Transition to AES (Early 2000s):
In the late 1990s and early 2000s, the need for a more secure and efficient encryption standard led to the development and selection of the Advanced Encryption Standard (AES) through a competitive process. AES, which uses different algorithms and larger key sizes, gradually replaced DES as the new encryption standard. AES was adopted by NIST in 2001.

Legacy and Impact (Present):
While DES is no longer considered secure for modern cryptographic applications due to its relatively small key size (56 bits), it remains historically significant for its role in the evolution of encryption standards. The challenges and breakthroughs in cryptanalysis of DES contributed to the understanding of cryptographic techniques and the importance of strong encryption.

In summary, the Data Encryption Standard (DES) served as a foundational block cipher and played a crucial role in the development of modern cryptography. Its eventual replacement by the Advanced Encryption Standard (AES) marked a significant milestone in the field of cryptography, leading to stronger and more secure encryption standards.

1.1.1 algorithm

1.Key Generation:
The 56-bit secret key is used to generate 16 round subkeys, one for each round of encryption/decryption. These subkeys are derived from the original key using a process called key scheduling. The key scheduling process involves permutations and transformations of the key bits to create the round subkeys.

2.Initial Permutation (IP):
The 64-bit plaintext block is subjected to an initial permutation, which rearranges the bits according to a predefined pattern.

3.Feistel Network (Rounds):
DES employs a Feistel network structure, where the plaintext block is divided into two halves: a left half (L) and a right half (R).
In each of the 16 rounds, the right half (R) is expanded and XORed with the round subkey. The result is then passed through a series of substitution (S-box) and permutation (P-box) operations.
The substitution boxes (S-boxes) are a critical component of DES, introducing confusion and non-linearity. They map 6-bit inputs to 4-bit outputs using predefined tables.
The permutation (P-box) operations rearrange the bits to introduce diffusion.

4.Swap and Final Permutation:
After all rounds are completed, the left and right halves are swapped, resulting in (R, L). The final ciphertext block is obtained by subjecting (R, L) to a final permutation, which is the inverse of the initial permutation.

5.Decryption:
The decryption process is similar to encryption, with the main difference being the use of the round subkeys in reverse order.

1.1.2 Key Generation:

The key generation process for the Data Encryption Standard (DES) involves several steps to generate the round subkeys used in the Feistel network during encryption and decryption. Here's a high-level overview of the key generation process for DES:

1.Initial Key Permutation (PC-1):
The original 56-bit secret key is permuted using a predefined table called PC-1. This permutation removes parity bits and rearranges the key bits to generate a 56-bit key.

2.Key Splitting:
The 56-bit key is split into two 28-bit halves, often referred to as C (left half) and D (right half).

3.Round Subkey Generation (Shifts and PC-2):
For each of the 16 rounds of DES encryption/decryption, the C and D halves of the key are shifted left by a varying number of positions. The number of positions to shift is determined by the round number.

After shifting, the C and D halves are combined and permuted using another predefined table called PC-2 to generate the round subkey for that round. The round subkey is a 48-bit key used in the Feistel network's substitution and permutation operations.

4.Final Subkey Generation:
After all 16 round subkeys have been generated, they are used in reverse order for decryption. The round subkeys generated in the first step of encryption will be used in the last step of decryption, and so on.

It's important to note that the key generation process is a fundamental part of DES's security. By generating round subkeys for each round, the algorithm introduces complexity and ensures that changes in the key impact multiple rounds, enhancing the security of the encryption process.

Note:-

PC-1 (Initial Key Permutation):
57 49 41 33 25 17 09 01
58 50 42 34 26 18 10 02
59 51 43 35 27 19 11 03
60 52 44 36 63 55 47 39
31 23 15 07 62 54 46 38
30 22 14 06 61 53 45 37
29 21 13 05 28 20 12 04

PC-2 (Second Permutation Choice):
14 17 11 24 01 05 03 28
15 06 21 10 23 19 12 04
26 08 16 07 27 20 13 02
41 52 31 37 47 55 30 40
51 45 33 48 44 49 39 56
34 53 46 42 50 36 29 32

Example 1.1.1

Let's assume our original 56-bit secret key is:
`01101110011011000110100101010101011011100110110100110110011000011`
Initial Key Permutation (PC-1):
The original key is permuted according to PC-1 to obtain:
`11110000110011001010101011111010101010110011000010111010101101101`
Key Splitting:
C half: `11110000110011001010101111`
D half: `01010101010110011000110110 11`
Round Subkey Generation (Shifts and PC-2):
Round 1: C1D1 (shifted left by 1) ⇒ `11100001100110010101010111 11`
Round 1 Subkey (PC-2): `00110110011010011001111011001100011010111110101 0`
(Repeat for each round)
Final Subkey Generation:
Round 16 Subkey: `00110011001110000101010101000010110011111000001 0`
These round subkeys are then used in the Feistel network during encryption and decryption.
Remember that this is a simplified example for educational purposes. In actual DES implementations, the tables and permutations are more complex, and the key generation process ensures that the generated keys are secure and suitable for use in the DES algorithm.

1.1.3 Initial Permutation (IP)

Initial Permutation (IP) is a crucial step in the Data Encryption Standard (DES) algorithm. It is applied to the plaintext data before the encryption process begins and helps to rearrange and shuffle the bits of the input data

in a specific manner. The IP step is the first operation performed on the plaintext before it undergoes the main rounds of DES encryption.
The purpose of the Initial Permutation (IP) is to introduce confusion and diffusion in the plaintext data, making it more resistant to certain types of attacks. Confusion refers to making the relationship between the input and the ciphertext more complex, while diffusion spreads the influence of each plaintext bit across multiple ciphertext bits.

Note:-

IP Table:
58 50 42 34 26 18 10 02
60 52 44 36 28 20 12 04
62 54 46 38 30 22 14 06
64 56 48 40 32 24 16 08
57 49 41 33 25 17 09 01
59 51 43 35 27 19 11 03
61 53 45 37 29 21 13 05
63 55 47 39 31 23 15 07

Example 1.1.2

Plaintext: 1101100010111010010101011100101110100101111101101101011110110011
IP Table:

58	50	42	34	26	18	10	02
60	52	44	36	28	20	12	04
62	54	46	38	30	22	14	06
64	56	48	40	32	24	16	08
57	49	41	33	25	17	09	01
59	51	43	35	27	19	11	03
61	53	45	37	29	21	13	05
63	55	47	39	31	23	15	07

Initial Permutation (IP):

1	0	1	0	1	0	0	0	1	0	0	0	0	1	1	1
0	1	0	1	1	0	0	0	1	0	1	0	1	1	0	1
1	0	0	0	0	0	1	0	1	0	1	1	0	1	1	1
0	1	1	0	0	1	1	1	0	1	0	0	0	1	0	1

The rearranged block after the IP is: 1010100100101100010101101001101011111101100111110111 0

1.1.4 Feistel Network(rounds)

The Feistel Network is a fundamental component of many block cipher encryption algorithms, including the Data Encryption Standard (DES). It is a structure that allows the same encryption and decryption functions to be used, making it suitable for symmetric-key cryptography. The Feistel Network divides the data into blocks and applies a series of transformations through multiple rounds to achieve encryption or decryption.

Here's an overview of how the Feistel Network works:

1.Block Division:
The input data, typically a plaintext or ciphertext block, is divided into two equal halves: the left half (L) and the right half (R).

2.Round Operations:
The Feistel Network consists of multiple rounds (typically 16 rounds in DES). In each round, the following oper-

ations are performed:
The right half (R) is subjected to a round function that takes both the right half and a round subkey as inputs.
The output of the round function is then XORed with the left half (L).
The left and right halves are swapped, with the result of the XOR operation becoming the new right half.

3.Round Subkeys:
For each round, a unique round subkey is derived from the original secret key. These round subkeys are generated through a key schedule and are used in the round function. The round subkeys introduce a mixing and diffusion effect, enhancing the security of the encryption process.

4.Final Round:
After all rounds are completed, the halves are swapped one last time, but without the XOR operation. The final output consists of the right and left halves concatenated together.

The Feistel Network structure offers several benefits:

• Symmetric Operation: The same algorithm is used for both encryption and decryption, requiring only a reversal of the round subkeys' order.

• Confusion and Diffusion: The round function introduces confusion by mixing data and keys, while the swapping and XOR operations create diffusion by spreading the effects of changes throughout the block.

• Parallelizability: The Feistel Network structure allows for parallel processing of the rounds, making it suitable for hardware implementations.

• Avalanche Effect: A small change in input results in drastic changes in output, which is crucial for security.

The Feistel Network is a key building block of many block ciphers, including DES. It ensures that even a simple round function, when applied iteratively, can create complex encryption and decryption processes that are highly secure and resistant to various cryptanalytic attacks.

Example 1.1.3

Plaintext: 11010110
Key: 10101010
Round Function (XOR):

$$\begin{aligned} \text{Right Half (R)} &: 0110 \\ \text{Key} &: 10101010 \\ \text{Output} &: 0110 \oplus 10101010 = 10101100 \end{aligned}$$

Feistel Network Round:

$$\begin{aligned} \text{Initial Plaintext} &: 11010110 \\ \text{Left Half (L)} &: 1101 \\ \text{Right Half (R)} &: 0110 \\ \text{Round Function Output} &: 10101100 \\ \text{New Right Half (R)} &: 10101100 \oplus 1101 = 10100001 \\ \text{New Left Half (L)} &: 0110 \end{aligned}$$

Final Round Output: The final output of the Feistel Network after a single round is:
Encrypted Block: 101000010110

1.1.5 S-box

In the context of the Feistel network used in the Data Encryption Standard (DES) algorithm, an S-box (Substitution box) is a crucial component that introduces confusion and non-linearity to the encryption process. S-boxes are used to perform substitutions of specific groups of bits within each round of the Feistel network. They play a significant role in achieving the cryptographic strength and security of DES.

Here's how S-boxes work within the Feistel network:

1.Substitution:
In each round of the Feistel network, a portion of the data (typically a 6-bit block) is substituted using an S-box. This substitution is based on the input bits to the S-box, which represent specific values.

2.Non-linearity:
S-boxes are designed to be highly non-linear, meaning that small changes in the input bits result in significant changes in the output bits. This non-linearity introduces complexity and makes the encryption process resistant to various types of attacks, such as linear and differential cryptanalysis.

3.Lookup Table:
S-boxes are typically implemented as lookup tables. Each S-box takes a specific input (usually a 6-bit value) and produces a corresponding output (usually a 4-bit value) based on the S-box's design. There are a total of eight S-boxes in DES, each with its own unique lookup table.

4.Key Mixing:
The output of each S-box is then combined with a part of the round key using an XOR operation. This step introduces diffusion, spreading the effects of the key throughout the data.

5.Permutation:
After S-box substitution and key mixing, a permutation operation (P-box) is usually applied to further shuffle the bits. The permutation helps to achieve the avalanche effect, where small changes in the input lead to significant changes in the output.

S-boxes are one of the key reasons why DES is resistant to various types of attacks and provides strong cryptographic security. They ensure that the encryption process is highly complex and that the relationship between the input and the output of each round is difficult to predict.
It's important to note that while S-boxes contribute to the security of DES, they are not immune to modern cryptanalysis techniques. Over time, DES has become less secure due to advances in computing power, and it is no longer recommended for use in practical applications. More modern encryption algorithms with larger key sizes and improved security features are now recommended for secure communication.

Example 1.1.4

S-box 1:

14	04	13	01	02	15	11	08	03	10	06	12	05	09	00	07
00	15	07	04	14	02	13	01	10	06	12	11	09	05	03	08
04	01	14	08	13	06	02	11	15	12	09	07	03	10	05	00
15	12	08	02	04	09	01	07	05	11	03	14	10	00	06	13

Input: 1010
S-box Substitution:

1. Divide the input into row bits (10) and column bits (10).
2. The row bits (10) correspond to decimal 2 (binary 10).
3. The column bits (10) correspond to decimal 2 (binary 10).

4. The value in row 2, column 2 of S-box 1 is 15.

S-box Substitution Output: 1111

1.1.6 P-box

In the context of the Data Encryption Standard (DES) algorithm, the P-box (Permutation box) is a component that is used to further shuffle the bits of a data block. The P-box is applied after the S-box substitutions and key mixing within each round of the Feistel network. Its primary purpose is to provide additional diffusion and confusion to enhance the cryptographic security of the encryption process.

Here's how the P-box works within the DES algorithm:

1.Permutation Operation:
The P-box is a fixed permutation table that determines how the bits of the data block are rearranged. Each bit in the output of the S-boxes is mapped to a new position based on the P-box permutation table. This permutation is deterministic and consistent across all rounds.

2.Avalanche Effect:
The P-box helps achieve the avalanche effect, which means that even a small change in the input data results in a significant change in the output. This property is essential for cryptographic security, as it ensures that small changes in the plaintext lead to unpredictable changes in the ciphertext.

3.Confusion and Diffusion:
Similar to the S-boxes, the P-box contributes to both confusion (complexity) and diffusion (spreading the effects of changes) in the encryption process. The combination of S-boxes and the P-box introduces non-linearity and ensures that the relationship between the input and output becomes highly complex.

4.Key Mixing:
The output of the P-box is typically combined with the other half of the data block (left or right, depending on the Feistel network's design) using an XOR operation. This mixing further ensures that the effect of the round key is distributed throughout the data block.

It's important to note that while the P-box enhances the security of the DES algorithm, modern cryptographic algorithms often use more advanced and sophisticated techniques to achieve security. The DES P-box, along with the other components of the algorithm, contributes to the strength of DES during its time but may not be sufficient against modern cryptanalysis methods.
In summary, the P-box is a key component of the DES algorithm that introduces further permutation and mixing to the data, contributing to the overall complexity and security of the encryption process.

Example 1.1.5

DES P-box (Permutation box):

16	7	20	21
29	12	28	17
1	15	23	26
5	18	31	10
2	8	24	14
32	27	3	9
19	13	30	6
22	11	4	25

P-box Permutation Example:
Input: 01100110111001100011010010110100
Step 1: Apply the P-box Permutation

P-box Positions:	16	7	20	21	29	12	28	17
Input Bits:	0	1	1	1	1	0	0	1
Permuted Bits:	1	0	0	0	0	1	1	1

Step 2: Continue Applying the P-box Permutation

P-box Positions:	1	15	23	26	5	18	31	10
Input Bits:	0	0	1	0	1	0	1	1
Permuted Bits:	0	1	1	0	0	1	1	0

P-box Positions:	2	8	24	14	32	27	3	9
Input Bits:	1	0	1	1	0	0	1	0
Permuted Bits:	0	1	0	1	0	1	1	1

P-box Positions:	19	13	30	6	22	11	4	25
Input Bits:	1	1	0	0	0	1	1	1
Permuted Bits:	0	1	0	0	0	1	1	1

Final Result (Permuted Right Half):
Permuted Right Half: 10000111110011010100111100111000

1.1.7 swap and final permutation

In the Data Encryption Standard (DES) algorithm, both the ”swap” operation and the ”final permutation” are crucial steps that are applied after all the main rounds of the Feistel network to produce the final ciphertext.

Swap Operation (Switch Operation):
After completing all the rounds of the Feistel network, the data block is divided into two halves: the left half (L0) and the right half (R16), where R16 is the result of the final round. The swap operation, also known as the ”switch” operation, involves exchanging the positions of these two halves.

The purpose of the swap operation is to prepare the data for the final permutation (FP) step and for generating the ciphertext. This swap ensures that the ciphertext will be composed of the right half followed by the left half, which is the opposite of the initial arrangement at the beginning of encryption. The swap is essential for maintaining consistency between encryption and decryption processes, as the same swap is performed in reverse during decryption.

Final Permutation (FP):
The final permutation, also known as FP, is the last step in the DES encryption process. It is applied to the entire data block (the result of the swap operation) to rearrange the bits and produce the final ciphertext. The purpose of the final permutation is to achieve additional diffusion and confusion, enhancing the security of the encrypted data.

The FP operation employs a fixed permutation table that shuffles the bits of the data block. Similar to the initial permutation (IP) and other permutations used in the algorithm, the FP step contributes to making the relationship between the plaintext and ciphertext complex and difficult to analyze.

In summary:
The swap operation exchanges the positions of the left and right halves of the data block after all rounds, preparing for the final permutation and ciphertext generation. The final permutation (FP) rearranges the bits of the entire data block to produce the final ciphertext, adding another layer of diffusion and confusion.
It's important to note that while these steps were designed to enhance the security of DES, the algorithm is now considered outdated and insecure due to advances in cryptanalysis and computing power. As a result, more modern encryption algorithms with larger key sizes and better security features are recommended for secure communication.

Example 1.1.6

Initial Plaintext:

11001110 01011010 11001001 11011011 00111010 10110011 00110110 01101110

Swap Operation:

00110110 01101110 11001110 01011010 11001001 11011011 00111010 10110011

Final Permutation (FP):

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

Final Result (Ciphertext):

00111001 10011011 11001011 11001000 01011011 11011110 10000101 01110110

1.1.8 Decryption

Decryption in the Data Encryption Standard (DES) algorithm is essentially the reverse process of encryption. It involves applying the same steps as encryption, but in reverse order, to transform the ciphertext back into the original plaintext using the same key that was used for encryption. The decryption process in DES consists of the following steps:

1.Initial Steps:
Start with the ciphertext that you want to decrypt.
Use the same secret key that was used for encryption.

2.Initial Permutation (IP):
Apply the initial permutation (IP) to the ciphertext to rearrange its bits. This step prepares the ciphertext for processing through the Feistel network.

3.Feistel Network Decryption:
Similar to encryption, perform a series of rounds in the Feistel network, but in reverse order. Each round consists of the following steps:
- Perform an expansion permutation on the right half of the data.
- XOR the expanded right half with the round subkey in reverse order.
- Apply S-box substitutions in reverse order.
- Perform a P-box permutation in reverse order.
- XOR the output of the P-box with the original left half.

• Swap the left and right halves of the data (reverse of swap during encryption).

4.Final Round:
After completing all the Feistel rounds, you'll have the result of the final round (L16 and R16).

5.Inverse Initial Permutation (IP^{-1}):
Apply the inverse initial permutation (IP^{-1}) to the final round result to rearrange the bits and produce the decrypted plaintext.

6.Final Steps:
The result of the inverse initial permutation is the decrypted plaintext.

It's important to note that while the decryption process follows the same steps as encryption, certain steps are reversed or applied in reverse order. Additionally, the subkeys used for decryption are derived from the original key in reverse order as well.
In summary, decryption in the DES algorithm involves applying the inverse of the encryption process, including reversing the Feistel network rounds, using the same secret key. The resulting plaintext should be identical to the original plaintext before encryption.

Example 1.1.7

Ciphertext to Decrypt:

Ciphertext:
0011100110011011110010111101101011001110111100001010111011000101

Inverse Initial Permutation (IP^{-1}):

Ciphertext after IP^{-1}:
1101100000100101011100011101010101100111001010111100110011011001

Feistel Network Decryption:

Result after Rounds (L0 and R0):
(Detailed steps for each round)

Swap Operation (Switch):

Swapped Result:
Right Half Left Half

Inverse Initial Permutation (IP^{-1}):

Original Plaintext:
0110011011100110001101001011010001100100110010011101101100111010

1.2 Advanced Encryption Standard (AES)

The Advanced Encryption Standard (AES) is a symmetric encryption algorithm that was established to provide a higher level of security and efficiency than its predecessor, the Data Encryption Standard (DES). The history of AES involves a comprehensive selection process, public scrutiny, and international collaboration. Here's a detailed overview of the history of AES:

1.Need for a New Standard:
By the late 20th century, the limitations of DES were becoming increasingly apparent. DES used a relatively short 56-bit key, which was vulnerable to advances in computing power and cryptanalysis. As a result, there was a growing need for a new encryption standard that could offer higher security while being efficient and adaptable to various platforms.

2.Initiation of AES Selection Process:
In 1997, the U.S. National Institute of Standards and Technology (NIST) initiated a public competition to select a new encryption algorithm that would serve as the successor to DES. The competition was open to cryptographic experts worldwide, and the goal was to identify a strong and secure algorithm that could be adopted as an international standard.

3.Candidate Algorithms:
During the AES selection process, a total of 15 candidate algorithms were submitted by researchers and cryptographers from around the world. These algorithms represented a diverse range of designs and approaches to encryption.

4.Evaluation and Analysis:
The submitted algorithms underwent rigorous evaluation and analysis in multiple rounds. NIST conducted thorough testing and analysis to assess each algorithm's security, efficiency, and suitability for different applications. The cryptographic community extensively reviewed the algorithms and provided feedback.

5.Criteria for Selection:
The AES selection criteria included security, performance, implementation complexity, and flexibility in terms of key lengths and block sizes. NIST sought an algorithm that would provide a high level of security against a wide range of cryptographic attacks while remaining practical for real-world use.

6.Selection of Rijndael:
After multiple rounds of evaluation, the algorithm Rijndael, designed by Belgian cryptographers Vincent Rijmen and Joan Daemen, emerged as the clear winner. Rijndael demonstrated strong security properties, efficient software and hardware implementations, and flexibility in supporting different key lengths and block sizes.

7.Standardization and Adoption:
In November 2001, NIST announced Rijndael as the Advanced Encryption Standard (AES) in its publication FIPS 197. AES quickly gained widespread adoption and acceptance as a global encryption standard. Its selection was based on a combination of its cryptographic strength, performance, and versatility.

8.Ongoing Security Analysis:
AES has since undergone extensive cryptanalysis by the cryptographic community, and no practical vulnerabilities have been found. Its security has held up remarkably well, making it a cornerstone of modern information security.

9.Global Impact:
AES is used in a wide range of applications, from securing data at rest to protecting communications over the internet. Its impact is felt across industries and sectors, including finance, healthcare, government, and more.

In summary, the history of AES reflects a collaborative effort by the cryptographic community to create a strong, efficient, and widely adopted encryption standard. AES has become a foundational component of modern cybersecurity, ensuring the confidentiality and integrity of sensitive information in an increasingly digital world.

1.2.1 algorithm

1.Block Size and Key Length:
AES operates on fixed-size blocks of data, with a block size of 128 bits (16 bytes). The key length can be 128, 192, or 256 bits, depending on the specific variant of AES being used.

2.Key Expansion:
Before encryption or decryption begins, the original encryption key is expanded into a set of round keys. These round keys are derived through a process called the "Key Expansion" routine. The number of rounds performed during encryption or decryption depends on the key length: 10 rounds for a 128-bit key, 12 rounds for a 192-bit key, and 14 rounds for a 256-bit key.

3.SubBytes Transformation:
In each encryption round, the input data block undergoes a byte-by-byte substitution. Each byte is replaced with a corresponding value from a fixed substitution table called the "SubBytes" table. This step introduces confusion and helps prevent patterns from being preserved in the ciphertext.

4.ShiftRows Transformation:
The "ShiftRows" step involves cyclically shifting the rows of the data block. This mixing operation ensures that even small changes in the input data produce significant differences in the ciphertext.

5.MixColumns Transformation (except for the last round):
In all rounds except the last one, the "MixColumns" step operates on the columns of the data block. It combines bytes in a way that provides diffusion, further increasing the complexity of the relationship between the plaintext and the ciphertext.

6.AddRoundKey Transformation: At the beginning of each round, the data block is combined (bitwise XOR) with the round key derived from the original key using the Key Expansion routine. This step introduces the secret key into the encryption process.

7.Final Round:
In the final round, the "MixColumns" transformation is omitted, and the "SubBytes," "ShiftRows," and "AddRoundKey" transformations are applied. This round produces the final ciphertext.

8.Decryption:
Decryption with AES is the reverse process of encryption. The round keys are used in reverse order, and the decryption steps are the inverse of the encryption steps. "InvSubBytes," "InvShiftRows," and "InvAddRoundKey" transformations are used in decryption, and the "InvMixColumns" transformation is applied except for the last round.

9.Security and Strength:
AES is known for its strong security properties and resistance to various cryptographic attacks. Its design and the number of rounds provide a high level of protection against known attacks.

AES operates efficiently in both hardware and software implementations, making it suitable for a wide range of applications, from secure communication protocols to data encryption at rest. It has become a cornerstone of modern cryptography and continues to be widely used across industries and sectors.

1.2.2 Block Size and Key Length

Block Size and Key Length are important parameters in encryption algorithms like the Advanced Encryption Standard (AES). They determine the size of the data blocks and the length of the encryption keys used in the algorithm. Let's delve deeper into these concepts:

1.Block Size:
Block size refers to the fixed length of data that the encryption algorithm processes at once. In AES, the block size is 128 bits, which is equivalent to 16 bytes. This means that AES operates on data blocks that are exactly 128 bits in length. The plaintext is divided into these fixed-size blocks, and each block is processed independently through the encryption or decryption process.
The fixed block size is a fundamental characteristic of block ciphers like AES. It affects how data is divided, processed, and encrypted. Larger block sizes offer higher security against certain types of attacks but may be less efficient in terms of speed and resource usage. Smaller block sizes might be faster but could be more susceptible to certain cryptographic vulnerabilities.

2.Key Length:
Key length refers to the size of the secret encryption key used by the algorithm. In AES, the key length can be

128, 192, or 256 bits. The choice of key length has a significant impact on the security of the encryption. Generally, longer keys provide stronger security because they increase the number of possible keys that an attacker needs to try in order to break the encryption.
The different key lengths in AES correspond to different numbers of rounds performed during the encryption and decryption processes. AES-128 uses 10 rounds, AES-192 uses 12 rounds, and AES-256 uses 14 rounds. More rounds typically offer higher security but may come at the cost of slightly slower performance.
It's worth noting that the security of AES is highly dependent on the key length. As computing power increases over time, longer key lengths are often recommended to maintain a high level of security against potential attacks.

Both block size and key length are critical design choices in encryption algorithms. They influence the algorithm's security, speed, and resource requirements. When selecting AES parameters for a specific application, it's important to consider the trade-offs between security and performance based on the desired level of protection and the computing resources available.

1.2.3 Key Expansion

Key expansion is a critical step in symmetric encryption algorithms like the Advanced Encryption Standard (AES). It involves deriving a set of round keys from the original encryption key. These round keys are used in each round of the encryption (and decryption) process to introduce the secret key material into the algorithm's operations. Key expansion ensures that each round has a unique subkey that adds complexity and security to the encryption process.

Here's how key expansion works in the context of AES:

1.Original Encryption Key:
The key expansion process starts with the original encryption key. In AES, the key length can be 128, 192, or 256 bits, depending on the chosen variant (AES-128, AES-192, or AES-256). This original key serves as the basis for deriving the round keys.

2.Generating Round Keys:
The key expansion process involves generating a set of round keys, one for each round of encryption. Each round key is derived from the previous round key and the original key. The process includes the following steps:
a. Initial Round Key:
The initial round key is simply the original encryption key itself. For example, in AES-128, the initial round key is the 128-bit key provided.
b. Key Schedule:
The key schedule involves performing transformations on the previous round key to generate the next round key. The specific transformations used depend on the key length and the number of rounds.
c. Substitution and Mixing:
The key schedule typically involves applying byte substitutions, bitwise operations, and mixing to the previous round key to generate the next round key. These operations help ensure that each round key is distinct and introduces variation into the encryption process.

3.Number of Rounds:
The number of rounds and the key expansion process vary based on the AES variant being used. AES-128 uses 10 rounds, AES-192 uses 12 rounds, and AES-256 uses 14 rounds. In each round, a distinct subkey derived from the key expansion process is used.

4.Application in Encryption (and Decryption):
In each round of AES encryption (and decryption), the derived round key is combined with the data block using bitwise XOR. This step introduces the secret key material into the encryption process, ensuring that each round operates on different data.

Key expansion plays a crucial role in ensuring the security and effectiveness of AES. By generating a unique

set of round keys for each round, AES achieves a high degree of confusion and diffusion, making the relationship between the plaintext, the key, and the ciphertext complex and resistant to cryptographic attacks.
It's important to note that while key expansion is a fundamental component of AES, the specific details of the key expansion process can be quite complex and involve various bitwise operations, substitution tables, and mathematical transformations. The exact mechanisms used in the key expansion process depend on the chosen key length and AES variant.

Note:-

S-box:

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	63	7C	77	7B	F2	6B	6F	C5	30	01	67	2B	FE	D7	AB	76
1	CA	82	C9	7D	FA	59	47	F0	AD	D4	A2	AF	9C	A4	72	C0
2	B7	FD	93	26	36	3F	F7	CC	34	A5	E5	F1	71	D8	31	15
3	04	C7	23	C3	18	96	05	9A	07	12	80	E2	EB	27	B2	75
4	09	83	2C	1A	1B	6E	5A	A0	52	3B	D6	B3	29	E3	2F	84
5	53	D1	00	ED	20	FC	B1	5B	6A	CB	BE	39	4A	4C	58	CF
6	D0	EF	AA	FB	43	4D	33	85	45	F9	02	7F	50	3C	9F	A8
7	51	A3	40	8F	92	9D	38	F5	BC	B6	DA	21	10	FF	F3	D2
8	CD	0C	13	EC	5F	97	44	17	C4	A7	7E	3D	64	5D	19	73
9	60	81	4F	DC	22	2A	90	88	46	EE	B8	14	DE	5E	0B	DB
A	E0	32	3A	0A	49	06	24	5C	C2	D3	AC	62	91	95	E4	79
B	E7	C8	37	6D	8D	D5	4E	A9	6C	56	F4	EA	65	7A	AE	08
C	BA	78	25	2E	1C	A6	B4	C6	E8	DD	74	1F	4B	BD	8B	8A
D	70	3E	B5	66	48	03	F6	0E	61	35	57	B9	86	C1	1D	9E
E	E1	F8	98	11	69	D9	8E	94	9B	1E	87	E9	CE	55	28	DF
F	8C	A1	89	0D	BF	E6	42	68	41	99	2D	0F	B0	54	BB	16

Example 1.2.1 (arg1)
arg2

1.2.4 SubBytes Transformation

The SubBytes transformation is a fundamental step in the Advanced Encryption Standard (AES) algorithm. It is a byte substitution operation that provides non-linearity and confusion during the encryption process. In SubBytes, each byte of the input state matrix is replaced with a corresponding byte from the AES S-box (Substitution box).

Here's how the SubBytes transformation works:

AES S-box Lookup:
The SubBytes transformation operates on each byte of the input state matrix individually. For each byte, the corresponding byte in the AES S-box is looked up based on the row and column of the byte's hexadecimal value.

Row and Column Substitution:
The byte's hexadecimal value is split into two parts: the leftmost 4 bits determine the row, and the rightmost 4 bits determine the column in the AES S-box. The value at the intersection of the determined row and column is the new byte value that replaces the original byte.

Example 1.2.2

32	88	31	e0
43	5a	31	37
f6	30	98	07
a8	8d	a2	34

$\Rightarrow$

23	2c	8e	63
a2	89	8e	97
a5	10	53	4a
63	e0	c1	7b

.

The SubBytes transformation adds a layer of confusion to the data, making it harder to deduce relationships between the input and output bytes. This, along with other AES operations, contributes to the algorithm's strength and security.
It's important to note that the AES S-box is a fixed lookup table, and its values are determined by a specific mathematical construction to provide desirable cryptographic properties, including non-linearity and resistance against various attacks.

1.2.5 ShiftRows Transformation

The ShiftRows transformation is one of the steps in the Advanced Encryption Standard (AES) algorithm that provides diffusion by shuffling the bytes within each row of the state matrix. This step is crucial for ensuring that patterns in the plaintext do not persist through the encryption process, contributing to the algorithm's security.

Here's how the ShiftRows transformation works:

State Matrix:
The input to the ShiftRows transformation is the state matrix, which is a 4x4 array of bytes representing the current state of the data being encrypted.

Byte Shifting:
In the ShiftRows transformation, each row of the state matrix is circularly shifted to the left by a certain number of bytes. The number of shifts depends on the row index:
- The first row (row 0) is not shifted.
- The second row (row 1) is shifted to the left by 1 byte.
- The third row (row 2) is shifted to the left by 2 bytes.
- The fourth row (row 3) is shifted to the left by 3 bytes.

Example 1.2.3

23	2c	8e	63
a2	89	8e	97
a5	10	53	4a
63	e0	c1	7b

$\Rightarrow$

23	2c	8e	63
89	8e	97	a2
53	4a	a5	10
7b	63	e0	c1

.

The ShiftRows transformation ensures that each byte in a row interacts with different bytes in the subsequent rounds, spreading out the influence of each byte across the entire state matrix. This enhances the confusion and diffusion properties of the AES algorithm, making it more resistant to various cryptographic attacks.
It's important to note that the ShiftRows transformation is a simple byte rearrangement within each row and does not involve complex mathematical operations like substitution or mixing.

1.2.6 MixColumns Transformation

The MixColumns transformation is a step in the Advanced Encryption Standard (AES) algorithm that provides diffusion by mixing the columns of the state matrix. This transformation enhances the cryptographic strength of AES by ensuring that each byte in the output state matrix depends on multiple bytes of the input, making it more resistant to attacks.

Here's how the MixColumns transformation works:

State Matrix:
The input to the MixColumns transformation is the state matrix, which is a 4x4 array of bytes representing the current state of the data being encrypted.

Matrix Multiplication:
Each column of the state matrix is treated as a polynomial over the finite field GF(2^8). The MixColumns transformation involves multiplying the column by a fixed matrix, resulting in a new column. This matrix multiplication is performed modulo a fixed polynomial to ensure that the result remains within the finite field.

Note:-

Fixed Matrix:

$$\begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix}$$

Example 1.2.4

23	2c	8e	63
89	8e	97	a2
53	4a	a5	10
7b	63	e0	c1

$\Rightarrow$

69	6a	08	c9
9b	8d	0f	81
98	d8	84	32
18	6c	72	01

.

The MixColumns transformation combines bytes from each column in a complex manner, ensuring that no single byte directly determines the transformation. This contributes to the avalanche effect, where a small change in the input leads to significant changes in the output.
It's important to note that the MixColumns transformation involves Galois field multiplication, which is a non-trivial mathematical operation over a finite field. The fixed matrix used in AES MixColumns is carefully chosen to provide the desired cryptographic properties while maintaining efficient implementation.

1.2.7 AddRoundKey Transformation

The AddRoundKey transformation is a key mixing operation in the Advanced Encryption Standard (AES) algorithm. It is an essential step that adds the current round's subkey to the state matrix. This transformation combines the state matrix with the round key, ensuring that each byte of the state matrix interacts with the corresponding byte of the round key.

Here's how the AddRoundKey transformation works:

State Matrix and Round Key:
The input to the AddRoundKey transformation is the current state matrix, which represents the intermediate result after previous transformations. Each byte of the state matrix is combined with the corresponding byte of the round key for the current round.

Key Mixing:
The AddRoundKey operation is a simple bitwise XOR (exclusive OR) operation. Each byte of the state matrix is XORed with the corresponding byte of the round key. This effectively combines the state matrix and the round key, introducing the key material into the encryption process.

Example 1.2.5

69	6a	08	c9
9b	8d	0f	81
98	d8	84	32
18	6c	72	01

$\oplus$

2b	7e	15	16
28	ae	d2	a6
ab	f7	97	46
e5	46	52	6d

$\Downarrow$

42	14	1d	df
b3	23	12	27
3b	2f	13	74
fd	2a	20	6c

.

The AddRoundKey transformation introduces the key's entropy into the state matrix, ensuring that each byte is influenced by the corresponding byte of the round key. This operation is crucial for the algorithm's security, as it combines the effects of previous transformations (SubBytes, ShiftRows, MixColumns) with the unique properties of the round key for each round.
In AES, the AddRoundKey transformation is performed at the beginning of each round during both encryption and decryption processes. It is a reversible operation, meaning that it can be undone during decryption by applying the same round key again.

1.2.8 Decryption

Decryption in the Advanced Encryption Standard (AES) is the process of reversing the encryption steps to recover the original plaintext from the encrypted ciphertext. AES uses a similar structure for decryption as it does for encryption, but the transformation steps are applied in reverse order, and the round keys are used in reverse order as well.

The decryption process in AES involves the following steps:

1.Key Expansion (Inverse):
Just like in encryption, the decryption process starts with key expansion. However, for decryption, the round keys are used in reverse order. The last round key used in encryption becomes the first round key for decryption, and so on.

2.Initial Round:
• AddRoundKey (Inverse): The last round key used in encryption is added to the ciphertext to recover the state.
• ShiftRows (Inverse): The rows of the state matrix are shifted to the right.

3.Main Rounds (Inverse):
• MixColumns (Inverse): The MixColumns transformation is reversed by applying the inverse matrix multiplication.
• SubBytes (Inverse): The SubBytes transformation is reversed by applying the inverse S-box substitution.
• AddRoundKey (Inverse): The round key used in encryption is added to the state matrix.

4.Final Round:

• ShiftRows (Inverse): The rows of the state matrix are shifted to the right.
• AddRoundKey (Inverse): The original encryption key (first round key) is added to the state matrix.

5.Output:
The resulting state matrix after the final round represents the decrypted plaintext.

It's important to note that while the transformations are applied in reverse order, the specific transformations used in decryption are the inverses of those used in encryption. For example, the inverse S-box is used in SubBytes (Inverse), and the inverse MixColumns transformation is used in MixColumns (Inverse).
In summary, AES decryption undoes the encryption process by reversing the steps, starting with the application of the round keys in reverse order and performing the inverse transformations. The result is the recovery of the original plaintext.

Chapter 2

Asymmetric Algorithm

Definition 2.0.1: Asymmetric Algorithm

Asymmetric cryptography, also known as public-key cryptography, is a fundamental concept in modern cryptography that involves the use of pairs of cryptographic keys to secure communication and digital transactions. Unlike symmetric cryptography, where the same key is used for both encryption and decryption, asymmetric cryptography uses two distinct keys: a public key and a private key. This approach provides a higher level of security and enables various cryptographic operations.

Here's how asymmetric cryptography works:

1.Key Pair Generation:
In asymmetric cryptography, a user generates a pair of keys – a public key and a private key – using a specific algorithm. The keys are mathematically related, but it's computationally infeasible to derive one key from the other. The public key can be shared openly, while the private key must be kept secret.

2.Encryption:
If User A wants to send a secure message to User B, User A can encrypt the message using User B's public key. Once encrypted with the public key, only User B's corresponding private key can decrypt the message. This means that even if someone intercepts the encrypted message, they won't be able to decipher it without the private key.

3.Decryption:
User B, who possesses the private key, can decrypt the received message encrypted with their public key. Since the private key is kept secret, only User B can perform this decryption.

4.Digital Signatures:
Asymmetric cryptography is also used for digital signatures. User A can create a digital signature by encrypting a hash of the message with their private key. User B, upon receiving the message and signature, can use User A's public key to verify the signature's authenticity. If the decrypted hash matches the hash of the received message, it indicates that the message hasn't been tampered with and originates from User A.

5.Key Exchange: Asymmetric cryptography solves the key distribution problem in symmetric cryptography. When two parties want to establish a secure communication channel, they can use asymmetric encryption to exchange a symmetric key securely. This symmetric key can then be used for subsequent communication using faster symmetric encryption algorithms.

Notable asymmetric encryption algorithms include RSA (Rivest-Shamir-Adleman) and ECC (Elliptic Curve Cryptography). RSA is based on the difficulty of factoring large composite numbers, while ECC relies on the difficulty of solving the elliptic curve discrete logarithm problem.
Asymmetric cryptography offers enhanced security because the private key never needs to be shared or transmitted, reducing the risk of exposure. However, it's computationally more intensive than symmetric cryptography, making it less suitable for encrypting large amounts of data. As a result, a common approach is to use asymmetric encryption for key exchange and digital signatures, and then use symmetric encryption for actual data transmission once the keys are securely established.

2.1 RSA(Rivest-Shamir-Adleman)

RSA (Rivest-Shamir-Adleman) is one of the most widely used and well-known asymmetric encryption algorithms in the field of cryptography. Its history traces back to the late 1970s and involves the contributions of three individuals: Ron Rivest, Adi Shamir, and Leonard Adleman. Here's a brief overview of the history of RSA:

Early Developments:
In the 1970s, researchers were exploring the concept of public-key cryptography, where different keys were used for encryption and decryption. However, no practical implementation had been devised. Ron Rivest, a computer

scientist and cryptographer, started working on the problem with his colleagues.

Invention of RSA:
In 1977, Ron Rivest, Adi Shamir, and Leonard Adleman jointly introduced the RSA algorithm. RSA was named after the first letter of each of their last names. The algorithm is based on the mathematical properties of large prime numbers, specifically the difficulty of factoring the product of two large primes.

Fundamental Idea:
The fundamental idea behind RSA is the use of two related keys: a public key for encryption and a private key for decryption. The security of RSA relies on the mathematical difficulty of factoring large composite numbers into their prime factors. While multiplication is relatively easy, factoring the product back into its primes is computationally difficult, especially for large prime numbers.

Public Release:
The RSA algorithm was initially published in a paper titled "A Method for Obtaining Digital Signatures and Public-Key Cryptosystems" in 1978. The paper introduced the concept of public-key cryptography and described the RSA encryption and digital signature schemes.

Commercialization and Widespread Use:
RSA quickly gained attention for its practical applications in secure communication and digital signatures. In the early 1980s, RSA Data Security, Inc. (now RSA Security LLC) was founded to commercialize the RSA algorithm and develop cryptographic products. The algorithm became a crucial component of secure communication protocols, digital certificates, and various cryptographic applications.

Impact and Influence: RSA revolutionized the field of cryptography by providing a practical solution to the key distribution problem. It enabled secure communication and data protection on a global scale. The concept of public-key cryptography introduced by RSA laid the foundation for the development of other cryptographic algorithms and protocols.

Evolution and Advancements:
Over the years, there have been advancements and refinements in RSA and related algorithms. Researchers have explored different key sizes and optimizations to adapt RSA to changing security requirements and computational capabilities.

While RSA remains a widely used algorithm, it's worth noting that as computing power has increased, the security of smaller key sizes has been compromised. As a result, larger key sizes are recommended to maintain the desired level of security. Additionally, more recent developments in cryptography, such as elliptic curve cryptography, have provided alternatives to RSA for certain applications.

2.1.1 Algorithm

1.Key Generation:
• Choose two distinct large prime numbers, p and q.
• Calculate their product, $n = p * q$. This value n is used as the modulus for both the public and private keys.
• Compute $\phi(n)$ (phi of n), where $\phi(n) = (p-1) * (q-1)$. This is Euler's totient function and represents the count of positive integers less than $n$that are coprime to n.
• Select an integer e (the public exponent) such that $1 < e < \phi(n)$ and e is coprime to $\phi(n)$. The public key consists of the pair (n, e).
• Compute the modular multiplicative inverse of e modulo $\phi(n)$, denoted as d. This is the private exponent. The private key consists of the pair (n, d).

2.Encryption:
• To send a message M, the sender obtains the recipient's public key (n, e).
• The sender converts the message M into a numerical value m, where $0 \leqslant m < n$.
• The sender computes the ciphertext c using the encryption formula: $c = m^e mod n$.

• The sender sends the ciphertext c to the recipient.

3.Decryption:
• The recipient uses their private key (n, d) to decrypt the ciphertext c.
• The recipient computes the decrypted numerical value m using the decryption formula: $m = c^d mod n$.
• The recipient converts the numerical value m back to the original message M.

RSA's security is based on the difficulty of factoring the product $n = p * q$ back into its prime factors p and q. If an attacker could efficiently factor n, they could compute the private key and break the encryption. As of now, no efficient algorithm for factoring large semiprime numbers (numbers with exactly two prime factors) is known, making RSA secure.

Additionally, RSA is used for digital signatures:

To create a digital signature, a sender computes a hash value of the message and then encrypts this hash using their private key. The recipient can verify the authenticity of the signature by decrypting the encrypted hash using the sender's public key and comparing it to a hash value they compute themselves.

While RSA is a powerful algorithm, its security relies on the appropriate choice of key sizes. As computing power increases over time, larger key sizes are needed to maintain the same level of security. In recent years, there has been interest in post-quantum cryptography due to the potential future impact of quantum computers on RSA and other traditional cryptographic schemes.

Example 2.1.1

Key Generation:
Let's choose two prime numbers: $p = 17$ and $q = 19$.
Calculate $n = p \cdot q = 17 \cdot 19 = 323$.
Compute $\phi(n) = (p - 1) \cdot (q - 1) = 16 \cdot 18 = 288$.
Choose a public exponent $e = 5$ (a common choice).
Compute the private exponent d such that $d \cdot e \equiv 1 \pmod{\phi(n)}$. In this case, $d = 173$.
Public key: $(n = 323, e = 5)$
Private key: $(n = 323, d = 173)$
Encryption:
Let's encrypt the message "HELLO".
H = 72, E = 69, L = 76, O = 79.
For simplicity, we'll combine these values into a single number: $m = 72697679$.
To encrypt m using the recipient's public key (n, e):

$$c = m^e \mod n = 72697679^5 \mod 323 = 132.$$

The encrypted ciphertext c is 132.
Decryption:
The recipient uses their private key (n, d) to decrypt the ciphertext.
To decrypt c using the private key (n, d):

$$m = c^d \mod n = 132^{173} \mod 323 = 72697679.$$

Convert the numerical value 72697679 back to the original message: H = 72, E = 69, L = 76, O = 79.
Therefore, the decrypted message is "HELLO".

2.2 Digital Signature Algorithm (DSA)

The Digital Signature Algorithm (DSA) is a widely used digital signature scheme that was developed as a part of the Digital Signature Standard (DSS) by the U.S. National Institute of Standards and Technology (NIST) in the early 1990s. DSA was designed to provide a secure and efficient method for creating and verifying digital signatures, which are crucial for ensuring the authenticity and integrity of digital documents and messages. Here's a brief overview of the history of DSA:

Origins of Digital Signatures:
The concept of digital signatures dates back to the late 1970s and early 1980s when researchers were exploring ways to apply cryptography to provide data integrity, authenticity, and non-repudiation in the digital realm. Asymmetric cryptography, with its public-key and private-key pair, provided a promising foundation for creating digital signatures.

Development of DSA:
In the early 1990s, as part of efforts to establish standardized cryptographic algorithms, NIST developed the Digital Signature Standard (DSS) to address the need for secure digital signatures. The DSS specified the use of the Digital Signature Algorithm (DSA) for creating and verifying digital signatures. DSA was intended to provide a secure alternative to existing signature schemes like the RSA digital signature scheme.

Introduction and Adoption:
DSA was first published as a draft standard by NIST in 1991 (Federal Information Processing Standard, FIPS 186). The final version, FIPS 186-1, was published in 1994. DSA was adopted for use in various applications that required digital signatures, including secure email, electronic transactions, and digital certificates.

Limitations and Criticisms:
While DSA was designed to be secure and efficient, it did face some criticisms and limitations. One notable limitation was its reliance on the discrete logarithm problem in finite fields, which could potentially be vulnerable to advances in cryptanalysis and future quantum computers. Additionally, DSA's key sizes needed to be carefully chosen to ensure security.

DSA in Practice:
Despite some limitations, DSA found practical use in various applications, especially where government regulations required the use of standardized cryptographic algorithms. It was used in digital certificates and secure communication protocols.

Evolutions and Replacements:
Over time, concerns about DSA's limitations and potential vulnerabilities led to the development of more advanced digital signature algorithms, such as ECDSA (Elliptic Curve Digital Signature Algorithm). ECDSA offers similar security with smaller key sizes, making it more efficient for resource-constrained devices. ECDSA has gained wider adoption, especially in modern cryptographic systems.

Legacy and Transition:
While DSA's popularity has diminished compared to newer algorithms like ECDSA, it still remains relevant in legacy systems and applications where compatibility with older standards is required. Additionally, DSA has contributed to the evolution of digital signature algorithms and has helped pave the way for the development of more secure and efficient solutions.

In summary, the Digital Signature Algorithm (DSA) played a significant role in the early development of standardized digital signature schemes, providing a foundation for secure authentication and data integrity in digital communications. Its history is intertwined with the evolution of cryptographic standards and the ongoing quest for stronger security mechanisms in the digital age.

2.2.1 Algorithm

1. Key Generation:
• Choose a prime number, denoted as "p," which is typically 512 to 1024 bits in length. This prime serves as a modulus for various calculations.
• Select a smaller prime, denoted as "q," which is typically 160 bits in length. This prime is used to divide the group order.
• Find an integer "g" that is a generator of a subgroup of order "q" modulo "p." This subgroup is used for certain calculations in the algorithm.
• Choose a private key "x" as a random integer within the range [1, q-1].
• Compute the public key "y" as $y = g^x \mod p$.

2.Signature Generation:
• Calculate the message digest (hash value) of the message you want to sign using a cryptographic hash function (e.g., SHA-1 or SHA-256).
• Choose a random value "k" within the range [1, q-1].
• Compute $r = (g^k \mod p) \mod q$.
• Calculate the modular multiplicative inverse of "k" modulo "q," denoted as "k^{-1}."
• Compute $s = (k^{-1} \cdot (H(m) + xr)) \mod q$, where "H(m)" is the hash value of the message.
• The signature is the pair (r, s).

3.Signature Verification:
• Receive the message, the signature (r, s), and the sender's public key (p, q, g, y).
• Verify that "r" and "s" are within the range [1, q-1].
• Compute the message digest "H(m)" of the received message.
• Calculate the modular multiplicative inverse of "s" modulo "q," denoted as "s^{-1}."
• Compute $w = s^-1 \mod q$.
• Calculate two values: $u_1 = (H(m) \cdot w) \mod q$ and $u_2 = (r \cdot w) \mod q$.
• Compute $v = ((g^{u_1} \cdot y^{u_2}) \mod p) \mod q$.
• If "v" matches the value of "r," the signature is valid; otherwise, it is not.

It's important to follow these steps precisely to ensure the security and correctness of the DSA algorithm. Additionally, care should be taken to generate strong random values for "k" during signature generation to prevent vulnerabilities.
DSA is widely used for digital signatures, but it's worth noting that newer algorithms like ECDSA (Elliptic Curve Digital Signature Algorithm) are gaining popularity due to their shorter key lengths and comparable security. Always consult the latest cryptographic guidelines and best practices when implementing digital signatures in your applications.

Example 2.2.1

Key Generation
Given:

$$p = 23$$
$$q = 11$$
$$g = 6$$
$$x = 7 \quad \text{(Private Key)}$$

Compute public key y:

$$y = (g^x \mod p)$$
$$y = (6^7 \mod 23)$$
$$y = 15$$

Signature Generation

Given:

$$\text{Message:} \quad \text{"Hello"}$$
$$H(m) = \text{SHA-256("Hello")} = 0x23$$
$$k = 3 \quad \text{(Random)}$$

Compute:

$$r = (g^k \mod p) \mod q$$
$$r = (6^3 \mod 23) \mod 11$$
$$r = 8$$

Calculate modular inverse of k modulo q:

$$k^{-1} \mod q = 4$$

Compute:

$$s = (k^{-1} \cdot (H(m) + xr)) \mod q$$
$$s = (4 \cdot (0x23 + 7 \cdot 8)) \mod 11$$
$$s = 6$$

Signature: $(r, s) = (8, 6)$

Signature Verification

Given:

$$\text{Received Message:} \quad \text{"Hello"}$$
$$\text{Received Signature:} \quad (r, s) = (8, 6)$$

Calculate hash of received message:

$$H(m) = \text{SHA-256("Hello")} = 0x23$$

Calculate $w = s^{-1} \mod q$:

$$w = 6^{-1} \mod 11$$
$$w = 2$$

Calculate $u_1 = (H(m) \cdot w) \mod q$:

$$u_1 = (0x23 \cdot 2) \mod 11$$
$$u_1 = 1$$

Calculate $u_2 = (r \cdot w) \mod q$:

$$u_2 = (8 \cdot 2) \mod 11$$
$$u_2 = 5$$

Calculate $v = ((g^{u_1} \cdot y^{u_2}) \mod p) \mod q$:

$$v = ((6^1 \cdot 15^5) \mod 23) \mod 11$$
$$v = 8$$

Since v matches the received r, the signature is valid.

Chapter 3

Hashing

3.1 What is Hashing?

Definition 3.1.1: Hashing

Hashing refers to the process of taking input data of any size and transforming it into a fixed-size string of characters, typically a sequence of numbers and letters. This output is known as a hash value or hash code. Hashing is commonly used in computer science and cryptography for various purposes, such as data retrieval, data storage, password security, and digital signatures.

Definition 3.1.2: Hash Function

A hash function is a mathematical algorithm that takes an input (or "message") and produces a fixed-size string of characters, which is typically a sequence of numbers and letters. This output is known as the hash value, hash code, or simply hash. Hash functions are designed to efficiently transform input data of arbitrary size into a fixed-size value, which is usually of a much smaller size.

Key characteristics of a good hash function include:

Deterministic:
For a given input, a hash function always produces the same hash value. This property ensures consistency and reliability.

Fixed Size:
The hash value has a predetermined length, regardless of the size of the input data. This allows for uniform representation of data.

Fast Computation:
Hash functions are designed to be computationally efficient, making them suitable for processing large volumes of data quickly.

Preimage Resistance:
It should be computationally infeasible to reverse-engineer the original input from its hash value. This property helps protect the integrity of the original data.

Collision Resistance:
It should be highly improbable for two different inputs to produce the same hash value. This property is crucial to avoid unintended clashes between different inputs.

Avalanche Effect:
A small change in the input data should result in a significantly different hash value. This ensures that even minor modifications produce vastly different hash codes.

Hash functions have various applications in computer science and cryptography, including:

Data Integrity:
Hashing is used to verify the integrity of data during transmission or storage. By comparing the hash of received data with the original hash, you can determine if the data has been altered.

Digital Signatures:
Hash functions are used in digital signatures to ensure the authenticity and integrity of electronic documents or messages.

Password Security:
Hashing is commonly used to securely store passwords. Instead of storing actual passwords, systems store their hash values, making it difficult for attackers to retrieve the original passwords.

Hash Tables:
Hash functions are fundamental in data structures like hash tables, used for efficient data retrieval and storage.

Cryptographic Applications:
Hash functions play a critical role in various cryptographic protocols, such as message authentication codes (MACs), key derivation functions (KDFs), and more.

Blockchain and Cryptocurrencies:
Hash functions are used extensively in blockchain technology for creating blocks, ensuring consensus, and securing transactions.

Examples of commonly used hash functions include MD5, SHA-1, SHA-256, and SHA-3. It's important to note that not all hash functions are equally secure, and some older hash functions like MD5 and SHA-1 are considered vulnerable to attacks. Modern applications should use stronger and more secure hash functions to ensure data integrity and security.

3.2 MD5

MD5 (Message Digest Algorithm 5) is a widely used cryptographic hash function that was developed by Ronald Rivest in 1991. It's designed to take an input message of any length and produce a fixed-size (128-bit) hash value. MD5 was initially developed for use in digital signatures and message integrity checks, but over time, its security vulnerabilities have become more pronounced, and it is no longer considered secure for many applications.

Here's an overview of how MD5 works and its key characteristics:

Hashing Process:
MD5 operates by processing the input message in blocks of data and iteratively transforming the data using a series of logical operations, including bitwise logical operations (AND, OR, XOR), rotations, and modular addition. The process involves four rounds of operations for each block of data.

Fixed-Size Output:
Regardless of the size of the input message, MD5 always produces a 128-bit (16-byte) hash value. This fixed output size makes MD5 suitable for applications that require a consistent-length hash code.

Deterministic and Fast:
Like other hash functions, MD5 is deterministic, meaning the same input will always produce the same hash output. It's also designed to be computationally efficient, allowing for fast processing of data.

Security Concerns:
MD5's security weaknesses have been highlighted over the years. Researchers have demonstrated various vulnerabilities, including collision attacks, where different inputs produce the same hash value. These vulnerabilities have significant implications for the security of systems that rely on MD5 for data integrity or authentication.

Collision Attacks:
The most significant concern with MD5 is its vulnerability to collision attacks, where attackers can deliberately create two different messages that produce the same MD5 hash value. This undermines the integrity of hash-based security applications.

Cryptanalysis and Deprecated Status:
As a result of these vulnerabilities, MD5 is considered cryptographically broken and deprecated for most security-sensitive applications. It is no longer recommended for tasks such as secure digital signatures, password hashing, or data integrity verification.

Usage Today:
Despite its weaknesses, MD5 is still occasionally used for non-cryptographic purposes, such as checksums for file integrity verification, where security is not the primary concern.

3.2.1 algorithm

Padding:
MD5 processes input messages in blocks of 512 bits (64 bytes). If the message's length is not a multiple of 64 bytes, it needs to be padded to a multiple of this size. The padding includes adding a '1' bit followed by enough '0' bits to reach 64 bits less than the next multiple of 512. Then, the length of the original message (in bits) is appended as a 64-bit integer.

Initialize Hash Values:
MD5 uses four 32-bit words (A, B, C, D) as its internal state. These are initialized to fixed constants, which serve as initial chaining values.

Main Loop:
The padded message is divided into 512-bit blocks, and the MD5 algorithm processes each block sequentially. Each block goes through four rounds of processing.

• Round 1: In this round, a series of bitwise logical operations (AND, OR, XOR), rotations, and modular addition are applied to the data and the current hash state.

• Round 2: Similar operations are applied with different functions and constants.

• Round 3: More operations are applied based on different functions and constants.

• Round 4: The operations are applied once again with different functions and constants.

Final Output:
After processing all blocks, the resulting hash values (A, B, C, D) are concatenated to produce the final MD5 hash value. The order of the bytes in each word is important, and the hash value is typically represented as a 128-bit (16-byte) hexadecimal number.

Example 3.2.1

Step 1: **Message Preparation and Padding**:

The ASCII representation of the message "Hello, MD5!" is: 72 101 108 108 111 44 32 77 68 53 33

To make the message a multiple of 512 bits (64 bytes), we need to pad it. The padding consists of a '1' bit followed by '0' bits, and then the length of the original message (in bits) represented as a 64-bit integer. The total length of the padded message will be a multiple of 512 bits.

Padded Message: 72 101 108 108 111 44 32 77 68 53 33 128 0 0 ... (additional zeroes to reach the block size)

Step 2: **Initialize Hash Values**:

The initial hash values (A, B, C, and D) are set:

- $A = 0x67452301$
- $B = 0xEFCDAB89$
- $C = 0x98BADCFE$
- $D = 0x10325476$

Step 3: **Main Loop**:

In this detailed example, we'll focus on one round (Round 1) of processing a single 512-bit block:

- Round 1:
  - Function $F(B, C, D) = (B\&C)|(\neg B\&D)$
  - Let's assume the current block's data is the padded message we calculated earlier.
  - Temp = D = 0x10325476
  - D = C = 0x98BADCFE
  - C = B = 0xEFCDAB89
  - Let's calculate $F(B, C, D)$:
  - $F(B, C, D) = (0xEFCDAB89\&0x98BADCFE)|(\neg 0xEFCDAB89\&0x10325476) = 0x98BC9D7E$
  - Let's assume the constant value for this round: Constant = 0x5A827999
  - Let's assume the shift amount for this round: ShiftAmount = 7
  - Calculate the new value of B:
  - $B = B + \text{LeftRotate}((A + F(B, C, D) + \text{Data} + \text{Constant}), \text{ShiftAmount})$
  - $B = 0xEFCDAB89 + \text{LeftRotate}((0x67452301 + 0x98BC9D7E + \text{Data} + 0x5A827999), 7)$
  - $B = 0xEFCDAB89 + 0x03B9AC4C$
  - $B = 0xF2AF9A75$

Step 4: **Final Output**:

After processing the entire block, the resulting values of A, B, C, and D are combined to produce the final MD5 hash value.

Assuming the final values are:

- A = ...
- B = 0xF2AF9A75
- C = ...
- D = ...

The hash value is represented as a 128-bit hexadecimal number: **MD5("Hello, MD5!")** = 7a4c7b877b7d8608e1e6daa64ffef61d

3.2.2 Weakness

Collision Vulnerabilities:
MD5 suffers from significant collision vulnerabilities. A collision occurs when two different inputs produce the same hash value. Researchers have demonstrated practical collision attacks, allowing them to find different inputs that result in the same MD5 hash. This undermines the integrity and authenticity assurances that hash functions are meant to provide.

Fast Computation:
MD5 is designed to be fast, which makes it susceptible to brute-force attacks. Modern hardware and computing power have made it feasible to calculate the MD5 hash of a large number of possible inputs in a short amount of time, greatly reducing the effort required for attackers to find collisions or reverse-engineer hashed passwords.

Lack of Resistance to Cryptanalysis:
Cryptanalysis is the study of breaking cryptographic systems. MD5's design lacks certain properties that are necessary for a secure cryptographic hash function. Researchers have found weaknesses in MD5's internal operations that can be exploited using advanced mathematical techniques.

Dependence on Merkle-Damgård Construction:
MD5 uses the Merkle-Damgård construction, which involves dividing the input message into blocks and processing them sequentially. While this construction is widely used in hash functions, MD5's weaknesses are exacerbated by the specific way it handles message blocks and updates its internal state.

Multiple Attack Vectors:
MD5's vulnerabilities have been exploited in various ways, including collision attacks, chosen-prefix collisions, and more. These attacks have been used to create malicious certificates, break digital signatures, and compromise authentication systems.

Widespread Use of MD5:
The widespread use of MD5 in various applications (such as SSL certificates, password storage, and digital signatures) means that if an attacker can generate collisions or find pre-images (input messages for a given hash value), they can exploit these weaknesses across multiple systems.

3.3 SHA-256

SHA-256 (Secure Hash Algorithm 256-bit) is a widely used cryptographic hash function that belongs to the SHA-2 family of hash functions. It is designed to take an input message and produce a fixed-size output hash value of 256 bits (32 bytes). SHA-256 is considered to be much more secure and robust than its predecessor, MD5, and is widely used for a variety of cryptographic applications.

Here's an overview of SHA-256 and its key features:
Hash Function Properties:
• One-way: It is computationally infeasible to reverse the hash to obtain the original input.
• Deterministic: The same input will always produce the same hash output.
• Fixed Output Size: SHA-256 always produces a 256-bit output, regardless of the input size.
• Avalanche Effect: A small change in the input results in a significantly different output.

Secure Design:
SHA-256 is designed to be resistant to various cryptographic attacks, including collision attacks (finding two different inputs that produce the same hash) and pre-image attacks (finding an input that matches a given hash).

Usage:
SHA-256 is used in a wide range of cryptographic applications, including:
• Digital Signatures: To sign and verify digital documents, ensuring their authenticity and integrity.
• Message Authentication Codes (MACs): To verify the integrity of messages and detect any tampering.
• Password Storage: To securely store passwords by hashing them before storage, protecting against unauthorized access even if the hash is compromised.
• Certificate Authorities: For generating and verifying SSL/TLS certificates.

SHA-256 Algorithm:
The SHA-256 algorithm processes the input message in blocks of 512 bits (64 bytes) and uses a series of logical and arithmetic operations, bitwise operations (AND, OR, XOR), modular arithmetic, and rotations to transform the input into the final hash value.

Security:
As of my knowledge cutoff date in September 2021, SHA-256 is considered secure and suitable for most cryptographic applications. However, it's important to note that the security of hash functions can degrade over time as computational power increases and new attacks are developed. Therefore, it's recommended to monitor developments in cryptographic research and use the latest recommended hash functions.

Variants:
The SHA-2 family includes other hash functions with different output sizes, such as SHA-224, SHA-256, SHA-384, SHA-512, SHA-512/224, and SHA-512/256. The numbers in the variant names indicate the output size in bits.

In summary, SHA-256 is a widely used cryptographic hash function that provides strong security guarantees for various applications requiring data integrity, authentication, and non-repudiation. It has replaced less secure

hash functions like MD5 and SHA-1 in many contexts.

3.3.1 algorithm

1. Padding and Message Length:
• The input message is represented as a sequence of bits. It may have any length, and the goal is to make it a multiple of 512 bits (64 bytes) to facilitate processing.
• A single '1' bit is appended to the end of the message.
• '0' bits are then added until the length of the message is 448 bits (56 bytes) less than a multiple of 512.
• Finally, the original length of the message (in bits) is added as a 64-bit binary representation at the end of the padded message.

2. Message Block Processing:
• The padded message is divided into chunks called "message blocks," each 512 bits in length (64 bytes).
• For example, if the padded message is 1280 bits long, it will be divided into three 512-bit blocks and one 256-bit block.

3. Message Schedule Expansion:
• For each message block, the initial 16 words (32 bits each) are derived directly from the block's 512 bits. These words are denoted as W0 to W15.
• The remaining 48 words (W16 to W63) are calculated using a formula that involves bitwise operations (e.g., XOR, AND, OR), rotations, and modular additions.

4. Initial Hash Values (H0 - H7):
• SHA-256 uses eight initial hash values, denoted as H0 through H7. These values are constants derived from the fractional parts of the square roots of the first eight prime numbers.

Note:-

$H0$ = `0x6a09e667`
$H1$ = `0xbb67ae85`
$H2$ = `0x3c6ef372`
$H3$ = `0xa54ff53a`
$H4$ = `0x510e527f`
$H5$ = `0x9b05688c`
$H6$ = `0x1f83d9ab`
$H7$ = `0x5be0cd19`

5. Main Compression Loop:
• The compression loop consists of 64 rounds (iterations), each with a dedicated set of operations.
• The eight working variables (a, b, c, d, e, f, g, h) are used to hold intermediate hash values during the computation.

6. Round Functions:
Each round involves four logical functions:
• Ch (Choose): This function chooses bits from "e" and "f," using "g" as a selector. It simulates a conditional bit selection based on "e" and "f."
• Maj (Majority): This function calculates the majority of bits from "a," "b," and "c." It simulates a majority vote on each bit position.
• Σ0 (Sigma 0): This function applies bitwise rotations and XOR operations to "a" to produce a new value.
• Σ1 (Sigma 1): This function applies similar operations to "e."

7. Final Hash Value:
• After all 64 rounds, the updated values of a, b, c, d, e, f, g, and h are combined with the initial hash values

(H0 to H7) to produce the final 256-bit hash value.
• This final hash value represents the cryptographic hash of the input message.

The combination of these steps and the intricate use of bitwise operations, modular arithmetic, and logical functions ensure that the SHA-256 algorithm produces a secure and irreversible hash value. This value serves as a unique representation of the input data and is crucial for various cryptographic applications, including data integrity verification, digital signatures, and password hashing.

Example 3.3.1

Step 1: **Message Preparation and Padding**:

The ASCII representation of the message "Hello, SHA-256!" is:
`72 101 108 108 111 44 32 83 72 65 45 50 53 54 33`

To make the message a multiple of 512 bits (64 bytes), we need to pad it. The padding consists of a '1' bit followed by '0' bits, and then the length of the original message (in bits) represented as a 64-bit binary number.

The binary representation of the message length is:
`0000000000000000000000000000000000000000000000000000000001101100000`

The padded message is then constructed with a '1' bit followed by zeroes, and the message length in binary: `72 101 108 108 111 44 32 83 72 65 45 50 53 54 33 10000000`
`0000000000000000000000000000000000000000000000000000000001101100000`

Step 2: **Initialize Hash Values**:

The initial hash values (H0 to H7) are set as hexadecimal constants:

$$H0 = \texttt{0x6a09e667}$$
$$H1 = \texttt{0xbb67ae85}$$
$$H2 = \texttt{0x3c6ef372}$$
$$H3 = \texttt{0xa54ff53a}$$
$$H4 = \texttt{0x510e527f}$$
$$H5 = \texttt{0x9b05688c}$$
$$H6 = \texttt{0x1f83d9ab}$$
$$H7 = \texttt{0x5be0cd19}$$

Step 3: **Main Loop**:

The padded message is divided into 512-bit blocks. In this example, we have one block. Each block is divided into 16 32-bit words: W_0 through W_{15}.

Step 4: **Round Processing**:

For each round (0 to 63), the compression function is applied to update the hash values. The compression function involves various bitwise and arithmetic operations that update the working variables a, b, c, and d.

Step 5: **Finalization**:

After processing the entire block, the final hash value is derived from the updated hash values (H0 to H7) by concatenating them.

Step 6: **Output**:

The final SHA-256 hash value for the message "Hello, SHA-256!" is:
`5b8c39b8a6f4db4a82bbf515eefb21798d0f59c1990b7bb9e9c5c3e4b8e04785`

Please note that this example provides a detailed breakdown of the SHA-256 algorithm for educational purposes. The actual algorithm involves intricate bitwise and arithmetic operations, as well as complex mathematical constants derived from prime numbers. The security of SHA-256 relies on the complexity of these operations and the difficulty of finding two different inputs that produce the same hash value (collision resistance).
For a real implementation and usage of SHA-256, cryptographic libraries or tools should be used. You can find SHA-256 implementations in programming languages like Python, Java, and others.