CRYPTO

块加密的工作模式

DES 只能加密 64 位的数据，不能直接用于数据加密。本文介绍 PKCS#5 填充算法，以及分组密码的 ECB、CBC、CFB 工作模式。

RUAN XINGZHI

14 APR 2020 • 8 MIN READ

上一篇文章讨论了 DES 算法，现在我们有了“给定 64-bit 的明文、64-bit 的密钥，输出 64-bit 的密文”的加密手段。这离实际应用还有一点点距离，因为要传递的信息当然不止 64 位。

要用 DES 加密一条信息，一般先把信息填充到 64 的倍数，于是就可以分成许多组，每组 8 个字节。利用密钥对每一组进行加密，最终的结果拼接起来，这就是 ECB(Electronic Code Book，电子密码本) 模式。

PKCS#5

现在我们要把信息填充到 8 的倍数个字节，还得能无歧义地恢复。PKCS#5 就是用来干这个事的。它的规则是：

- 若原文长度不是 8 的倍数，则设其模 8 为 p，在消息后面填充 (8-p) 个 (8-p).
  例子： `12345678hello` 被填充为 `12345678hello\x03\x03\x03`
- 若原文长度是 8 的倍数，则往消息后面填充 8 个 `\x08` .

为什么原文长度已经是 8 的倍数时，还要进行填充？这是为了消除歧义。假设信息本来就是 `hello\x03\x03\x03` ，长度为 8，如果不再填充，解码时可能就认为原文是“hello”.

要解码 PKCS#5 填充的数据，解码器直接去看整个数据的最后一个字节 x，抛弃掉末尾 x 位就得到了原文，这是快速且无歧义的。

```
# PKCS5
def addPadding(s):
```

```
    p = 8 - len(s)%8
    return s + p * chr(p).encode()

def delPadding(s):
    p = s[-1]
    return s[:-p]
```

ECB（电子密码本）

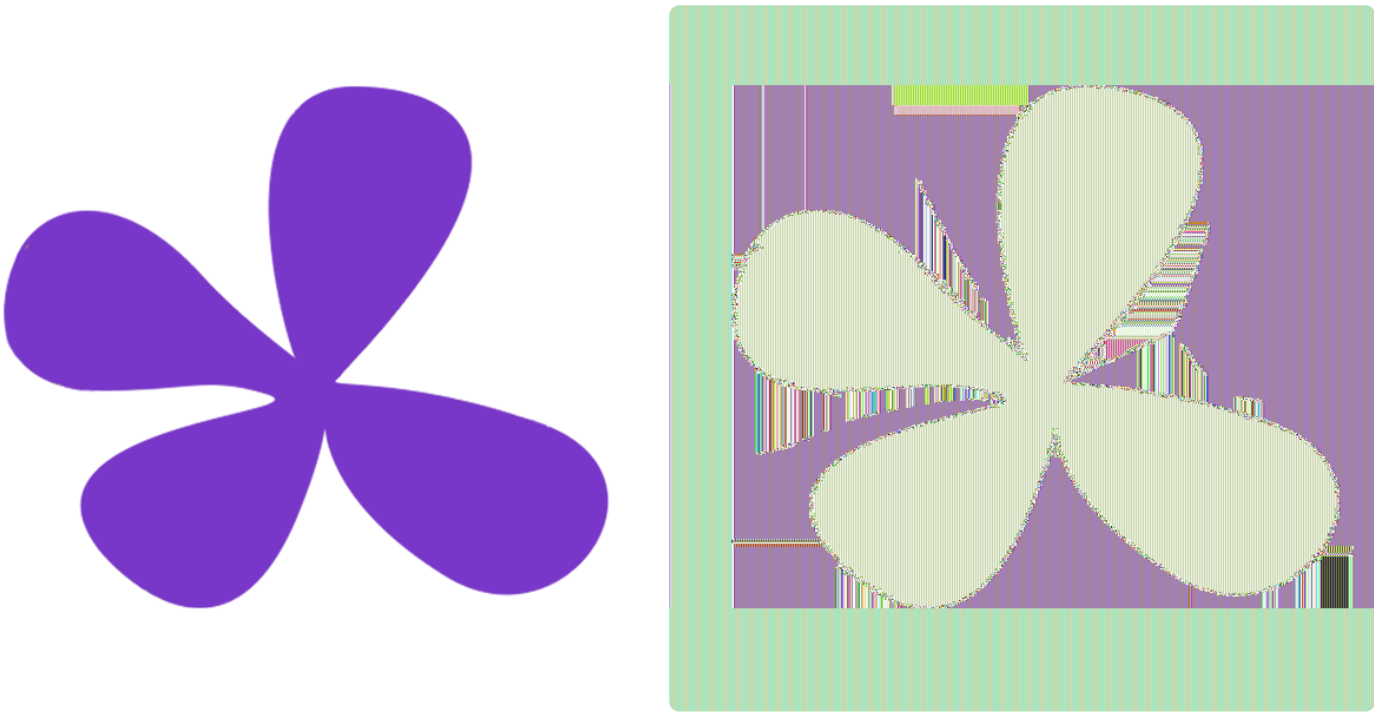
ECB 是最朴素的工作模式。把明文每 8 位一组进行加密，得到的密文直接拼接。代码如下：

```
def des_ecb_enc(plain, key):
    m = addPadding(plain)
    m = [m[x : x+8] for x in range(0, len(m), 8)]

    c = [DES(int.from_bytes(x, 'big'),
             int.from_bytes(key, 'big'),
             'encrypt') for x in m]

    c = [x.to_bytes(8, 'big') for x in c]

    return reduce(lambda x, y: x + y, c)

def des_ecb_dec(enc, key):
    assert len(enc) % 8 == 0

    c = [enc[x : x+8] for x in range(0, len(enc), 8)]

    c = [DES(int.from_bytes(x, 'big'),
             int.from_bytes(key, 'big'),
             'decrypt') for x in c]

    c = [x.to_bytes(8, 'big') for x in c]

    return delPadding(reduce(lambda x, y: x + y, c))

m = b'helloQAQwww'
key = b'lilac666'

c = des_ecb_enc(m, key)
print(c)
# b'w\xb4\n\xccM\xd6\xd1\xcd4\xe6aQ\x0c\x88\x826'
```

```
p = des_ecb_dec(c, key)
print(p)
# b'helloQAQwww'
```

ECB 模式是最简单的模式。由于每个块都是独立加密的，所以它对并行加密解密很友好。此外，传输过程中出现的偶然错误，不会扩大到影响整个数据。但正因为每个块都独立加密，重复的块会被加密成重复的明文。这会导致泄漏明文的一些统计信息。

下面演示一个例子。对 Lilac 的 logo 位图（左）进行加密，得到的结果修复一下文件头，即得到加密后的 logo（右）。

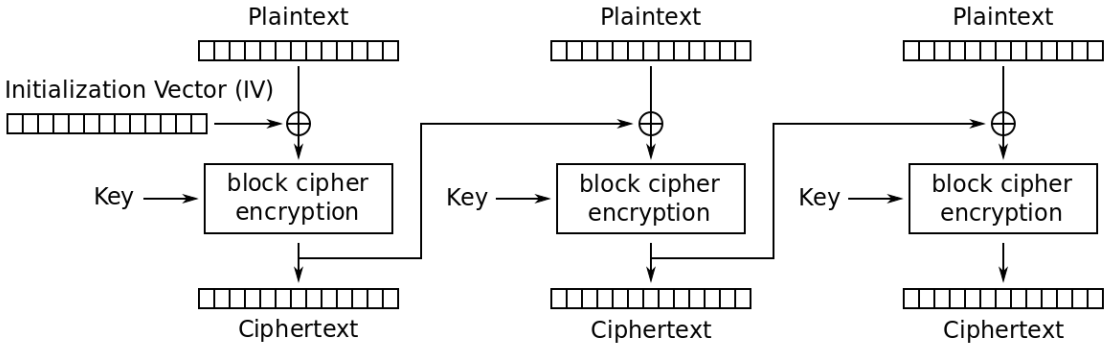


ECB 模式加密的 Lilac 队徽。 左：原图　右：加密后的位图

可见，图像的轮廓被完整地保留了下来。所以 ECB 显然不是一个好的工作模式：尽管窃听者不知道原图的任何一个像素，但他一眼可以看出原图大概是长什么样的。

CBC（密码块链接）

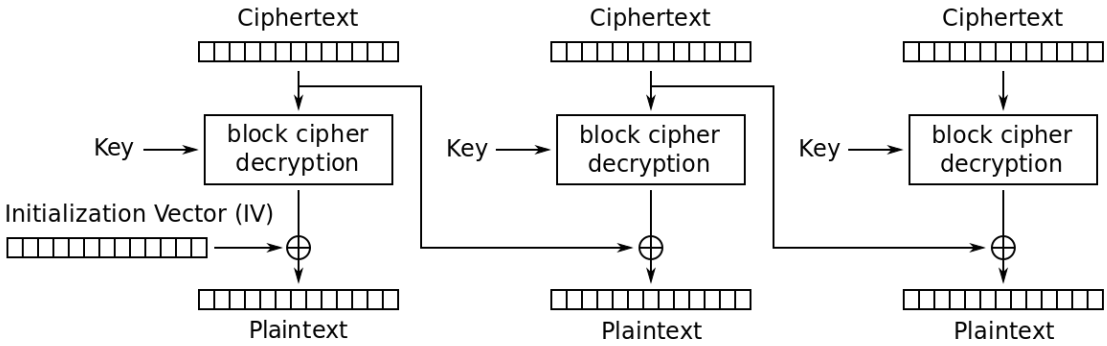
CBC（Cipher-block chaining，密码块链接）模式需要一个初始向量 IV. 加密过程中，先把 IV 与第一块明文混合，再交由 DES 加密；对于下一块，其 IV 采用这一块的加密结果。



Cipher Block Chaining (CBC) mode encryption

CBC 加密流程图。 图源：wikipedia

CBC 由于每一个块在加密的时候，都需要把明文与上一个块的加密结果进行异或，故加密是串行的。不过解密可以并行，因为解密时已知每一个密文块，直接拿来异或即可。



Cipher Block Chaining (CBC) mode decryption

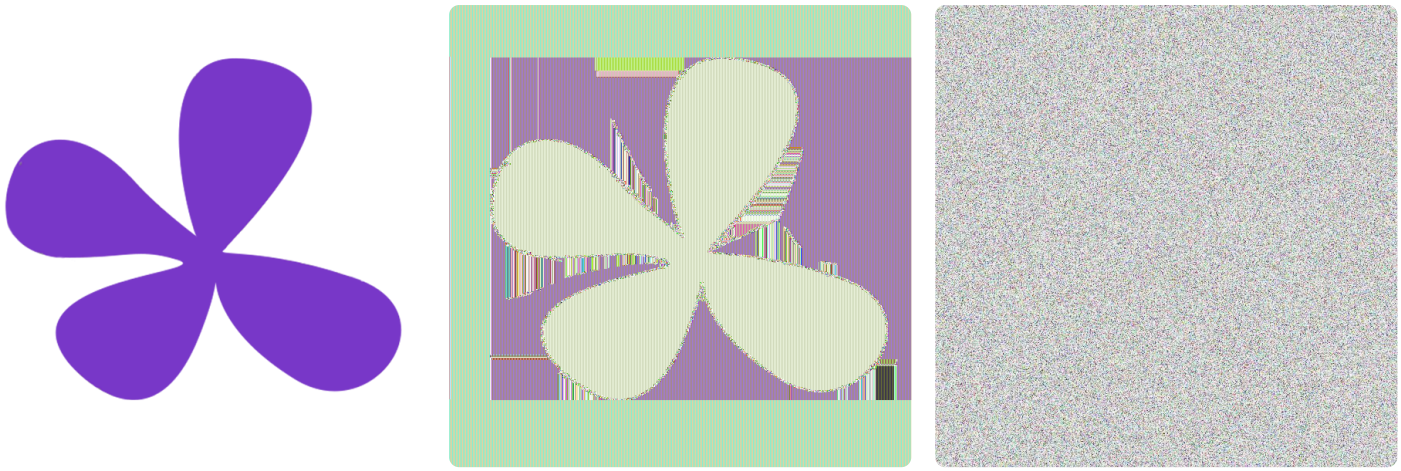
CBC 解密流程图

代码实现如下：

```
def des_cbc_enc(plain, key, iv):
    key = int.from_bytes(key, 'big')
    iv = int.from_bytes(iv, 'big')

    m = addPadding(plain)
    m = [int.from_bytes(m[x : x+8], 'big') for x in range(0, len(m), 8)]

    c = []

    for i in range(len(m)):
        r = iv ^ m[i]
        iv = DES(r, key, 'encrypt')
        c.append(iv)
```

```
    c = [x.to_bytes(8, 'big') for x in c]

    return reduce(lambda x, y: x + y, c)

def des_cbc_dec(enc, key, iv):
    key = int.from_bytes(key, 'big')
    iv = int.from_bytes(iv, 'big')

    c = [int.from_bytes(enc[x : x+8], 'big') for x in range(0, len(enc), 8)]

    m = []

    for i in range(len(c)):
        p = DES(c[i], key, 'decrypt')

        m.append(p ^ iv)
        iv = c[i]

    m = [x.to_bytes(8, 'big') for x in m]

    return delPadding(reduce(lambda x, y: x + y, m))

m = b'helloQAQwww'
key = b'lilac666'
iv = b'ruan1234'

c = des_cbc_enc(m, key, iv)
print(c)
# b'\x93W\xb7\xf5\x93\xd2?\n\xe2\x9bR\x179\x94\xa1\x00'

p = des_cbc_dec(c, key, iv)
print(p)
# b'helloQAQwww'
```

显然，由于明文每次加密开始的时候，都得异或上前一块的密文，故对任何明文的改动，都将会影响到此后的所有块。CBC 克服了 ECB “相同明文块变成相同密文块” 的缺点。以 CBC 模式加密 Lilac logo，得到的结果如右图，就像整个图都是噪点：



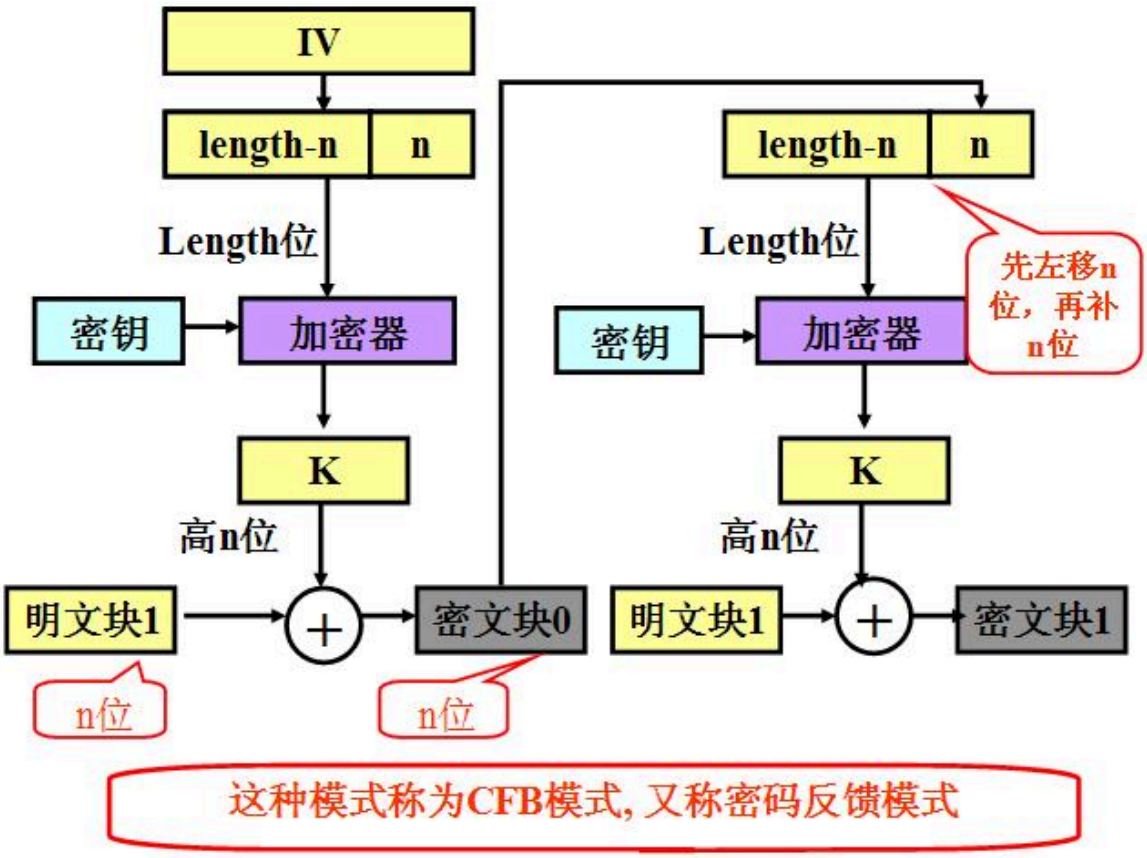
左：原图 中：ECB 加密模式 右：CBC 加密模式

从 CBC 模式加密后的图像上，已经看不出原图的特点。

CFB（密文反馈）

CFB（Cipher feedback，密文反馈）是一种接近于**流密码**的工作模式。它不用 DES 加密明文，而是把 DES 作为伪随机数生成器，把 IV 作为种子。这样就可以从 IV 和指定的 key 生成字节流，然后把明文与这个字节流异或。

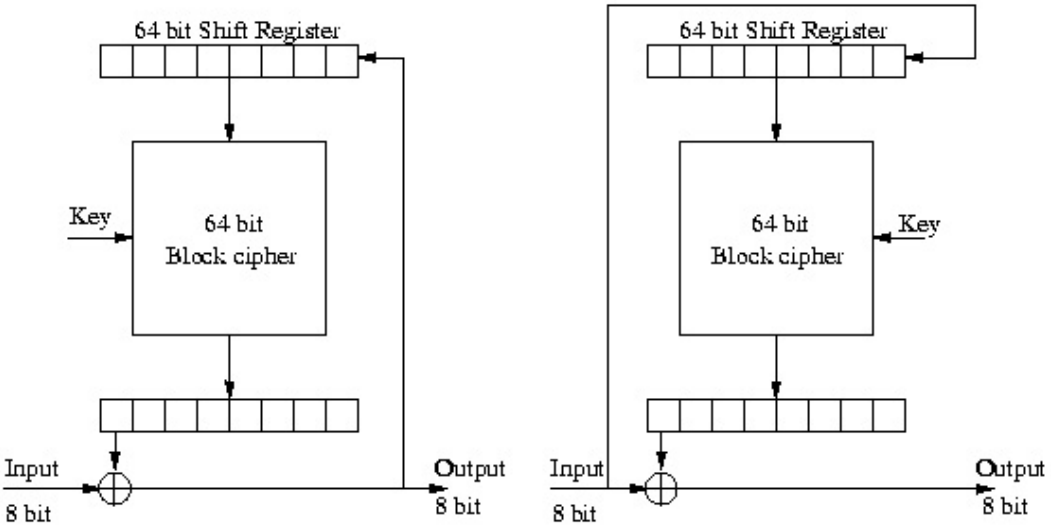
CFB 模式中，每次加密明文的一个字节。故 CFB 模式不需要添加Padding. 工作流程如下：



CFB 加密流程，图片来源：对称加密和分组加密中的四种模式

加密过程中，维护了一个寄存器 `reg` ，长度为 64-bit，初始值为 IV. 每次对明文的一个字节进行加密时，把这个 `reg` 交给 DES 加密，生成一个 64-bit 的结果 `K`；然后把 `K` 的高 8-bit 与明文的这一个字节异或，得到了一个字节的密文。接下来，把这个 8-bit 密文补充到移位寄存器 `reg` 的最右边，抛弃 `reg` 此前的高 8 位。

随着加密的进行， `reg` 不断地变化，每次都往右边加 8 位（也就是上一轮得到的密文）、抛弃掉最左边的 8 位。



CFB 模式的加密（左）和解密（右）。 图片来源：DES加密模式详解

而解密过程也很简单：既然 CFB 模式是流密码，那么解密的时候，只要得到**与加密过程一样的密钥流**，拿去与密文异或就能得到明文了。加密、解密代码如下：

```
def des_cfb_enc(plain, key, iv):
    key = int.from_bytes(key, 'big')
    reg = int.from_bytes(iv, 'big')

    m = [x for x in plain]

    c = []

    for i in range(len(m)):
        K = DES(reg, key, 'encrypt')
        cip = (K >> 56) ^ m[i]

        c.append(cip)
        reg = ((reg << 8) & 0xFFFFFFFFFFFFFFFF) | cip

    c = [x.to_bytes(1, 'big') for x in c]
```

```
    return reduce(lambda x, y: x + y, c)

def des_cfb_dec(enc, key, iv):
    key = int.from_bytes(key, 'big')
    reg = int.from_bytes(iv, 'big')

    c = [x for x in enc]

    m = []

    for cip in c:
        K = DES(reg, key, 'encrypt')
        msg = (K >> 56) ^ cip

        m.append(msg)
        reg = ((reg << 8) & 0xFFFFFFFFFFFFFFFF) | cip

    m = [x.to_bytes(1, 'big') for x in m]

    return reduce(lambda x, y: x + y, m)

m = b'helloQAQwww'
key = b'lilac666'
iv = b'ruan1234'

c = des_cfb_enc(m, key, iv)
print(c)
# b'\xaf\x1a\xba\xeb-\xc2\xaa\x0b\t\xea\x83'

p = des_cfb_dec(c, key, iv)
print(p)
# b'helloQAQwww'
```

与 CyberChef 的结果不一样，但是与 pycrypto 的结果一样。我也不知道为什么。

总结

我们讨论了 ECB, CBC, CFB 三种工作模式，其中

- ECB 模式是不安全的模式。
- 对明文进行改动，CBC、CFB 模式都会使得后序所有块错误。
- CBC 模式，如果密文传输过程中有错误，则后面所有密文都出错。

- CFB 模式，如果密文传输过程中出错，只会影响到之后几个块（因为移位寄存器的存在，在 8 个块之后这个错误就会离开 `reg`，此后的数据都会正常）。
- ECB、CBC 模式需要填充。填充常用 PKCS#5 规则。CFB 模式是类似于流密码的逐字节加密，无需填充。