Lec3_Cryptography II

Block ciphers

Definition

- A function that maps n-bit plaintext blocks to n-bit ciphertext blocks
  - n: block length - DES: 64bits, AES: 128bits
- parameterized by k-bit key K(random), k is keylength
- **Encryption** C = E(K,M)
- **Decryption** M = D(K,C)
- K is k-bits long, number of keys is 2^k

Encryption function is **bijection** - one-to-one and onto

E(k) be bijection as be reserved decryption

Security requirement

block cipher be secure: **pseudo random permutation(PRP)**

- key is secret, attacker shouldn't be able to discover any pattern in the input/output value of block cipher

truly random permutation

Computationally safe: 2^112 bits

DES/AES

*Feistel cipher: depend on before

56-bit key

Properties of block cipher

confusion: input undergoes complex transformation - depth

diffusion: transformation depend equaly on all bits of the input - breadth

substitution–permutation networks

Avalanche effect: Slight change in input significantly changes the output

- block cipher
- cryptographic hash functions

Modes of operation

block cipher: length of data > block size

- how to employ for large message
  - divide msg & padding last block
- Initialization Vector (IV):
  - block of data used in addition to input message
  - randomize encryption process

ECB mode

Ci = EK(Pi)

Pi=DK(Ci)

- **simple encryption** mode: encrypts each block of plaintext independently use same key
- Identical plaintext (with same key) result in identical ciphertext
- Bit errors: only affect that block

Weakness

- does not hide data pattern
- malicious substitution of cipher text is possible
- cannot use

- multi-block messages, keys are reused for more than a single block

```
from Crypto.Cipher import AES
from Crypto.Util.Padding import pad, unpad

def encrypt_ecb(key, plaintext):
    cipher = AES.new(key, AES.MODE_ECB)
    padded_plaintext = pad(plaintext, AES.block_size)
    ciphertext = cipher.encrypt(padded_plaintext)
    return ciphertext

def decrypt_ecb(key, ciphertext):
    cipher = AES.new(key, AES.MODE_ECB)
    decrypted_data = cipher.decrypt(ciphertext)
    plaintext = unpad(decrypted_data, AES.block_size)
    return plaintext

# Example usage
key = b'thisisa16bytekey'
plaintext = b'This is the plaintext message'

# Encryption
encrypted_data = encrypt_ecb(key, plaintext)
print("Encrypted data:", encrypted_data)

# Decryption
decrypted_data = decrypt_ecb(key, encrypted_data)
print("Decrypted data:", decrypted_data.decode())
```

CBC mode

C0 = IV

$Ci = \mathrm{E}K(Pi) \oplus Ci - 1$

$Pi = \mathrm{D}K(Ci) \oplus Ci - 1$

ciphertext C_i to depend on all preceding plaintext

Properties

- IV must be integrity-protected, otherwise predictable bit change in 1st block
- same key, IV, plaintext result in identical cipher text
- A single bit error in C_i affects decryption of blocks C_i and C_{i+1}
- self-synchronizing
  - error in C_i but not in C_{i+1}, C_{i+2}
  - P_{i+2} is correctly decrypted

```
from cryptography.hazmat.primitives.ciphers import Cipher, algo
from cryptography.hazmat.backends import default_backend
from cryptography.hazmat.primitives import padding
from cryptography.hazmat.primitives import hashes

def pad_data(data):
    padder = padding.PKCS7(128).padder()
    padded_data = padder.update(data)
    padded_data += padder.finalize()
    return padded_data

def unpad_data(padded_data):
    unpadder = padding.PKCS7(128).unpadder()
    data = unpadder.update(padded_data)
    data += unpadder.finalize()
    return data

def encrypt_CBC(plaintext, key, iv):
    backend = default_backend()
    cipher = Cipher(algorithms.AES(key), modes.CBC(iv), backend=
    encryptor = cipher.encryptor()
    encrypted_data = encryptor.update(pad_data(plaintext)) + enc
    return encrypted_data
```

```
def decrypt_CBC(ciphertext, key, iv):
    backend = default_backend()
    cipher = Cipher(algorithms.AES(key), modes.CBC(iv), backend=
    decryptor = cipher.decryptor()
    decrypted_data = decryptor.update(ciphertext) + decryptor.fi
    return unpad_data(decrypted_data)

# Example usage
key = b'0123456789abcdef'  # 128-bit key
iv = b'abcdefghijklmnop'  # 128-bit IV

plaintext = b'This is the plaintext message.'

# Encrypt the plaintext
ciphertext = encrypt_CBC(plaintext, key, iv)
print("Ciphertext:", ciphertext.hex())

# Decrypt the ciphertext
decrypted_text = decrypt_CBC(ciphertext, key, iv)
print("Decrypted text:", decrypted_text.decode())
```

CTR (counter) mode

C0 = IV $\oplus$ counter

$C_i = E_K(P_i) \oplus C_{i-1}$

$P_i = D_K(C_i) \oplus C_{i-1}$

- **IV concatenated / XOR with counter**
- Counter must be different for each block

Properties

- **Software and hardware efficiency** - blocks enctypted in parallel

- **Preprocessing** - encryption part can be done offline, when message is know, do XOR
- **Random access** - decryption of block done in random order
- useful for hard-disk encryption

```
from cryptography.hazmat.primitives.ciphers import Cipher, algo
from cryptography.hazmat.backends import default_backend
from cryptography.hazmat.primitives import padding
from cryptography.hazmat.primitives import hashes
from cryptography.hazmat.primitives import counter

def pad_data(data):
    padder = padding.PKCS7(128).padder()
    padded_data = padder.update(data)
    padded_data += padder.finalize()
    return padded_data

def unpad_data(padded_data):
    unpadder = padding.PKCS7(128).unpadder()
    data = unpadder.update(padded_data)
    data += unpadder.finalize()
    return data

def encrypt_CTR(plaintext, key, nonce):
    backend = default_backend()
    ctr = counter.Counter(nonce)
    cipher = Cipher(algorithms.AES(key), modes.CTR(ctr), backen
    encryptor = cipher.encryptor()
    encrypted_data = encryptor.update(pad_data(plaintext)) + en
    return encrypted_data

def decrypt_CTR(ciphertext, key, nonce):
    backend = default_backend()
    ctr = counter.Counter(nonce)
    cipher = Cipher(algorithms.AES(key), modes.CTR(ctr), backen
```

```
    decryptor = cipher.decryptor()
    decrypted_data = decryptor.update(ciphertext) + decryptor.fi
    return unpad_data(decrypted_data)

# Example usage
key = b'0123456789abcdef'  # 128-bit key
nonce = b'1234567890abcdef'  # 128-bit nonce

plaintext = b'This is the plaintext message.'

# Encrypt the plaintext
ciphertext = encrypt_CTR(plaintext, key, nonce)
print("Ciphertext:", ciphertext.hex())

# Decrypt the ciphertext
decrypted_text = decrypt_CTR(ciphertext, key, nonce)
print("Decrypted text:", decrypted_text.decode())
```

Other mode - XTS mode