Kriptográfia és Információbiztonság
5. előadás

MÁRTON Gyöngyvér

Sapientia Egyetem, Matematika-Informatika Tanszék
Marosvásárhely, Románia
mgyongyi@ms.sapientia.ro

2025

Miről volt szó az elmúlt előadáson?

- folyamtitkosító rendszerek: A5/1, Salsa20, ChaCha20,
- a Poly1305 üzenet hitelesítő mód

Miről lesz szó?

- blokktitkosító rendszerek
- blokktitkosító módok: ECB, CBC, CTR
- DES, 3DES, TEA, Blowfish

Blokktitkosító rendszerek

- a blokktitkosító rendszerek a kripto alapelemei, számos rendszer felépítésében játszanak fontos szerepet,
- bájttömbönként alkalmaznak egy álvéletlen függvényt/permutációt,
- a rendszer biztonságát az alkalmazott álvéletlen permutáció határozza meg
- fix méretű blokkokkal végeznek műveleteket,
- minél nagyobb a blokkméret annál biztonságosabb az algoritmus, de annál lassúbb is,
- mit, illetve mit nem oldanak meg?
  - a titkosítás nem lesz randomizált,
  - nem garantálják az üzenet integritását,
  - nem oldják meg a rejtjelezéshez használt kulcs megosztását,
  - nem teszik lehetővé a kommunikáló felek hitelesítését,
- a rendszer biztonságát az alkalmazott álvéletlen permutáció határozza meg.

Blokktitkosító rendszerek

- egyszerre n bájot rejtjeleznek, a kimenet is n bájt lesz,
- a kulcsot bájt tömbönként újra használják $\Rightarrow$ blokktitkosító módok,
- minél nagyobb a kulcs annál biztonságosabb a titkosító, de annál lassúbb,
- egy blokktitkosító rendszer átalakítható folyamtitkosító rendszerré,
- több körből áll a titkosítás, az eredeti kulcs alapján különböző körkulcsokat hoznak létre, amelyeket különböző körökben használnak
- két nagy csoport:
  - Feistel sémán alapuló rendszerek (pl. DES):
    - minden körben felosztják a bemeneti bitsort, egyik részen egy helyettesítést hajtanak végre, amely után felcserélik a két részt,
  - helyettesítést és permutációt alkalmazó rendszerek (pl. AES).

Blokktitkosító módok

- ha a nyílt szöveg nagyobb mint n bájt, akkor:
  - szükséges alkalmazni valamilyen blokk titkosító módot,
  - a nyíltszöveget fel kell osztani n bájtos részekre,
  - a nyíltszöveget "padding"-olni (kiegészíteni) kell
  - `PKCS#7`: ha `N` darab bájtot kell hozzáadni, akkor az `N` értékét adjuk hozzá `N`-szer
- a kulcs többszöri felhasználása:
  - biztonsági problémák,
  - NIST által standardizált blokktitkosító módok,
  - egy adott blokktitkosító mód: különböző biztonsági szintet határoz meg, különböző protokollokban használható.

Blokktitkosítók a gyakorlatban

- legtöbbször iterált szerkezetűek:
  - round function: egy belső $\hat{\mathcal{E}} = (\hat{E}, \hat{D})$ kör függvényt használnak, amely helyettesítést és permutációt hajt végre,
  - key schedule: egy álvéletlenszám generátort, egy *Gen* expansion függvényt alkalmaznak, hogy a *key* kulcs alapján a $k_1, \ldots k_i$ körkulcsokat (round key) meghatározzák
- minden körben más körkulcsot alkalmaznak:

$$(k_1 \ldots, k_i) \leftarrow Gen(key)$$

$$y \leftarrow \hat{Enc}(k_i, \hat{Enc}(k_{i-1}, \ldots, \hat{Enc}(k_2, \hat{Enc}(k_1, x)) \ldots))$$

- $\hat{Enc}(\cdot)$-nak két bemenete van: az aktuális állapotérték, és a megfelelő körkulcs, ahol a kezdő állapotérték a nyíltszöveg,
- úgy a nyíltszöveg, mint a rejtjelezett szöveg pontosan meghatározott méretű lesz pl. az AES-nél 128 bit,
- a visszafejtő függvény ugyanaz, a körkulcsokat azonban fordított sorrendben kell alkalmazni:

$$x \leftarrow \hat{Dec}(k_1, \hat{Dec}(k_2, \ldots, \hat{Dec}(k_{i-1}, \hat{Dec}(k_i, y)) \ldots))$$

Blokktitkosítók a gyakorlatban

	kulcsméret (bitek)	blokkméret (bitek)	körök száma	hatékonyság (MB/sec)
DES	56	64	16	80
3DES	168	64	48	30
AES-128	128	128	10	163
AES-256	256	128	14	115

- a gyakorlati tapasztalat azt mutatja, hogy az iterált szerkesztés biztonságos blokktitkosítót eredményez
- egy lineáris függvény iterálása nem eredményez biztonságos blokktitkosítót
- a blokktitkosítók tervezése, implementálása nem egy triviális feladat.

Blokktitkosítók a gyakorlatban

- a kör függvény (round function) helyettesítést és permutációt alkalmaz
- helyettesítés:
  - n darab bit másik, különböző n darab bittel való helyettesítését jelenti
  - a műveletet végző függvényt S-boxnak is hívják, amely substitution vagy akár secret boxot is jelent,
- permutáció: n darab bit permutálása, nem az értéküket, hanem a sorrendjüket változtatják meg,
- több kört alkalmaznak, amelynek során helyettesítést, permutációt végeznek, majd az utolsó kör végén egy permutációt hajtanak végre

Blokktitkosítók a gyakorlatban

- a biztonság növelése érdekében a kulcsokat *fehéríteni* (whitening) szokták
- az első kör előtt és az utolsó kör után egy-egy $\oplus$ műveletre kerül sor:
  - titkosításkor:
    - a blokktitkosító alkalmazása előtt meghatározzák a nyíltszöveg és egy k_1 kulcs $\oplus$ értékét,
    - a blokktitkosítás után kapott rejtjelezett blokk értékének és egy másik k_2 kulcsnak határozzák meg az $\oplus$ értékét:

$$c = E_{key}(k_1, k_2, m) = (E_{key}(m \oplus k_1)) \oplus k_2$$

  - visszafejtéskor:

$$m = D_{key}(k_1, k_2, c) = (D_{key}(c \oplus k_2)) \oplus k_1$$

- a blokktitkosító kevésbé lesz támadható brute-force-al: a k_1, k_2 alkalmazásával ugyanis megnő a lehetséges kulcsok száma

Blokktitkosító módok, jellemzők

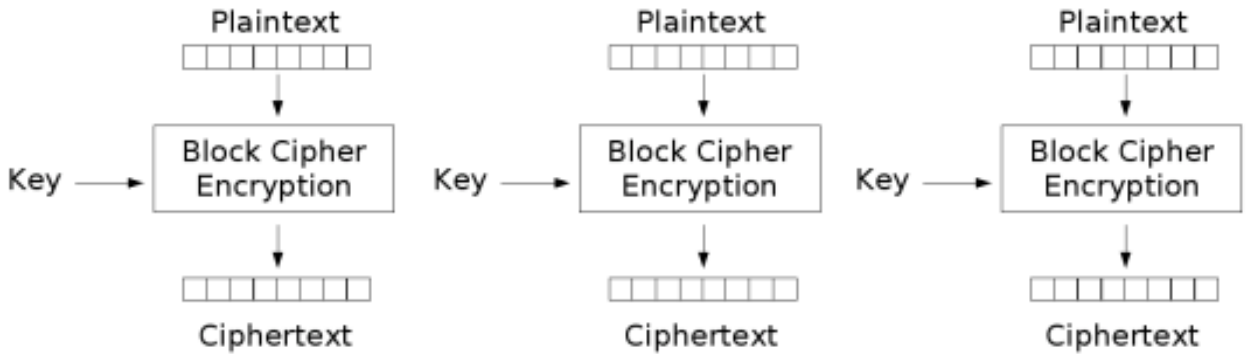
a következőket a DES-titkosító számára dolgozták ki még 1981-ben:

blokktitkosító mód	alkalmazhatóság
ECB	csak egy blokk például a kulcs titkosítására használható, másképp nem biztonságos
CBC	általános blokktitkosító, a titkosítás nem, de a visszafejtés párhuzamosítható
CFB	a titkosítást átalakítja folyamtitkosítóvá
OFB	a titkosítást átalakítja folyamtitkosítóvá, hibajavító kódok esetében használják

- bármilyen blokktitkosító esetében használhatóak
- további blokktitkosító módok: CTR, általános blokktitkosító, nagyon gyors
- nem biztosítják az üzenetek sértetlenségét, manipulálását

ECB (Electronic Code Book)

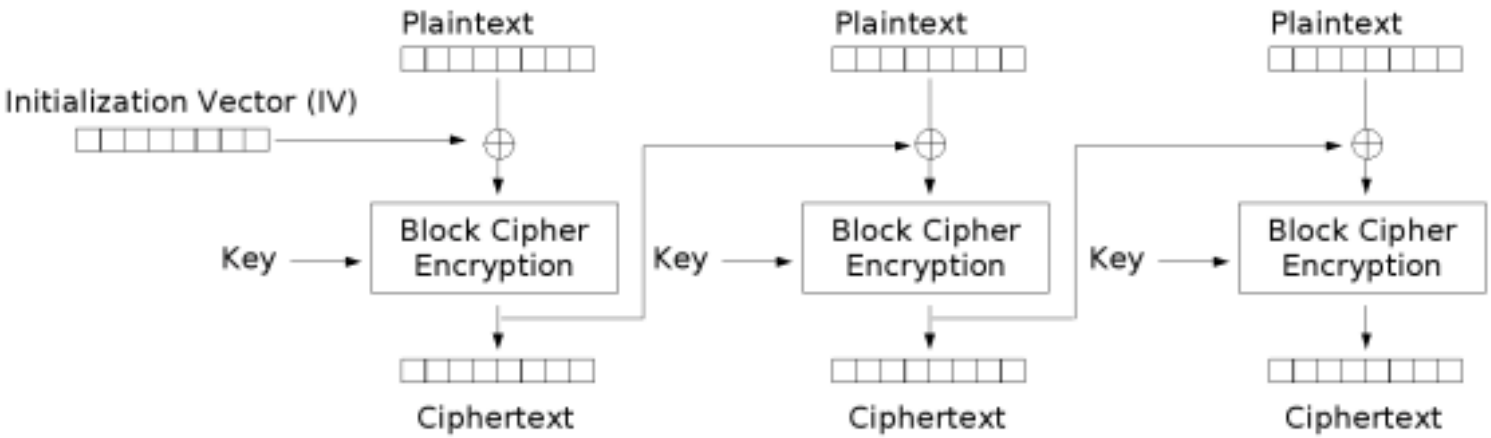
- a nyíltszöveg blokkjait egymástól függetlenül titkosítják, a titkosított szöveg a titkosított blokkok konkatenációja lesz, visszafejtésnél ugyanígy járnak el,
- rövid adathalmaz titkosítására használják, pl. kulcsok,
- biztonsági problémák:
  - ha két blokk egyforma, akkor ezeknek a rejtjelezett értéke is egyforma lesz
  - anélkül, hogy a kommunikáló felek észrevennék egy támadó felcserélhet két rejtjelezett blokkot, vagy kitörölhet, kicserélhet egy-egy blokkot,



Képek forrásai: *http://en.wikipedia.org/wiki/Block_cipher_modes_of_operation*

CBC (Cipher-Block Chaining)

- titkosítás: a titkosító bemenete az aktuális nyíltszöveg és az előző rejtjelezett szöveg $\oplus$ szerint meghatározott értéke,



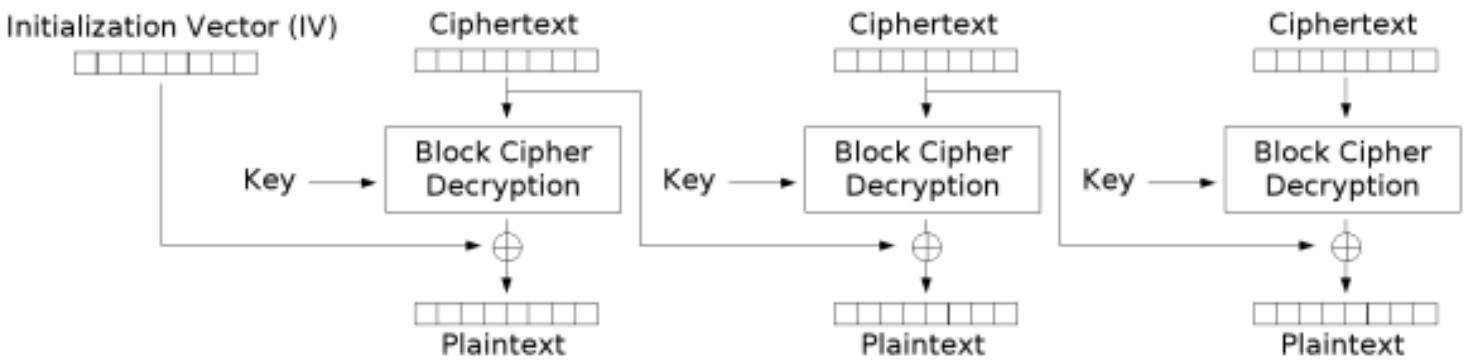
$$C_0 = IV, C_i = E_{key}(M_i \oplus C_{i-1}), i = 1, 2, \ldots$$

CBC (Cipher-Block Chaining)

- egy blokk titkosítása az előző blokk titkosított értékétől függ,
- az első blokk esetében egy kezdeti bájttömbböt, egy IV-t (initialization vector) használunk:
  - az IV-t nem muszáj titokban tartani, de kiszámíthatatlan kell legyen,
  - az IV lehet
    - egy időpecsét (time stamp),
    - egy számláló (counter),
    - álvéletlen módon generált bájttömb, amelyet nonce-nak (a number used only once) is hívnak,
- ha a nyílt szöveg egy bitje megváltozik (pl. sérül az adattovábbítás során), akkor a megfelelő rejtjelezett blokk értéke és az utána következő még két blokk rejtjelezett értéke meg fog változni,
- használata:
  - nagy adathalmaz titkosításakor,
  - üzenet hitelesítő kódok szerkesztésekor

CBC (Cipher-Block Chaining)

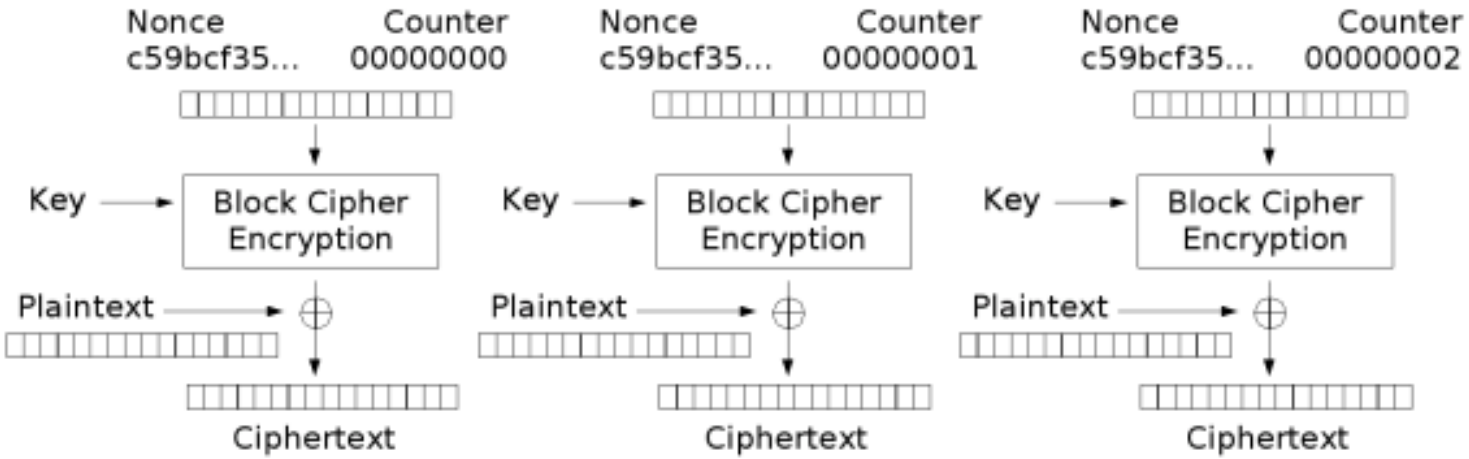
- visszafejtés: visszafejtjük az aktuális blokkot, majd meghatározzuk az eredmény és az előző rejtjelezett blokk $\oplus$ értékét



$$C_0 = IV, M_i = D_{key}(C_i) \oplus C_{i-1}, i = 1, 2, \ldots$$

CTR (Counter mode)

- egy folyamtitkosítást hajt végre,
- egy számláló és egy nonce érték alapján, a nyíltszövegtől függetlenül egy blokk titkosítót alkalmazva egy kulcsfolyamot generál,
- a kapott kulcsfolyamot és a nyíltszöveg blokkjait $\oplus$-al összeadja

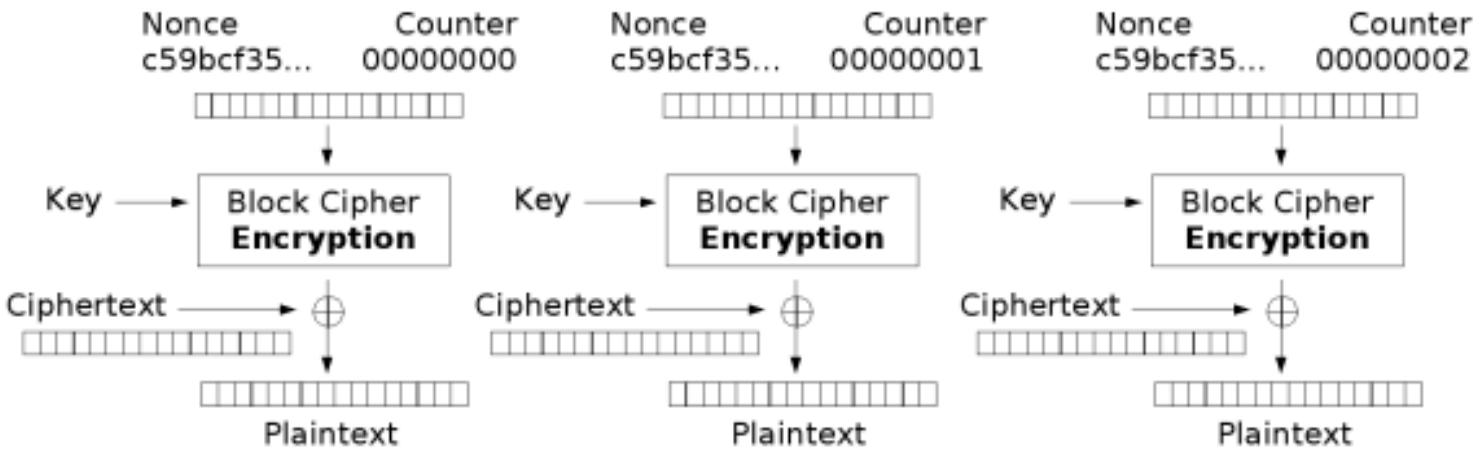


$$K_i = E_{key}(\texttt{Nonce}, \texttt{Counter}_i), C_i = K_i \oplus M_i, i = 1, 2, \ldots$$

CTR (Counter mode)

visszafejtés:

- újra kigenerálják a kulcsfolyamot,
- mivel az $\oplus$ művelet szimmetrikus, a titkosítás és visszafejtés ugyanaz



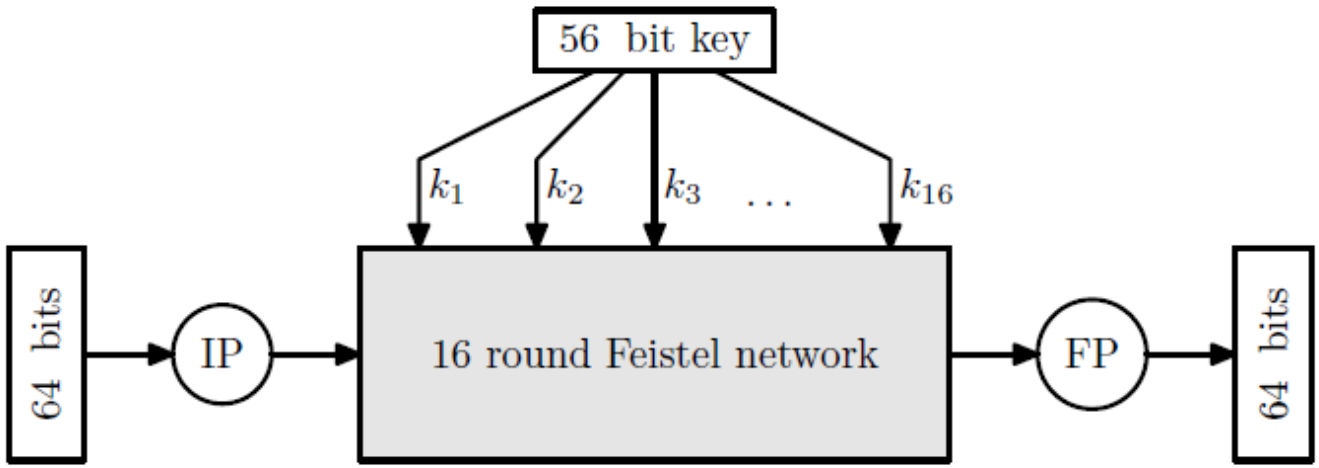
$$K_i = E_{key}(\texttt{Nonce}, \texttt{Counter}_i), M_i = K_i \oplus C_i, i = 1, 2, \ldots$$

DES (Data Encryption Standard)

- az IBM tervezte a NIST kérésére, 1975-ben fogadták el standardként,
- megjelenésével a kriptoanalizis is fejlődésnek indult,
- elődje a Lucifer blokktitkosító, amelynek 128 bites volt a blokkmérete, kulcsmérete szintén 128 bit volt,
- a NIST kisebb kulcsmérettel rendelkező blokktitkosítót kért, így a DES kulcsmérete 56 bit lett,
- azóta számos tudományos elemzés látott napvilágot, és felmerült a gyanú hogy szándékos volt a kis kulcsméret melletti döntés, illetve hogy backdoort tartalmaz,
- a biztonság növelése érdekében bevezették a triple-DES-t: DES-EDE (DES-encryption-decryption-encryption)
- a 3DES megőrzi a DES belső szerkezetét, háromszor nagyobb a kulcsmérete, de háromszor olyan lassú,
- smart kártyák esetében 3DES-t használnak a MAC előállítására,
- a NIST kormányzati célokra, 2030-ig elfogadta a triple-DES használatát,
- 2002-ben váltja fel az AES.

DES (Data Encryption Standard)

- iterációszáma (a körök száma): 16, blokkmérete 64 bit,
- az iterációt megelőzően az 56 bites kulcsot minden 7 bit után kiegészítik egy paritásbitel, amely eredményeként egy 64 bites kezdeti kulcsot kapnak, ez alapján lesz kigenerálva a további 16 kulcs

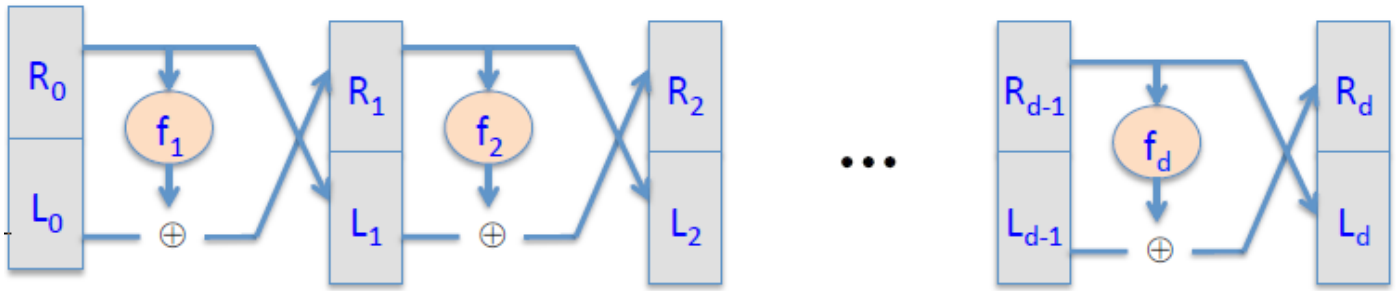


kép forrása: *Boneh and Shoup/DES*

DES, titkosítás

Feistel sémán alapszik:

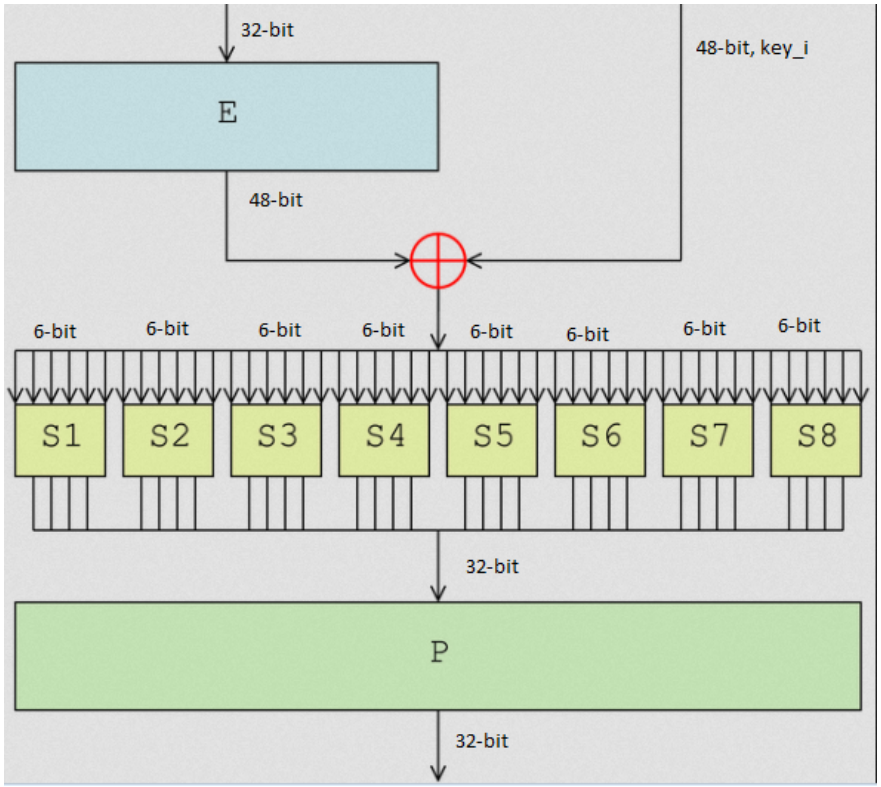
- minden körben felosztják a bemeneti bitsort,
- egyik részen permutációt (P-box) és helyettesítést (S-box) hajtanak végre (f_i alkalmazása),
- majd megcserélik a két részt:



kép forrása: *Boneh/Online Cryptography*

DES, titkosítás

- **E**: expansion művelet, a 32 bites bemenetből 48 bitet hoz létre
- **P**: permutációs művelet, a 32 bites bemenetnek meghatározza egy permutációját,
- **S-boxok**: S1, S2, S3..., helyettesítő, művelet, minden S box a bemeneti 6 bit alapján létrehoz 4 bitet.



kép forrása: *wiki/DES*

DES, S-boxok

S_1															
14	4	13	1	2	15	11	8	3	10	6	12	5	9	0	7
0	15	7	4	14	2	13	1	10	6	12	11	9	5	3	8
4	1	14	8	13	6	2	11	15	12	9	7	3	10	5	0
15	12	8	2	4	9	1	7	5	11	3	14	10	0	6	13
S_2															
15	1	8	14	6	11	3	4	9	7	2	13	12	0	5	10
3	13	4	7	15	2	8	14	12	0	1	10	6	9	11	5
0	14	7	11	10	4	13	1	5	8	12	6	9	3	2	15
13	8	10	1	3	15	4	2	11	6	7	12	0	5	14	9
								⋮							

$S_i(B)$-t, ahol $B = b_1 b_2 b_3 b_4 b_5 b_6$ a következőképpen határozzuk meg:

- $b_1 b_6$ meghatározza a sorindexet,
- $b_2 b_3 b_4 b_5$ meghatározza az oszlopindexet,
- $S_i(B)$, a megfelelő indexnél levő érték bináris alakja,

Példa: $S_1(111100) = 0101$, mert $b_1 b_6 = 10$ (2. sor), és $b_2 b_3 b_4 b_5 = 1110$ (14. oszlop). Az itt található érték 5, aminek bináris alakja 0101.

DES, biztonság

- Az alkalmazott alapműveletek:
  - helyettesítés (substitution): az F alkalmazása, majd a $\oplus$,
  - permutáció (permutation): a két 32 bites rész felcserélése,
  - Shannon: helyettesítést és permutációt alkalmazva megváltozik nyílt-üzenet statisztikai tulajdonsága $\Rightarrow$ nem alkalmazhatóak statisztikai elemzések a feltörés során.
- Az S-, és P-boxok megválasztása,
  - nem lehetnek lineárisak,
  - az S-boxban elvégzett műveleteken kívül minden más művelet lineáris,
  - nem lehetnek véletlenszerűek,
  - mai napig sem publikus az a kritériumrendszer ami alapján az S-boxokat szerkesztették $\rightarrow$ az a vélemény, hogy ha ez nyilvános lenne, akkor könnyebb lenne a kriptoanalízis.
- a visszafejtés ugyanaz, mint a titkosítás, csak fordított sorrendben kell alkalmazni a kulcsokat $\rightarrow$ nem kell két algoritmust írni.
- érvényesül a lavina effektus,
- a biztonság fokozása: előbb tömörítik a nyíltszöveget utána titkosítják,

3DES, triple DES

- a feltöréshez szükséges idő: 2^{56}, ami komoly veszélyt jelent $\rightarrow$ ma már nem használjuk, helyette a $3DES$-t használjuk, amely NIST standard.
- 3DES:
  - a biztonság megerősítése anélkül, hogy a DES belső szerkezetén változtatnánk,
  - háromszor lassúbb, mint a DES,
  - a kulcsméret: $3 \times 56 = 168$ bit lesz,
  - legyen $E : K \times M \rightarrow M$ a DES blokktitkosító, ekkor $3E : K^3 \times M \rightarrow M$, ahol

$$3E((key_1, key_2, key_3), m) = E(key_1, D(key_2, E(key_3, m))),$$

  - hardver implementációnál fontos, hogy $key_1 = key_2 = key_3$, mert ekkor a DES-t kapom,
  - az összes kulcs kipróbálásának módszerével a feltőréshez szükséges idő: 2^{168}, de létezik 2^{118} nagyságrendű feltörési algoritmus is,

DES3, CBC mód, Python

```
import os
from Crypto.Cipher import DES3
from Crypto.Random import get_random_bytes

BS = DES3.block_size
CS = 128 * BS

def encryptCBC(inFile, outFile, key, iv):
    fileSize = os.stat(inFile)
    nrBlock = fileSize.st_size // CS
    with open(inFile, 'rb') as inF, open(outFile, 'wb') as outF:
        cipher = DES3.new(key, DES3.MODE_CBC, iv)
        for i in range(nrBlock):
            tBlock = inF.read(CS)
            tBlock = cipher.encrypt(tBlock)
            outF.write(tBlock)
        tBlock = inF.read(CS)
        rem = len(tBlock) % BS
        N = BS - rem
        if rem == 0: tBlock += bytes([BS]) * BS
        else: tBlock += bytes([N] * N)
        tBlock = cipher.encrypt(tBlock)
        outF.write(tBlock)
```

DES3, CBC mód, Python

```
def decryptCBC(inFile, outFile, key, iv):
    fileSize = os.stat(inFile)
    nrBlock = fileSize.st_size // CS
    with open(inFile, 'rb') as inF, open(outFile,'wb') as outF:
        cipher = DES3.new(key, DES3.MODE_CBC, iv)
        for i in range(nrBlock):
            tBlock = inF.read(CS)
            tBlock = cipher.decrypt(tBlock)
            outF.write(tBlock)
        tBlock = inF.read(CS)
        tBlock = cipher.decrypt(tBlock)
        N = tBlock[-1]
        tBlock = tBlock[:-N]
        outF.write(tBlock)
```

DES3, CBC mód, Python

```
iv = get_random_bytes(BS)
print('blocksize: ', BS)
keyS1, keyS2 = DES3.key_size
key = get_random_bytes(keyS1)
print(key)

plaintextFile = ...
encryptedFile = ...
decryptedFile = ...

encryptCBC(plaintextFile, encryptedFile, key, iv)
decryptCBC(encryptedFile, decryptedFile, key, iv)

with open(plaintextFile, 'rb') as inF1, open(decryptedFile, 'rb') as inF2:
    if inF1.read() == inF2.read():
        print('A titkosítás és visszafejtés sikeres volt!')
    else:
        print('A titkosítás és visszafejtés sikertelen!')
```

DES3, CTR mód, Python

```
import os

from Crypto.Cipher import DES3
from Crypto.Random import get_random_bytes
from Crypto.Util import Counter

BS = DES3.block_size
CS = 128 * BS

def encryptCTR(inFile, outFile, key, nonce):
    fileSize = os.stat(inFile)
    nrBlock = fileSize.st_size // CS
    with open(inFile, 'rb') as inF, open(outFile, 'wb') as outF:
        ctr = Counter.new(BS*8//2, prefix=nonce)
        cipher = DES3.new(key, DES3.MODE_CTR, counter=ctr)
        for i in range(nrBlock + 1):
            tBlock = inF.read(CS)
            tBlock = cipher.encrypt(tBlock)
            outF.write(tBlock)
```

DES3, CTR mód, Python

```
def decryptCTR(inFile, outFile, key, nonce):
    fileSize = os.stat(inFile)
    nrBlock = fileSize.st_size // CS
    with open(inFile, 'rb') as inF, open(outFile,'wb') as outF:
        ctr = Counter.new(BS*8//2, prefix=nonce)
        cipher = DES3.new(key, DES3.MODE_CTR, counter=ctr)
        for i in range(nrBlock + 1):
            tBlock = inF.read(CS)
            tBlock = cipher.decrypt(tBlock)
            outF.write(tBlock)
```

DES3, CTR mód, Python

```
nonce = get_random_bytes(BS//2)
keyS1, keyS2 = DES3.key_size
key = get_random_bytes(keyS1)
print(key)

plaintextFile = ...
encryptedFile = ...
decryptedFile = ...

encryptCTR(plaintextFile, encryptedFile, key, nonce)
decryptCTR(encryptedFile, decryptedFile, key, nonce)

with open(plaintextFile, 'rb') as inF1, open(decryptedFile, 'rb') as inF2:
    if inF1.read() == inF2.read():
        print('A titkosítás és visszafejtés sikeres volt!')
    else:
        print('A titkosítás és visszafejtés sikertelen!')
```

TEA (Tiny Encryption Algorithm)

- 1994-ben publikálta a Cambridge Egyetem két tanára, Roger Needham és David Wheeler
- tervezésekor feladtak egy keveset a biztonságból, hogy a blokktitkosító könnyen implementálható, hatékony legyen
- Feistel típusú, a titkosító és visszafejtő algoritmusok különböznek
- könnyen implementálható hardver-re, illetve softver-re
- ha két üzenetet olyan kulcsokkal titkosítunk, amelyek kapcsolódnak egymáshoz, akkor kivitelezhető a "related key" típusú támadás
- az XTEA egy későbbi változat, amely kiküszöböli a "related key" támadást

TEA, specifikáció

- az ajánlott körök száma: 64, a minimális körszám: 32,
- 128 bites kulccsal dolgozik, amit négy 32 bites blokkra oszt,
- minden matematikai műveletet $\pmod{2^{32}}$-ben végez
- a blokk mérete 64 bit, amelyet két 32 bites részre oszt: L (Left), illetve R (Right) részek,
- az L és R 8 bájtos szekvenciákat 256-os számrendszerbeli számjegyeknek tekinti, amelyekből létrehoz egy tízes számrendszerbeli számot, amivel a további számításokat végzi
- alkalmaz egy konstanst: `delta = 0x9e3779b9 = 2654435769`, azaz `delta` a $2^{32}/phi$ osztási egész része, ahol $phi = 1.6180339887$ az aranyarány,
- a phi használatával oldja meg az algoritmus, hogy minden körben más lesz a titkosítás.

TEA, implementáció

```
from Crypto.Random import get_random_bytes

def teaMain():
    blockSize, keySize = 8, 16
    rawKey = [get_random_bytes(4) for i in range(4)]
    key = [int.from_bytes(k) for k in rawKey]

    pText = b'plaintxt'
    L, R = byteToLR(pText[0 : 8], blockSize)
    L, R = encrypt(L, R, key)
    cryptedBlock = byteFromLR(L, R, blockSize)
    print('crypted blokk: ', cryptedBlock.hex())

    L, R = byteToLR(cryptedBlock, blockSize)
    L, R = decrypt(L, R, key)
    pTextK = byteFromLR(L, R, blockSize)
    print('decrypted text: ', pTextK.decode())
```

TEA, implementáció

```
mask = 0xFFFFFFFF
def encrypt(L, R, Key):
    delta = 0x9e3779b9
    sum = 0
    for i in range(0, 32):
        sum += (delta) & mask
        L = (L + (((R << 4) + Key[0]) ^ (R + sum) ^ ((R >> 5) + Key[1]))) & mask
        R = (R + (((L << 4) + Key[2]) ^ (L + sum) ^ ((L >> 5) + Key[3]))) & mask
    return L, R

def decrypt(L, R, Key):
    delta = 0x9e3779b9
    sum = (delta << 5) & mask
    for i in range(0, 32):
        R = (R - (((L << 4) + Key[2]) ^ (L + sum) ^ ((L >> 5) + Key[3]))) & mask
        L = (L - (((R << 4) + Key[0]) ^ (R + sum) ^ ((R >> 5) + Key[1]))) & mask
        sum = (sum - delta) & mask
    return L, R
```

TEA, implementáció

```
def byteToLR(byteLs, len):
    L = int.from_bytes(byteLs[ : len//2])
    R = int.from_bytes(byteLs[len//2 : ])
    return L, R

def byteFromLR(L, R, len):
    tempL = L.to_bytes(len//2)
    tempR = R.to_bytes(len//2)
    return tempL + tempR
```

Blowfish

- Bruce Schneier szerkesztette 1993-ban,
- szabadon felhasználható,
- Feistel típusú,
- a körök száma: 16
- a kulcsméret változó: 32 bittől 448 bitig,
- a blokk mérete 64 bit,
- a körkulcsok meghatározásához és az S-boxok inicializálása során felhasználja a π tizedesjegyeit,
- nem találtak ellene hatékony támadást,
- bonyolult a kulcsfeldolgozása, ezért ha gyakran szükséges egy új kulcs kigenerálása, akkor nem ajánlott a használata

Twofish, bcrypt

A Blowfish egyik alternatív változata a Twofish:

- a NIST által 1997-ben meghírdetett pályázat egyik kiválasztottja volt (a nyertes az AES lett), nem standardizálták,
- lassúbb volt az AES-128-nál, de gyorsabb az AES-256-nál,
- nincs levédve, bárki szabadon használhatja,
- egyike azon néhány titkosítónak, amelyeket az OpenPGP szabvány (RFC 4880) tartalmaz
- a PGP (Pretty Good Privacy) az emailek titkosítására használja,
- alkalmazza a kulcs fehérítést,

A bcrypt-et a Blowfish titkosító alapján szerkesztették 1999-ben,

- jelszavak tárolására használják több Unix alapú operációs rendszerben alapértelmezett
- egy véletlenszerű értéket használ, a *salt*-ot, szivárvány táblán alapuló támadással szemben