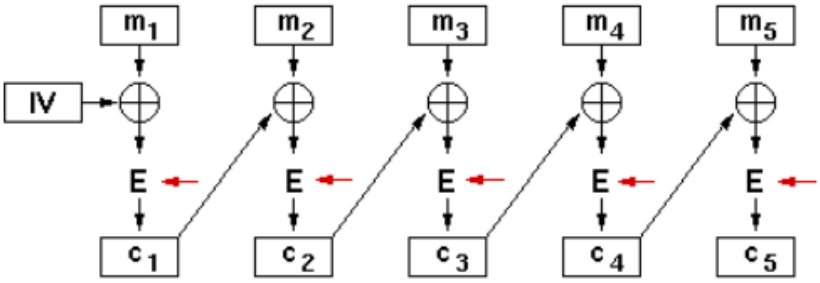
Aufgaben

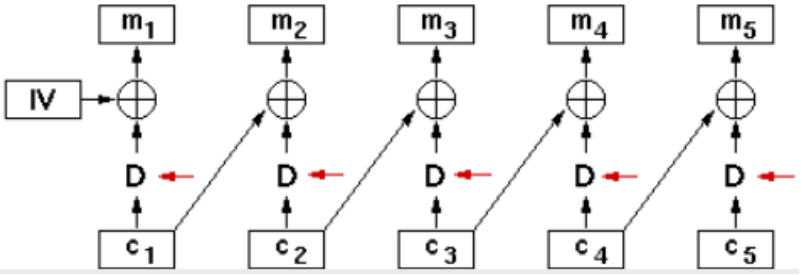
24. Zum CBC Betriebsmodus von Blockciphern: Erklären sie, (i) warum die angegebene Entschlüsselungsformel tatsächlich funktioniert, (ii) wie es zu den beschriebenen Plaintextfehlern nach dem Auftreten eines 1-Bit Ciphertextfehler kommt und (iii) warum CBC self-recovering ist

(i) Wir wissen wenn $a \oplus b = c$ dann $c \oplus a = b$ und $c \oplus b = a$



Hier gilt jeweils $c_i \oplus m_{i+1} = D(c_{i+1})$ mit ausnahme von i = 1 dort nutzt man IV statt einem c_0

somit gilt $D(c_{i+1}) \oplus c_i = m_{i+1}$ bzw. $D(c_1) \oplus IV = m_1$
genau dieses schema sehen wir bei der Entschlüsselung:



für m_1 sehen wir $D(c_1) \oplus IV = m_1$
und für alle folgenden m wird das unentschlüsselte c_i mit dem entschlüsselten $D(c_{i+1})$ XOR-Verknüpft also: $D(c_{i+1}) \oplus c_i = m_{i+1}$

demnach funktioniert die Entschlüsselung aufgrund der grundlegenden Eigenschaften des XOR.

(ii) Wenn ein Bit von c_i Fehlerhaft ist wird aufgrund des Entschlüsselungsmechanismus $D(c_i)$ komplett fehlerhaft sein (confusion und diffusion) beim XOR mit dem nächsten c_{i+1} passiert dort allerdings nur ein 1 Bit Fehler, da das XOR ja erst auf das bereits entschlüsselte $D(c_{i+1})$ angewandt wird, somit wirkt sich der Fehler nur in der XOR Operation aus, was bei einem Falschen Bit eben nur *ein* Falsches Bit bewirken kann:

also gibt es zwei 1 Bit Fehler Situationen:

1:
c_i mit 1 Bit Fehler → Entschlüsseln → komplett Fehlerbehaftetes $D(c_i)$ → XOR mit normalen c_{i-1} → komplett Fehlerbehaftetes m_i

2:
normales c_i → entschlüsseln → normales $D(c_i)$ → XOR mit Fehlerhaftem c_{i-1}
→ 1 Bit Fehler in m_i

Tritt die 1. ein Folgt für das nächste i+1 logischerweise die 2.

(iii) der $i+2$ te Block ist hier dann wieder korrekt, da man dort ja nur $c_{i+1} \oplus D(c_{i+2})$ Rechnet und das Fehlerbehaftete c_i nicht mehr vorkommt. Nachdem also nur jeweils zwei Blöcke von so einem Fehler betroffen sind nennt man CBC self recovering.

---

25. Implementieren sie (natürlich unter zu-Hilfenahme einer oder mehrerer Libraries) ECM und CBC unter DES und bestätigen sie experimentell das Verhalten von CBC bei Ciphertextfehlern.

Mit einer einfachne DES library für Python sieht ECM und CBC folgendermaßen aus:

Library

```
import des
```

- Library

ECM

```
plaintext = b"Lorem Ipsum dolores sit amet, consectetur adipiscing elit. Nullam auctor, nunc
nec lacinia fermentum, nunc nunc fermentum nunc"

encrypted = key.encrypt(plaintext, padding=True)
decrypted = key.decrypt(encrypted, padding=True)

print(decrypted)
```

CBC

```
encrypted = key.encrypt(plaintext, padding=True, initial=b"87654321")
decrypted = key.decrypt(encrypted, padding=True, initial=b"87654321")

print(decrypted)
```

Viel zu erklären gibt es hier nicht, gibt man den funktionen einen initial Wert wird automatisch CBC verwendet.

Modifizieren wir den verschlüsselten text folgendermaßen:

```
encrypted = key.encrypt(plaintext, padding=True, initial=b"87654321")

encrypted = bytearray(encrypted)
encrypted[12] = encrypted[12] ^ 1
encrypted = bytes(encrypted)

decrypted = key.decrypt(encrypted, padding=True, initial=b"87654321")

print(decrypted)
```

sehen wir beim output wie erwartet nur einen Lokalen Fehler:
`b'Lorem Ip\xe3\xe1C\rM\xe4\x1f\x05res rit amet, consectetur adipiscing elit. Nullam auctor, nunc nec lacinia fermentum, nunc nunc fermentum nunc`

26. Implementieren sie (natürlich unter zu-Hilfenahme einer oder mehrerer Libraries) zusätzlich CFB und CTR und führen sie Experimente zum Laufzeitverhalten dieser vier Modi durch. Variieren sie insbesondere für CFB die Menge der aus der verschlüsselten Queue entnommenen bits pro Verschlüsselungsvorgang des Blockciphers (ich habe darauf in der VO bereits hingewiesen).

Zuerst habe ich für CFB und CTR eine andere python library verwendet, da die vorige diese Modi nicht unterstützt:

Library

```
from Crypto.Cipher import DES
from Crypto.Random import get_random_bytes
```

CFB

für CFB brauchte ich zuerst padding funktionen um den input/output auf die richtige Länge zu bringen, da CFB nur vielfache von 8 akzeptiert:

```
def pad(data):
    padding_length = 8 - (len(data) % 8)
    return data + bytes([padding_length] * padding_length)

def unpad(data):
    padding_length = data[-1]
    return data[:-padding_length]
```

Die tatsächlichen encrypt und decrypt funktionen sind den vorherigen allerdings sher ähnlich:

```
def encrypt_cfb(des_key, iv, plaintext):
    cipher = DES.new(des_key, DES.MODE_CFB, iv)
    encrypted = cipher.encrypt(plaintext)
    return encrypted

def decrypt_cfb(des_key, iv, ciphertext):
    cipher = DES.new(des_key, DES.MODE_CFB, iv)
    decrypted = cipher.decrypt(ciphertext)
    return decrypted
```

CTR

CTR sieht wieder sehr ähnlich aus:

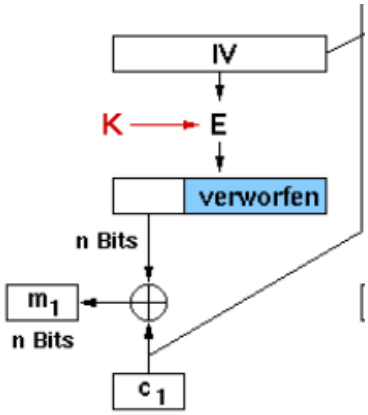
```
def encrypt_ctr(des_key, nonce, plaintext):
    cipher = DES.new(des_key, DES.MODE_CTR, nonce=nonce)
    encrypted = cipher.encrypt(plaintext)
    return encrypted

def decrypt_ctr(des_key, nonce, ciphertext):
    cipher = DES.new(des_key, DES.MODE_CTR, nonce=nonce)
    decrypted = cipher.decrypt(ciphertext)
    return decrypted
```

Variation des Feeback Werts

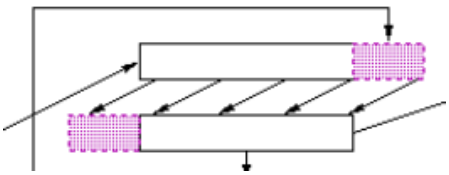
Um zu variieren wie viele der Ciphertext bits in die enqueued werden musste brauchte ich eine custom Implemnentierung von CFB, hier die encrypt Funktion:

```
def custom_encrypt_cfb(des_key, iv, plaintext, feedback_size):
    cipher = DES.new(des_key, DES.MODE_ECB)
    encrypted = b""
    prev_block = iv

    for i in range(0, len(plaintext), feedback_size // 8):
        block = plaintext[i:i + feedback_size // 8]
        cipher_output = cipher.encrypt(prev_block)
        encrypted_block = bytes(a ^ b for a, b in zip(cipher_output, block))
        encrypted += encrypted_block
        prev_block = (prev_block[feedback_size // 8:] + encrypted_block)[-len(prev_block):]

    return encrypted
```

Hier wird jeweils ein Block in größe der feedback size vom plaintext genommen (division durch 8 da feedback size in bits angegeben wird.)



Dieser wird dann mit dem Anfang der verschlüsselten Queue XOR Verknüpft. Wir fügen die Verschlüsselten Daten zu encrpted hinzu welches wir später ausgeben.



Dann entfernen wir den Anfang der queue in größe der feedback size und fügen unseren neuen ciphertext am ende hinzu.

Die Entschlüsselung funktionier analog:

```
def custom_decrypt_cfb(des_key, iv, ciphertext, feedback_size):
    cipher = DES.new(des_key, DES.MODE_ECB)
    decrypted = b""
    prev_block = iv

    for i in range(0, len(ciphertext), feedback_size // 8):
        block = ciphertext[i:i + feedback_size // 8]
        cipher_output = cipher.encrypt(prev_block)
        decrypted_block = bytes(a ^ b for a, b in zip(cipher_output, block))
        decrypted += decrypted_block
```

```
            prev_block = (prev_block[feedback_size // 8:] + block)[-len(prev_block):]

        return decrypted
```

Wir nehmen einen Block in feedback_size größe aus dem ciphertext und verschlüsseln wieder den IV welcher natürlich gleich sein muss wie bei der Verschlüsselung.

Dann XOR Verknüpfen wir diese beiden und speichern den entschlüsselten Block für die spätere Ausgabe. Dann wird nur noch der Ciphertext-Block angehangen und der Anfang aus der Queue entfernt.

Messungen

Zuerst die Standard Modi, dessen Implementierung ich zuerst beschrieben habe:

Mode	Encryption (ms)	Decryption (ms)
ECM	6.055116653442383	4.9991607666015625
CBC	4.999399185180664	6.506681442260742
CFB	1.0006427764892578	0.0
CTR	0.0	0.0

Interresanterweise sind einige der Werte 0ms
nachdem die besonders bei der zweiten Library so ist gehe ich davon aus, dass die Implementierung bei realtiv kurzen Plaintexten schnell genug ist, dass man mit pythons `time.time()` keinen Unterschied erkennt.

Custom CFB

Hier die Werte der custom Variante von CFB mit verschiedenen Feedback Größen:

Feedback Größe	8	16	32
Encryption	1.9989013671875	1.9979476928710938	0.9999275207519531
Decryption	2.052783966064453	1.0006427764892578	0.99945068359375

Spannenderweise ändert sich zwischen 32 und 64 bei der Verschlüsselungszeit kaum etwas, die Entschlüsslung fällt aber bei allen Versuchen bei diesem Schritt unter die Messgrenze.

Zusatz

- Repository