**Symmetric Key Cryptosystem**

In the Symmetric Key Encryption algorithm, the dK is either the same as eK or can be easily derived from eK.

**Algorithms**

1. **DES**
   * Data Encryption Standard is a ***16 round Feistel Cipher***
   * It is a block cipher i.e, a block of data is encrypted and decrypted at a time.
   * Each block of message (plaintext) is encrypted using a secret key using a substitution box & a permutation box.
   * Decryption is the inverse of encryption.

***pyDes*** is a python package for the implementation of DES.

* *key*: Contains the encryption key
* *pad*: Used to pad the inputted string so that is a multiple of 8 for encryption & decryption in DES, the mode is set as ***None*** assuming the inputted string is a multiple of 8
* *padmode*: Ensures the inputted string is of the required block size for encryption & decryption
* encrypt(): predefined encryption function
* decrypt(): predefined decryption function

des = pyDes.des(key, pyDes.ECB, pad=None, padmode=pyDes.PAD\_PKCS5)

cipher = des.encrypt(plaintext)

plain = des.decrypt(ciphertext)

1. **AES**
   * Advanced Encryption Standard is an iterative block cipher.
   * Block Size: 128 bits
   * Rounds: 10, 12, 14
   * Key Length: 128, 192, 256
   * Main Components
     1. *SUBBYTES():* applies a substitution table (S-box) to each byte of the block.
     2. *SHIFTROWS():* bytes in the last three rows of the block matrix are cyclically shifted
     3. *MIXCOLUMNS():* each of the four columns of the block matrix are multiplied by a fixed matrix
     4. *ADDROUNDKEY():* round key is applied using XOR operation on the block.

***cryptography*** is a python package for the implementation of AES.

1. Line 1: creation of cipher object to perform encryption & decryption

* ***modes.CBC*** during encryption is CBC where an Initial Value (iv) is defined & used during encryption & decryption.
* ***algorithms.AES(key)*** defines implementation of AES encryption & decryption

1. Line 2: ***encryptor():*** used for performing encryption on the inputted string
2. Line 3: ***padding.PKCS7(128).padder():*** used to pad the inputted string to make it a 128 bit block for encryption & decryption
3. Line 4: ***padder.update(plaintext) + padder.finalize():*** will make the plaintext of 128 bits
4. Line 5: ***encryptor.update(padded\_plaintext) + encryptor.finalize():*** encrypts the padded plaintext with the key & iv.
5. Line 6: ***decryptor.update(ciphertext) + decryptor.finalize():*** performs decryption on the ciphertext inputted by the user
6. Line 7: ***padding.PKCS7(128).unpadder():*** used to remove the padding done for encryption & decryption as it can be only performed on 128 bit block
7. Line 8: ***unpadder.update(padded\_plaintext) + unpadder.finalize():*** to remove the padding from the padding from ciphertext after decryption & to store in variable named *plaintext*.

Line 1: cipher = Cipher(algorithms.AES(key), modes.CBC(iv))

Line 2: encryptor = cipher.encryptor()

Line 3: padder = padding.PKCS7(128).padder()

Line 4: padded\_plaintext = padder.update(plaintext) + padder.finalize()

Line 5: ciphertext = encryptor.update(padded\_plaintext) + encryptor.finalize()

Line 6: padded\_plaintext = decryptor.update(ciphertext) + decryptor.finalize()

Line 7: unpadder = padding.PKCS7(128).unpadder()

Line 8: plaintext = unpadder.update(padded\_plaintext) + unpadder.finalize()

**Drawback of Symmetric Key Cryptosystem**

* The key K must be communicated using a secure channel before any text (ciphertext) is communicated.
* Exposure of eK or dK compromises the system.

**Asymmetric Key Cryptosystem**

In the Asymmetric Key Encryption algorithm, it is difficult to compute dK from the given eK.

* The encryption key, eK is a public key which is put in the directory. The text can be send without having to send the public key through a secure channel.
* The receiver is the only person who can decrypt the ciphertext using the dK which is a private key.

**Algorithms**

1. **Diffie-Hellman**

The Diffie-Hellman algorithm is being used to establish a shared secret that can be used for secret communications

* + **mod\_exp():** will generate modular exponentiation of a prime number.

**def mod\_exp(base, exponent, modulus):**

**result = 1**

**while exponent > 0:**

**if exponent % 2 == 1:**

**result = (result \* base) % modulus**

**exponent //= 2**

**base = pow(base, 2, modulus)**

**return result**

1. **RSA**

* In RSA, the public and private key are a product of 2 large prime numbers.
* The security of the message is dependent on the size of the key. Larger the key size the more difficult it is to factorize it.
* Key Size: 1024 bits, 2048 bits
* ***gcd(a, b):*** Using Euclidean Theorem will calculate the gcd of 2 nos. namely a,b
* ***is\_prime(num, k=10):*** using the *sympy* packet that contains primality test will check whether the given number is prime or not
* ***generate\_prime(bits):*** to generate random prime numbers of a particular bit size
* ***generate\_keypair(bits):*** generates public & private key for encryption and decryption using Euler’s Totient function
* ***encrypt(message, public\_key):*** will encrypt the message with the public key by converting each character to its ASCII value & then find its exponent and modulus value.
* ***decrypt(encrypted\_message, private\_key):*** the ciphertext will be decrypted using the receivers private key after decryption it will convert the ASCII values to characters

def encrypt(message, public\_key):

e, n = public\_key

cipher = []

for char in message:

encrypted\_char = pow(ord(char), e, n)

cipher.append(encrypted\_char)

return cipher

def decrypt(encrypted\_message, private\_key):

d, n = private\_key

plain = []

for char in encrypted\_message:

decrypted\_char = chr(pow(char, d, n))

plain.append(decrypted\_char)

return ''.join(plain)

**Hashing**

A cryptographic hash function provides data integrity to a message. It produces a message digest, if the data is altered then the message digest is not valid.

Python contains a predefined hash() function that returns the hash value of the inputted string.

Let ***hash()*** be a hash function & let ***x*** be the string of messages of arbitrary length. Then the message digest is defined as

***md=hash(x)***

def hashing(o):

return hash(o)

a = str(input("Enter a String: "))

md = hashing(a) #message digest

print("Hash Value:", md)