Kriptográfia és Információbiztonság
4. előadás

MÁRTON Gyöngyvér

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

2023

Miről volt szó az elmúlt előadáson?

- titkos-kulcsú rendszerek: matematikai modell, osztályozások
- folyam-titkosító rendszerek: OTP, RC4, linear feedback shift register, A5/1, Salsa20

Miről lesz szó?

- blokk-titkosító rendszerek
- blokk-titkosító módok
- a Crypto++ könyvtárcsomag
- 3DES ECB, CBC módban, C++ kód
- a DES (Data Encryption Standard): specifikáció, biztonság, tervezési szempontok
- TEA, Blowfish

Blokk-titkosító rendszerek

- bájt-tömbönként alkalmaznak egy álvéletlen függvényt/permutációt,
- a rendszer biztonságát az alkalmazott álvéletlen függvény (pseudo-random function), illetve álvéletlen permutáció (pseudo-random permutation) határozza meg,
- a kulcsot bájt-tömbönként újra használják,
- fix méretű tömbökkel végeznek műveleteket: n-bites nagyságú nyílt-szöveg rejtjelezett értékét határozzák meg, a kimenet is n-bites lesz,
- blokk-titkosító módot alkalmaznak:
  - hogy elkerüljék a kulcs többszöri felhasználásának problémáját,
  - hogy tetszőleges hosszúságú üzenetet legyenek képesek titkosítani,
  - hogy randomizálják a titkosítási folyamatot,
- egy blokktitkosító rendszer átalakítható folyamtitkosító rendszerré, ha valamilyen blokk-titkosító módot használunk, pl. CFB, CTR.

Blokk-titkosító rendszerek

1. értelmezés

Legyen $F = F(n) : K_n \times X \to Y$ egy függvénycsalád, ahol az F értékei meghatározhatóak polinom időben (hatékonyan). $K_n = \{0,1\}^n$-el a kulcsok halmazát jelöljük, ahol n egy biztonsági paraméter általában a kulcs hossza. Egy rögzített $key \in K_n$-re az $F_{key} : X \to Y$ biztonságos pszeudorandom függvény, ha a key hiányában nem lehet polinom időben különbséget tenni az F_{key} és egy valódi random függvény között.

2. értelmezés

Legyen $F = F(n) : K_n \times X \to X$, ahol az F értékei meghatározhatóak polinom időben, $K_n = \{0,1\}^n$ a kulcsok halmaza, n egy biztonsági paraméter, a kulcs hossza és bármely $key \in K$-ra F_{key} egy permutáció, azaz egy bijektív függvény. Az $F_{key} : X \to X$ biztonságos pszeudorandom permutáció ha nem lehet polinom időben különbséget tenni az F_{key} és egy valódi random permutáció között.

- Nem egy triviális feladat biztonságos pszeudorandom függvényt, illetve pszeudorandom permutációt szerkeszteni.
- Egy pszeudorandom permutáció egyben pszeudorandom függvény is.

Blokk-titkosító rendszerek

3. értelmezés

Az $E = E(n) : K_n \times X \to X$ permutáció, ahol $K_n = \{0,1\}^n$, n egy biztonsági paraméter a kulcs hossza egy blokk titkosító, ha

- *bármely $key \in K_n$-re létezik polinom idejű (hatékony), determinisztikus algoritmus az $E(key, x)$ értékek meghatározására, ahol $x \in X$,*
- *létezik polinom idejű algoritmus az E inverz értékeinek a meghatározására,*
- *az $|X| = l$ a blokktitkosító blokk mérete lesz.*

- A blokk titkosítók a pszeudorandom permutációk konkrét esetei, ahol minden $key \in K_n$-re egy bijektív függvény (permutáció) adható meg.
- A pszeudorandom függvények átalakíthatóak pszeudorandom generátorokká, a következőképpen:
  - legyen $E = E(n) = K_n \times \{0,1\}^n \to \{0,1\}^n$ egy pszeudorandom függvény
  - ekkor $G = G(n) : K_n \to \{0,1\}^{nt}$ pszeudorandom generátor lesz

$$G_{key} = E_{key}(0)||E_{key}(1)||\ldots||E_{key}(t)$$

Blokk-titkosító rendszerek

A blokk titkosítók két nagy csoportra oszthatóak:

- Feistel sémán alapuló rendszerek (DES):
  - a titkosítás több körből áll,
  - az eredeti kulcs alapján különböző kör-kulcsokat (round key) hoznak létre, amelyeket a megfelelő körökben használnak fel,
  - minden körben felosztják a bemeneti bitsort, egyik részen permutációt és helyettesítést (S-box alkalmazása) hajtanak végre, amely után megcserélik a két részt,
- Helyettesítést és permutációt alkalmazó rendszerek (AES):
  - a titkosítás több körből áll,
  - az eredeti kulcs alapján különböző kör-kulcsokat (round key) hoznak létre, amelyeket a megfelelő körökben használnak fel,
  - minden körben a kulcsokon egy keverést alkalmaznak,
  - a bemeneten minden körben egy helyettesítést, egy permutációt (S-box alkalmazása) és egy lineáris transzformációt hajtanak végre,

Blokk-titkosító rendszerek

- egy blokk-titkosító egyszerre n darab byte-ot titkosít, azonban ha a nyílt szöveg nagyobb mint n-byte, akkor a nyílt-szöveget fel kell osztani n-byteos részekre,
- a nyílt-szöveget "padding"-olni (kiegészíteni) kell: `PKCS#7`, ha `N` darab bájtot kell hozzáadni, akkor az `N` értékét adjuk hozzá `N`-szer
- a kulcsok fehérítése (whitening): az első kör előtt és az utolsó kör után XOR műveletre kerül sor
  - a blokktitkosító alkalmazása előtt a nyílt szöveg blokkját XOR-olják egy kulccsal, a blokktitkosítás után kapott rejtjelezett blokkot megint XOR-olják egy másik kulccsal:

$$\text{titkosítás: } c = E_{key}(k_1, k_2, m) = E_{key}(m \oplus k_1) \oplus k_2$$
$$\text{visszafejtés: } m = E^{-1}_{key}(k_1, k_2, c) = E^{-1}_{key}(c \oplus k_2) \oplus k_1$$

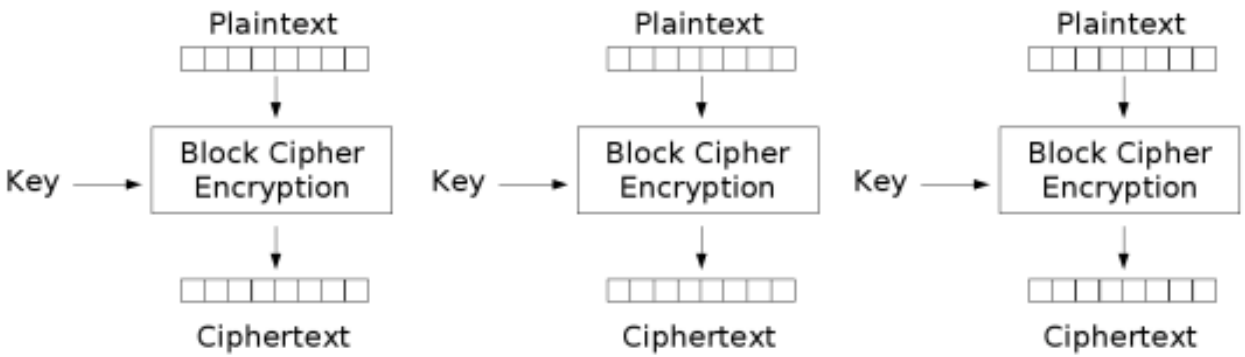
  - a blokk titkosító kevésbé lesz támadható brute-force-al: a k_1, k_2 alkalmazásával megnőtt a lehetséges kulcsok száma
- megjelenik a kulcs többszöri felhasználásának problémája $\rightarrow$ blokk titkosítási módokat kell alkalmazni
- 1981-ben a NIST standard blokk titkosítási módokat dolgozott ki, az elején a DES számára: *Block cipher mode*,
- később további blokk titkosítási módok jelentek meg: 2001-ben adják hozzá a CTR-t az addigi standard blokk titkosítási módokhoz,

Blokk-titkosító módok, jellemzők

Titkosítási mód	Alkalmazhatóság
ECB	csak akkor biztonságos, ha csak egy blokk titkosítására használjuk, pl. kulcs titkosítására
CBC	leggyakrabban használt, hitelesítést is biztosít, a titkosítást nem, de a visszafejtést lehet párhuzamosítani, egy hibás blokk az összes utána következőt elrontja
CFB	folyam-titkosító, hitelesítést is biztosít, nincs padding a titkosítást nem, de a visszafejtést lehet párhuzamosítani, egy hibás blokk az összes utána következőt elrontja
OFB	folyam-titkosító, hibajavító kódok esetében használnak, nincs padding, nem lehet párhuzamosítani, előkészíthető a titkosítás/visszafejtés
CTR	általános blokk-titkosító, nem biztosít hitelesítést, nincs padding, nagy gyorsaságú, párhuzamosítható, előkészíthető a titkosítás/visszafejtés, nem egyszerű a küldő és fogadó oldalon szinkronizálni a számlálót egy hibás blokk nem csak az aktuális blokkot rontja el

ECB (Electronic Code Book)

- a nyílt szöveg blokkjait egymástól függetlenül titkosítjuk, a titkosított szöveg a titkosított blokkok konkatenációja lesz, visszafejtésnél ugyanígy járunk el,
- csak egy blokkból álló adathalmaz esetén szabad használlni, pl. kulcsok,
- Biztonsági problémák:
  - ha két blokk egyforma, akkor ezeknek a rejtjelezett értéke is egyforma lesz
  - egy harmadik fél (ellenség) felcserélhet két rejtjelezett blokkot, vagy kitörölhet, kicserélhet egy-egy blokkot, anélkül, hogy a kommunikáló felek ezt észrevennék



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

Az ECB mód, a Crypto++ könyvtárcsomag

1. feladat

Titkosítsunk és fejtsünk vissza DES3-ECB blokk-titkosítóval egy tetszőleges bmp állományt, majd a rejtjelezett `cryptECB.bmp` állomány első 32 bájtját cseréljük le az eredeti állomány első 32 bájtjára. Figyeljük meg, hogy az újonnan kapott `cryptECB_.bmp` állományban azzal, hogy az állomány tulajdonképpeni header-jét nem rejtjeleztük következtetni lehet az eredeti tartalomra: nyíssuk meg a `cryptECB_.bmp`-t egy képszerkesztővel.

A feladatot C++-ban a crypto++ konyvtár függvényeit használva oldjuk meg:

- crypto++: kriptográfiai algoritmusok C++-ban, a NIST által elfogadott standard implementációkkal,

  http://www.cryptopp.com/#download

- használatához MSDN alatt a következőket kell tenni:
  - letöltés, kicsomagolás,
  - a `cryptest.sln` projekt megnyitása,
  - build-elés után létrejön a `\cryptopp\Win32\Output\Debug` mappába: ⇒ `cryptlib.lib`, ezt kell hozzácsatolni azokhoz a projektekhez, ahol a crypto++ függvényeket akarom használni.

A Crypto++ könyvtárcsomag

A projektnél, ahol használni akarom a crypto++ függvényeit a következő beállításokat kell elvégezni:

- `Project/Properties/Configuration Properties/C/C++/Code generation/Runtime Library` ⇒ Multi-threaded Debug (/MTd)-ra állítani,
- `Project/Properties/Configuration Properties/C/C++/General /Additional Include Directories`-nél adjuk meg a header állományok elérési útvonalát: `..\cryptopp563`
- a `cryptlib.lib`-et hozzá kell adni a projekthez,
- az alkalmazás elejére be kell írni:

```
#define CRYPTOPP_DEFAULT_NO_DLL
#include "dll.h"
```

3DES ECB módban, NEM BIZTONSÁGOS

```
#define CRYPTOPP_DEFAULT_NO_DLL
#include "dll.h"
#include "des.h"
#include <fstream>
USING_NAMESPACE(CryptoPP)
USING_NAMESPACE(std)

void DES3_ECB_Encrypt(SecByteBlock, const char *, const char *);
void DES3_ECB_Decrypt(SecByteBlock, const char *, const char *);

int main()
{
    // a key ertékét véletlenszerűen generáljuk
    AutoSeededRandomPool rnd;
    SecByteBlock key(0x00, DES_EDE2::DEFAULT_KEYLENGTH);
    rnd.GenerateBlock(key, key.size());

    DES3_ECB_Encrypt(key, "kep.bmp", "cryptECB.bmp")
    DES3_ECB_Decrypt(key, "cryptECB.bmp", "kepNew.bmp");
    return 0;
}
```

3DES ECB módban, NEM BIZTONSÁGOS

```
void DES3_ECB_Encrypt(SecByteBlock key,
        const char *infile, const char *outfile)
{
        ECB_Mode<DES_EDE2>::Encryption bf;
        bf.SetKey(key, key.size());
        FileSource(infile, true,
        new StreamTransformationFilter(bf, new FileSink(outfile)));
}

void DES3_ECB_Decrypt(SecByteBlock key,
        const char *infile, const char *outfile)
{
        ECB_Mode<DES_EDE2>::Decryption bf;
        bf.SetKey(key, key.size());
        FileSource(infile, true,
        new StreamTransformationFilter(bf, new FileSink(outfile)));
}
```

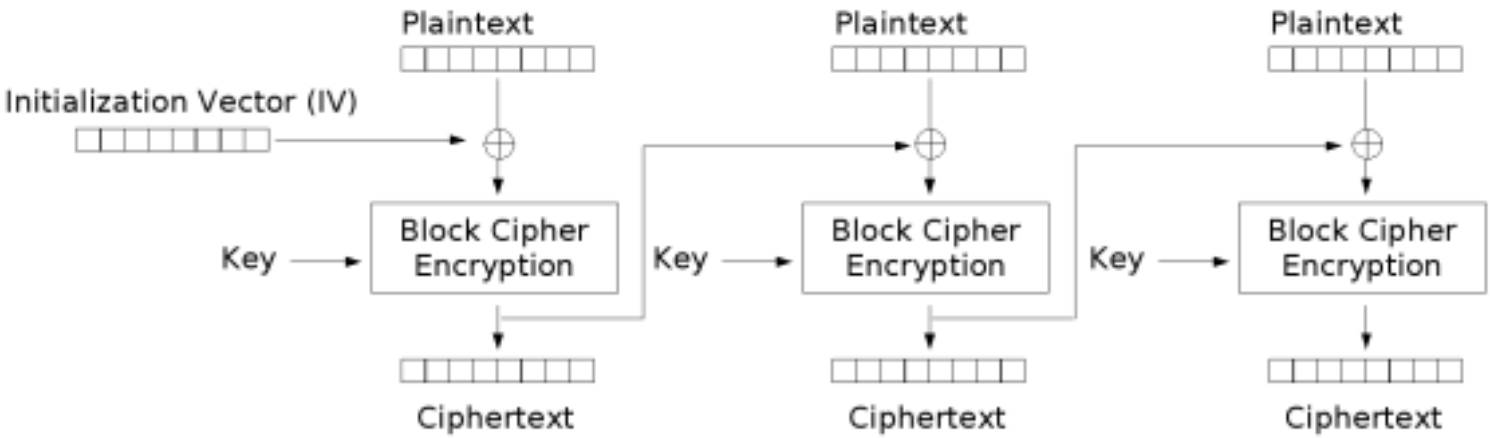
3DES ECB módban, NEM BIZTONSÁGOS

Ha a titkosított állományban lecseréljük az első 32 bájtot, egy bmp első 32 standard bájtjára, akkor az ECB mód szerint titkosított kép felismerhető:

```
int main() {
        int d = 32;
        ifstream in1("kep.bmp", ifstream::binary);
        char *mbyte = new char[d];
        in1.read(mbyte, d);
        in1.close();

        ifstream in("cryptECB.bmp", ifstream::binary)
        in.seekg(0, ios::end);
        int fsize = in.tellg();
        in.seekg(0, ios::beg);

        char *byte = new char[fsize];
        in.read(byte, fsize);
        in.close();

        ofstream out("cryptECB_.bmp", ofstream::binary)

        for (int i = 0; i < d; ++i)
                byte[i] = mbyte[i];
        out.write(byte, fsize);
        out.close();
        return 0;
}
```

CBC (Cipher-Block Chaining)

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



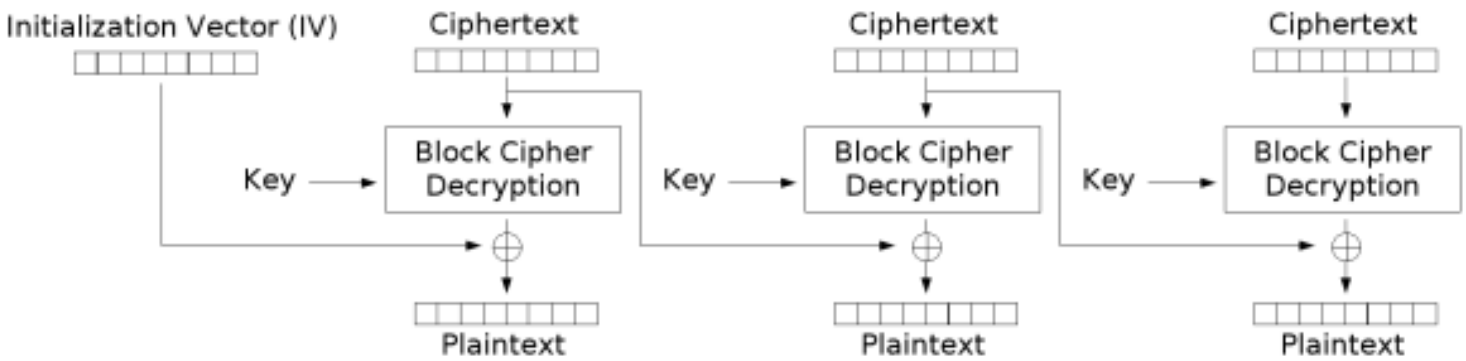
$$C_i = E_{key}(M_i \oplus C_{i-1}), C_0 = IV$$

CBC (Cipher-Block Chaining)

- a leggyakrabban használt blokk-titkosító mód
- egy blokk titkosítása az előző blokk titkosított értékétől függ,
- az első blokk esetében egy inicializáló vektort IV-t, használunk, amit nem muszáj titokban tartani
- a IV lehet
  - nonce (number used once): egy idő-pecsét, egy számláló,
  - álvéletlen módon generált byte-tömb,
- ha a nyílt szöveg egy bitje megváltozik (pl. sérül az adattovábbítás során), akkor a megfelelő rejtjelezett blokk értéke és az utána következő még két rejtjelezett blokkérték meg fog változni, emiatt alkalmazzák a mobil kommunikációban inkább a stream cipher-eket
- általános cél esetében, nagy adathalmaz titkosítására használják, illetve üzenet hitelesítő kódok (MAC) szerkesztése során

CBC (Cipher-Block Chaining)

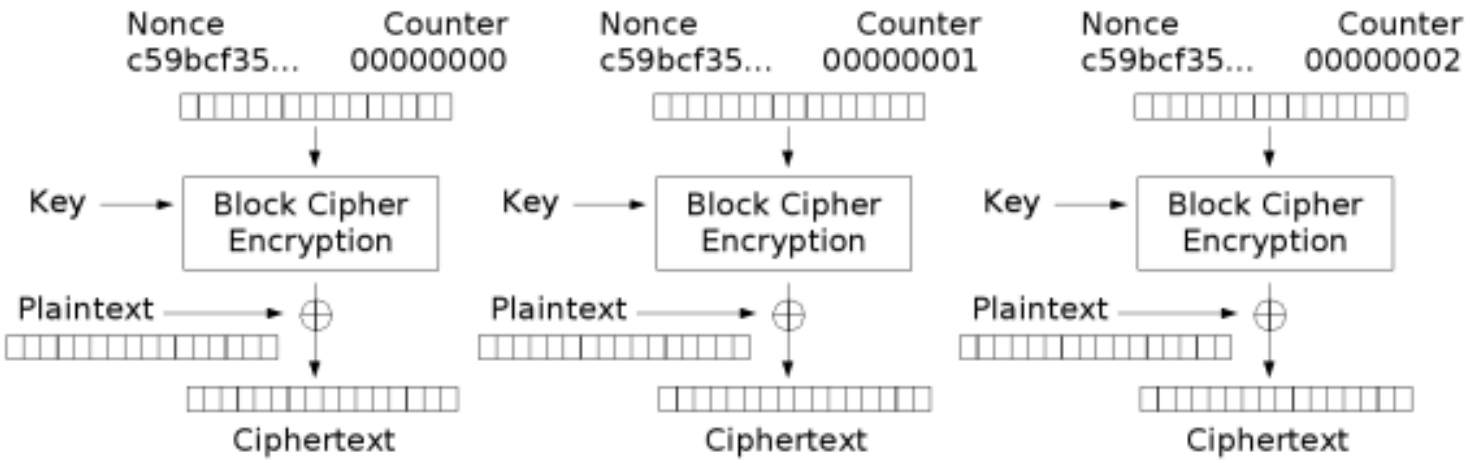
- visszafejtés: a rejtjelezett blokk-tömböt átadjuk a titkosító algoritmusnak, majd az eredmény és az előző rejtjelezett blokk $\oplus$ értékét határozzuk meg,



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

CTR (Counter mode)

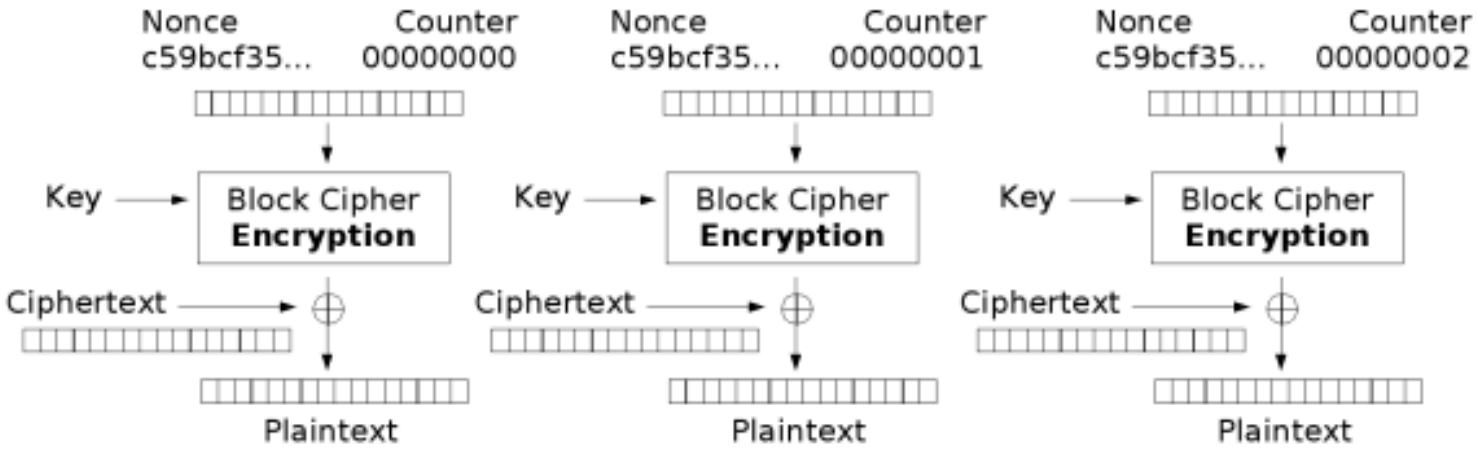
- egy folyam-titkosítást hajt végre,
- titkosítás: egy kulcs-folyamot generál, a nyílt-szövegtől függetlenül, egy számláló érték alapján, mely blokkról blokkra változik. A kapott kulcs-folyamot és a nyílt-szöveg blokkjait $\oplus$-al összeadja.



$$C_i = K_i \oplus M_i, K_i = E_{key}(Nonce, Counter_i)$$

CTR (Counter mode)

- visszafejtés: mivel az $\oplus$ művelet szimmetrikus, a titkosítás és visszafejtés ugyanaz,



$$M_i = K_i \oplus C_i, K_i = E_{key}(Nonce, Counter_i)$$

DES (Data Encryption Standard)

- 1970-ben tervezte Horst Feistel $\rightarrow$ Feistel séma,
- 1976-ban fogadták el standardként,
- 1997-ben feltörték, az összes kulcs kipróbálásának módszerével,
- kulcs-mérete: 56 bit,
- blokk-mérete: 64 bit,
- a 3DES hatékonysága: 13 Mb/sec,
- nagyon sikeres titkosító rendszer: alkalmazták a banki szektorban integritás vizsgálatok során, alkalmazták kereskedelmi társaságok, sokáig a web egyik alap titkosító mechanizmusa volt.

DES, tervezési szempontok

- Blokk-méret: nagyobb blokk-méret esetén csökken a rejtjelezési sebesség, de nagyobb a biztonság,
- Kulcs méret: nagyobb kulcs méret esetén is csökken a rejtjelezési sebesség, de nagyobb a biztonság is,
- Mennyi legyen a körök száma?
- Hogy generáljuk a körkulcsokat?, nagy komplexitás
- Hogyan értelmezzük a titkosító függvényt?, nagy komlexitás
- A visszafejtés ugyanaz, mint a titkosítás, csak fordított sorrendben kell alkalmazni a kulcsokat $\rightarrow$ nem kell két algoritmust írni.

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 nem publikus az a kritériumrendszer ami alapján az S-boxokat szerkesztették $\rightarrow$ az a vélemény, hogy ha ez nilvános lenne, akkor könnyebb lenne a kriptoanalízis.

DES, biztonság

- lavina effektus a titkosító és kulcsgeneráló algoritmusnál: kis változtatás a bemeneten nagy változtatást eredményez a kimeneten,
- a biztonság fokozása: előbb tömörítem a nyílt-szöveget utána titkosítom,
- 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 DES biztonságának a megerősítése anélkül, hogy a belső szerkezetén változtatnánk, biztonságosabb, de háromszor lassúbb, mint a szimpla DES.

DES, Kulcsgenerálás

- iterációs szerkesztési séma, alkalmazzák az AES-128 -nál is,
- a DES-nél az iteráció száma 16, a 3DES-nél az iteráció száma 48.
- Az 56 bites kulcsot kiegészítve minden 7 bit után egy paritás-bitel egy 64 bites key_0 kezdeti kulcsot kapunk,
- Példa: legyen az 56 bites kulcs *key*: $0x12, 0x69, 0x5b, 0xc9, 0xb7, 0xb7, 0xf8$, amelynek a bináris alakja a következő:
  00010010 01101001 01011011 11001001 10110111 10110111 11111000
- 7-vel felosztva kapjuk:
  0001001 0011010 0101011 0111100 1001101 1011110 1101111 1111000
- a *key*-t paritásbittekkel kiegészítve kapjuk key_0-t:

00010011 00110100 01010111 01111001 10011011 10111100 11011111 11110001

0	0	0	1	0	0	1	1
0	0	1	1	0	1	0	0
0	1	0	1	0	1	1	1
0	1	1	1	1	0	0	1
1	0	0	1	1	0	1	1
1	0	1	1	1	1	0	0
1	1	0	1	1	1	1	1
1	1	1	1	0	0	0	1

DES, Kulcsgenerálás

- a kezdeti kulcs alapján 16 körkulcsot generálunk: $key_1, \ldots, key_{16}$.
- key_0-ra alkalmazzuk a PC_1 függvényt $\rightarrow$ 56 bit, melyet felosztunk két 28 bites blokkra $\rightarrow C_{i-1}, D_{i-1}$,
- minden $i = 1, 2, \ldots 16$-re elvégezzük:
  - C_i: v_i db balra történő bitrotáció C_{i-1}-en,
  - D_i: v_i db balra történő bitrotáció D_{i-1}-en, ahol

$$v_i = \begin{cases} 1, & \text{ha } i = 1,\ 2,\ 9,\ 16 \\ 2, & \text{egyébként,} \end{cases}$$

  - alkalmazzuk a PC_2 függvényt (C_i, D_i)-re $\rightarrow key_i$.

DES, PC-1, PC-2 függvény

A PC_1 függvény

C_{i-1}						
57	49	41	33	25	17	9
1	58	50	42	34	26	18
10	2	59	51	43	35	27
19	11	3	60	52	44	36
D_{i-1}						
63	55	47	39	31	23	15
7	62	54	46	38	30	22
14	6	61	53	45	37	29
21	13	5	28	20	12	4

A PC_2 függvény

14	17	11	24	1	5
3	28	15	6	21	10
23	19	12	4	26	8
16	7	27	20	13	2
41	52	31	37	47	55
30	40	51	45	33	48
44	49	39	56	34	53
46	42	50	36	29	32

- $PC_1(b_1 b_2 \dots b_{64}) = C_{i-1} || D_{i-1} = b_{57} b_{49} \dots b_{36} b_{63} \dots b_4$, kimenete 56 bit,
- $PC_2(b_1 b_2 \dots b_{56}) = b_{14} b_{17} \dots b_{32}$, kimenete 48 bit.

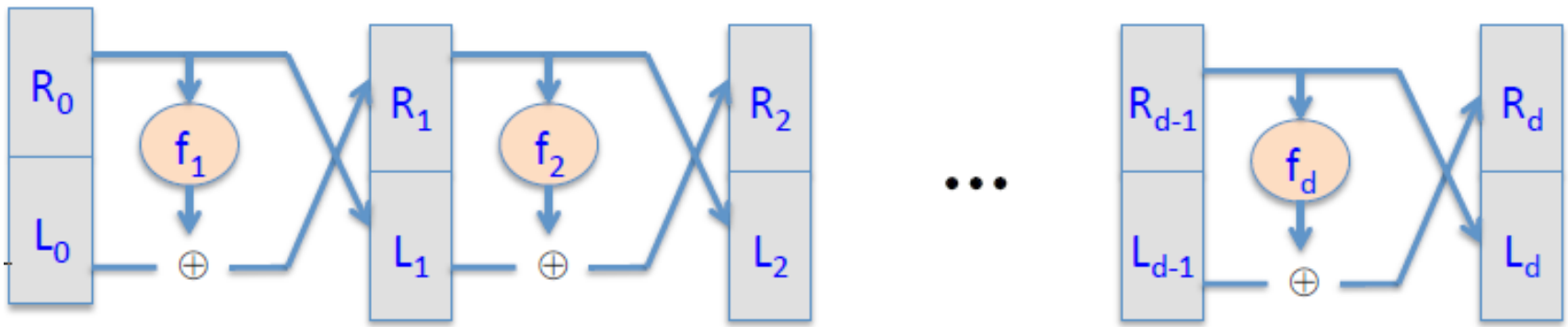
00010011 00110100 01010111 01111001 10011011 10111100 11011111 11110001
a PC_1 függvény alkalmazása után:
1111000 0110011 0010101 0101111 0101010 1011001 1001111 0001111

DES, titkosítás

- A 64 bites bemeneten alkalmazunk egy kezdeti IP-permutációt,
- A kapott bitsort 2 részre osztjukt: L_0, R_0,
- meghatározzuk L_i-t és R_i-t, $(i = 1, 2, \ldots, 16)$: $\begin{array}{ll} L_i & = R_{i-1}, \\ R_i & = L_{i-1} \oplus f_i(R_{i-1}), \end{array}$

  $f_i \quad : \{0,1\}^{32} \to \{0,1\}^{32} \quad f_i(x) \quad = F(key_i, x).$
- $L_{16}||R_{16}$-ra alkalmazzuk az IP^{-1}-permutációt.

DES, titkosítás



Forrás: Boneh D.: Introduction to Cryptography. Online Cryptography.

Titkosítás: $L_i = R_{i-1}, \quad R_i = L_{i-1} \oplus f_i(R_{i-1}),$

DES, permutáció táblák

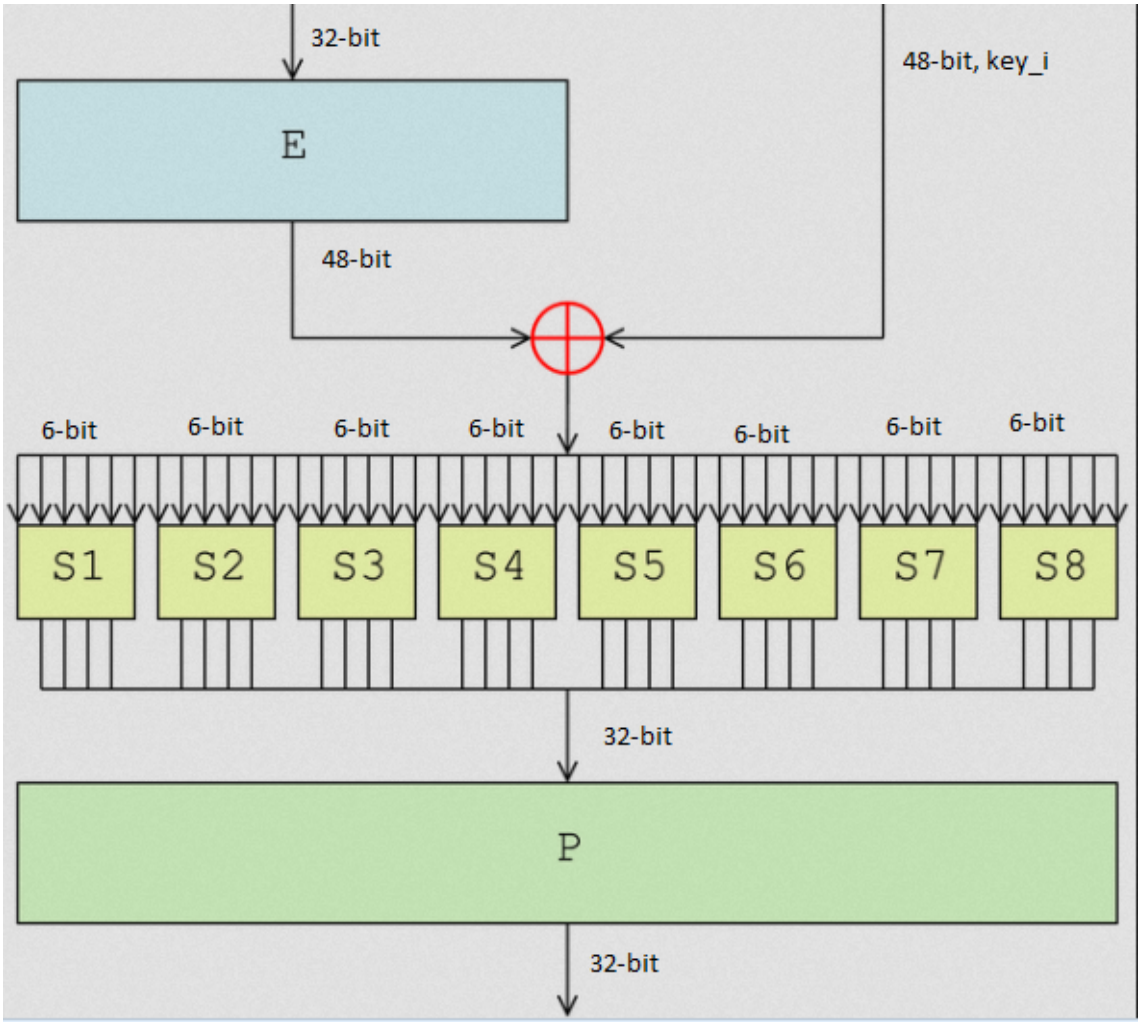
Az IP permutáció

58	50	42	34	26	18	10	2
60	52	44	36	28	20	12	4
62	54	46	38	30	22	14	6
64	56	48	40	32	24	16	8
57	49	41	33	25	17	9	1
59	51	43	35	27	19	11	3
61	53	45	37	29	21	13	5
63	55	47	39	31	23	15	7

Az IP^{-1} permutáció

40	8	48	16	56	24	64	32
39	7	47	15	55	23	63	31
38	6	46	14	54	22	62	30
37	5	45	13	53	21	61	29
36	4	44	12	52	20	60	28
35	3	43	11	51	19	59	27
34	2	42	10	50	18	58	26
33	1	41	9	49	17	57	25

DES, $f_i(x) = F(key_i, x)$ függvény



Forrás: *en.wikipedia.org/wiki/Data_Encryption_Standard*

DES, permutáció táblák

Az E expansion függvény

32	1	2	3	4	5
4	5	6	7	8	9
8	9	10	11	12	13
12	13	14	15	16	17
16	17	18	19	20	21
20	21	22	23	24	25
24	25	26	27	28	29
28	29	30	31	32	1

A P permutáció

16	7	20	21
29	12	28	17
1	15	23	26
5	18	31	10
2	8	24	14
32	27	3	9
19	13	30	6
22	11	4	25

DES, S-boxok

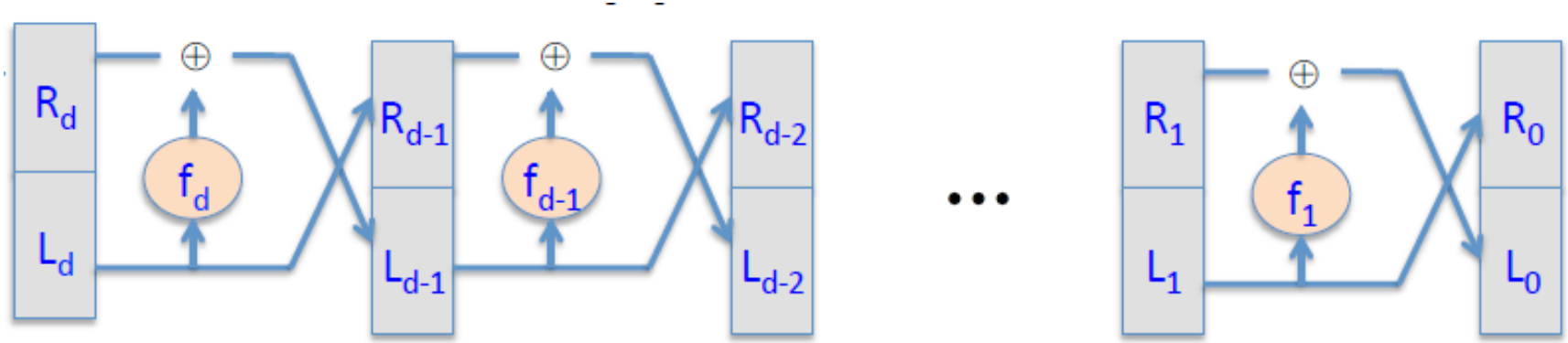
S_1															
14	4	13	1	2	15	11	8	3	10	6	12	5	9	0	7
0	15	7	4	14	2	13	1	10	6	12	11	9	5	3	8
4	1	14	8	13	6	2	11	15	12	9	7	3	10	5	0
15	12	8	2	4	9	1	7	5	11	3	14	10	0	6	13
S_2															
15	1	8	14	6	11	3	4	9	7	2	13	12	0	5	10
3	13	4	7	15	2	8	14	12	0	1	10	6	9	11	5
0	14	7	11	10	4	13	1	5	8	12	6	9	3	2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

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

- $b_1 b_6$ meghatározza a sor-indexet,
- $b_2 b_3 b_4 b_5$ meghatározza az oszlop-indexet,
- $S_i(B)$, a megfelelő indexnél levő érték bináris alakja,
- Pl. $S_1(111100) = 0101$, mert $b_1 b_6 = 10$, azaz a 2. sor, $b_2 b_3 b_4 b_5 = 1110$, azaz a 14. oszlop. Az itt található érték 5, aminek bináris alakja 0101.

DES, visszafejtés

- Visszafejtés: az alkalmazott körkulcsokat fordított sorrendben kell venni,



Forrás: Boneh D.: Introduction to Cryptography. Online Cryptography.

$$\text{Visszafejtés: } R_{i-1} = L_i, \quad L_{i-1} = R_i \oplus f_i(L_i).$$

3DES, triple DES

- a DES biztonságának a megerősítése anélkül, hogy a belső szerkezetén változtatnánk.
- Ha $E : K \times M \to M$ egy blokk-titkosító, akkor értelmezhető $3E : K^3 \times M \to M$ blokk-titkosító, ahol $3E((key_1, key_2, key_3), m) = E(key_1, D(key_2, E(key_3, m)))$,
- a kulcsméret: $3 \times 56 = 168$-bit lesz,
- biztonságosabb, de háromszor lassúbb, mint a szimpla DES,
- Miért nem $3E((key_1, key_2, key_3), m) = E(key_1, E(key_2, E(key_3, m)))$ szerint van értelmezve a $3DES$?
- ha $key_1 = key_2 = key_3$, akkor a szimpla DES-t kapom, a hardver implementációnál fel lehet használni a 3DES-t a szimpla DES implementációjára
- a feltőréshez szükséges idő, az összes kulcs kipróbálásának módszerével 2^{168},
- létezik 2^{118} nagyságrendű feltörési algoritmus.

2DES, double DES

- a $2DES$ nem növeli a titkosító biztonságát,
- $2DES : K^2 \times M \to M$, ahol $2E((key_1, key_2), m) = E(key_1, E(key_2, m))$
- meet in the middle típusú támadás alkalmazható, amely gyorsabb, mint az összes kulcs kipróbálásának módszere:
  - a cél megtalálni key_1, key_2-t, tudva, hogy m rejtjelezett értéke c, amikor is fennáll: $E(key_1, E(key_2, m)) = c$,
  - a megoldandó egyenlet ekvivalens: $E(key_2, m) = D(key_1, c)$
  - minden key lehetséges értékére meghatározzák $E(key, m)$-t, és a kapott értékeket rendezik,
  - ellenőrzik, hogy melyik key-re kapják meg $D(key_1, c)$-t; ahol találat van, ott meghatározzák a key_1, key_2-t.
  - feltöréshez szükséges idő = a rendezéshez + kereséshez szükséges idő: $2^{56} \cdot log(2^{56}) + 2^{56} \cdot log(2^{56}) < 2^{63}$,
  - feltöréshez szükséges tárhely: 2^{56}.

DESX

- Ha $E : K \times M \to M$ egy blokk-titkosító, akkor értelmezhető $EX : K^3 \times M \to M$ blokk-titkosító, ahol $EX((key_1, key_2, key_3), m) = key_{\oplus} E(key_2, key_3 \oplus m)))$,
- nem standardizált, de kivédi az összes kulcs kipróbálásának módszerét,
- ha csak az egyszer alkalmazom a $\oplus$ műveletet, azaz a $key_1 \oplus E(key_2, m)$ vagy $E(key_2, m \oplus key_1)$ konstrukciók nem jelentenek nagyobb biztonságot mint a szimpla DES,
- a blokk titkosító algoritmusok esetében számos támadási típus létezik. amelyek kivédésére a legjobb módszer, ha nem a saját implementációinkat használjuk, hanem a nyílt forráskódú könyvtárakban található implementációkkal dolgozunk, pl. OpenSSL, Crypto++.

DES, egyéb támadási módok

- differential cryptoanalysis (diferenciál kriptoanalízis)
  - sok titkos/rejtjelezett pár ismeretében gyakran az összes kulcs kipróbálásának módszerénél hatékonyabb feltörés végezhető,
- side-channel típusú támadás
  - az implementáció adta gyengeségeket használják ki,
  - meghatározzuk a titkosítás/visszafejtés időigényét,
  - meghatározzuk az áramfogyasztást nagyságát.

TEA (Tiny Encryption Algorithm)

- 1994-ben publikálta a Cambridge Egyetem két tanára, Roger Needham és Daviv Wheeler
- tervezésekor feladtak egy keveset a biztonságból, hogy a rendszer könnyen implementálható, hatékony legyen
- Feistel típusú titkosító, a titkosító és visszafejtő algoritmusok különböznek
- könnyen implementálható hardver-re, illetve softver-re
- 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, ezek alapján létrehozza az L, illetve R értékeket: a 8 bájtos szekvenciát 256-os számrendszerbeli számjegyeknek tekinti és ezekből létrehoz egy tízes számrendszerbeli számot
- alkalmaz egy konstanst: $delta = 0x9e3779b9 = 2654435769$, azaz *delta* a $2^{32}/phi$ osztási egész része, ahol $phi = 1.6180339887$ az aranyarány.
- ha két üzenetet olyan kulcsokkal titkosítunk, amelyek kapcsolódnak egymáshoz, akkor kivitelezhető a "related key" típusú támadás
- az XTEA egy később megjelent változata, amely kiküszöböli a "related key" támadást

Related key típusú támadás: hasonló a választott nyílt szöveg támadáshoz, azzal a plusszal, hogy a támadónak lehetősége van két olyan titkosított szöveghez jutni, amelyek különböző kulccsal voltak rejtjelezve.

A Blowfish

- Bruce Schneier szerkesztette 1993-ban és szabadon felhasználhatóvá tette
- 64 bites a blokk mérete, a kulcs méret változó 32 bit-től 448 bit-ig változtatható, a alkalmazott körök száma 16
- nem találtak ellene hatékony támadást
- bonyolult a kulcsfeldolgozása, ezért ha gyakran szükség van új kulcs kigenerálására akkor nem ajánlott
- a Fiestel szerkesztési sémán alapszik:
  - a 64 bites bemeneti blokkot két részre osztja: (L_0, R_0)
  - az S-boxok kimenete függ a kulcs értékétől
  - 4 S-boxal dolgozik, mindegyik 32 bites, amelyeket kezdetben a π tizdesjegyeivel inicializál
  - a körkulcsokat egy *P-array* alapján számolja ki, amelyet kezdetben szintén a π tizdesjegyeivel inicializál
  - a *P-array* elemszáma 18, mindegyik 32 bites
  - a kulcs első 32 bitejét XOR-olja a P1-el, a következő 32 bitet a P2-vel és így tovább, ezek lesznek a körkulcsok
- *www.geeksforgeeks.org/blowfish*,

A Blowfish

- egyik alternatív változata a Twofish:
  - az AES verseny kiválasztottja volt, még 5 titkosítóval együtt
  - lassúbb volt az AES-128-nál, de gyorsabb az AES-256-nál
  - nincs levédve, bárki szabadon használhatja, korlátozás nélkül
  - egyike azon néhány titkosítónak, amelyeket az OpenPGP szabvány (RFC 4880) tartalmaz
  - a PGP (Pretty Good Privacy) is használja az emailek titkosítására
  - blokk mérete 128 bit, kulcsmérete 128, 192, 256 bit
  - bonyolult a körkulcsok meghatározása, az S-boxok értékei a kulcs értéke alapján kerülnek meghatározásra
  - alkalmazza a kulcs fehérítés technikát
- a bcrypt-et is a Blowfish titkosító alapján szerkesztették 1999-ben,
  - módosították a körkulcsok maghatározásának algoritmusát
  - jelszavak tárolására használják több Unix alapú operációs rendszerben alapértelmezett
  - egy véletlenszerű értéket használ, a *salt*-ot, hogy védett legyen a szivárvány táblán alapuló támadással szemben