Symmetric and Asymmetric Algorithms

191180083 - Abdullah Berkay Uymaz

2024

1 Introduction

In the realm of cryptography, encryption algorithms are crucial for securing data. These algorithms can be broadly categorized into two types: symmetric and asymmetric algorithms. Each type has its unique mechanisms, advantages, and use cases. This document will explore the workings of both symmetric and asymmetric algorithms, providing detailed explanations and sample code to illustrate their functionalities.

2 Symmetric Algorithm

Symmetric algorithms, also known as secret-key algorithms, use the same key for both encryption and decryption. This key must be shared between the sender and the receiver and kept secret from unauthorized parties. Symmetric algorithms are generally faster than asymmetric algorithms and are suitable for encrypting large amounts of data. One of the most widely used symmetric algorithms is the Advanced Encryption Standard (AES).

2.1 How AES Works

AES (Advanced Encryption Standard) is a symmetric encryption algorithm standardized by NIST (National Institute of Standards and Technology) in 2001. It encrypts data in fixed-size blocks of 128 bits using keys of 128, 192, or 256 bits. The AES encryption process involves several rounds of transformation, each consisting of substitution, permutation, mixing, and key addition operations. The number of rounds depends on the key length: 10 rounds for 128-bit keys, 12 rounds for 192-bit keys, and 14 rounds for 256-bit keys.

- **SubBytes**: A non-linear substitution step where each byte is replaced with another according to a lookup table.
- **ShiftRows**: A transposition step where the rows of the state are shifted cyclically.
- **MixColumns**: A mixing operation which operates on the columns of the state, combining the four bytes in each column.
- **AddRoundKey**: Each byte of the state is combined with a byte of the round key using bitwise XOR.

2.2 AES Sample Code

Here is an example of how AES encryption works in Python using the `pycryptodome` library:

```python
from Crypto.Cipher import AES
from Crypto.Random import get_random_bytes

# Generate a random key
key = get_random_bytes(16)  # 128-bit key

# Create a cipher object
cipher = AES.new(key, AES.MODE_EAX)

# Encrypt some data
data = b'This is a secret message'
nonce = cipher.nonce
ciphertext, tag = cipher.encrypt_and_digest(data)

print(f'Ciphertext: {ciphertext.hex()}')
```

Listing 1: AES Encryption Example

In this example, the AES algorithm encrypts a simple message using a randomly generated key. The nonce (number used once) is essential for ensuring the uniqueness of the encryption and must be shared with the decryption party along with the ciphertext.

3 Asymmetric Algorithm

Asymmetric algorithms, also known as public-key algorithms, use a pair of keys: a public key for encryption and a private key for decryption. The public key can be shared openly, while the private key must be kept secure. This method provides a higher level of security and is often used for securing sensitive data, digital signatures, and key exchanges. One of the most common asymmetric algorithms is the RSA (Rivest-Shamir-Adleman) algorithm.

3.1 How RSA Works

RSA is an asymmetric encryption algorithm based on the mathematical difficulty of factoring the product of two large prime numbers. The RSA algorithm involves the following steps:

1. **Key Generation**: Two large prime numbers are chosen, and their product, n, forms the modulus for both the public and private keys. The public key consists of n and an exponent e, while the private key consists of n and a different exponent d.

2. **Encryption**: A plaintext message is converted into an integer m and encrypted using the public key with the formula $c = m^e \mod n$, where c is the ciphertext.

3. **Decryption**: The ciphertext is decrypted using the private key with the formula $m = c^d \mod n$ to retrieve the original message.

3.2 RSA Sample Code

Here is an example of how RSA encryption works in Python using the `pycryptodome` library:

```python
from Crypto.PublicKey import RSA
from Crypto.Cipher import PKCS1_OAEP
from Crypto.Random import get_random_bytes

# Generate RSA keys
key = RSA.generate(2048)
public_key = key.publickey()

# Encrypt some data
cipher = PKCS1_OAEP.new(public_key)
data = b'This is a secret message'
ciphertext = cipher.encrypt(data)

print(f'Ciphertext: {ciphertext.hex()}')
```

Listing 2: RSA Encryption Example

In this example, the RSA algorithm encrypts a simple message using a generated public key. The private key, which is kept secure, is required to decrypt the message. The `PKCS1_OAEP` padding scheme is used to provide additional security.

4 Comparison of Symmetric and Asymmetric Algorithms

4.1 Performance

Symmetric algorithms are generally faster than asymmetric algorithms due to their simpler mathematical operations. This makes them suitable for encrypting large amounts of data, such as securing communication channels or storage devices. Asymmetric algorithms, while more secure, are computationally intensive and slower. They are often used for key exchange, digital signatures, and scenarios where data needs to be securely transmitted without pre-sharing a secret key.

4.2 Security

Asymmetric algorithms provide a higher level of security due to the use of two separate keys. Even if the public key is exposed, the private key remains secure, preventing unauthorized decryption. Symmetric algorithms rely on the secrecy of the shared key, which must be securely exchanged and stored. If the key is compromised, the encrypted data can be easily decrypted.

4.3 Use Cases

Symmetric algorithms are commonly used for encrypting data at rest, securing communication channels (e.g., SSL/TLS), and in applications requiring high performance. Asymmetric algorithms are used for digital signatures, key exchange protocols (e.g., Diffie-Hellman), and securing data in transit, especially in scenarios where secure key distribution is challenging.

5 Conclusion

Both symmetric and asymmetric algorithms play vital roles in modern cryptography, each with its strengths and weaknesses. Symmetric algorithms, with their speed and efficiency, are ideal for encrypting large amounts of data, while asymmetric algorithms provide robust security for key exchange and digital signatures. Understanding the principles and applications of these algorithms is crucial for designing secure systems and protecting sensitive information.