1 VYBRANÉ ALGORITMY FEISTELOVHO TYPU

1.1 ÚVOD

Algoritmy **Feistelovho typu** využívajú známy princíp, ktorý umožňuje **postupnou** aplikáciou relatívne **jednoduchých** transformácií[1] (na báze **nelineárnych posuvných registrov**) vytvoriť **zložitý** kryptografický algoritmus. Tento princíp je využívaný v symetrických šifrovacích algoritmoch, pričom najznámejším algoritmom tohoto typu je algoritmus **DES** (Data Encryption Standard). Keďže tento algoritmus bol podrobne prebraný počas prednášky, po vysvetlení základného princípu opíšeme podrobnejšie novší algoritmus **Skipjack**.

1.2 ZÁKLADNÝ PRINCÍP

Základný princíp algoritmov Feistelovho typu je možné názorne vysvetliť pomocou nasledujúceho jednoduchého príkladu.

Príklad
Zvoľme dve permutácie[2]

$$f_1 = \begin{pmatrix} 1 & 2 & 3 & 4 \\ 3 & 1 & 2 & 4 \end{pmatrix} \qquad f_2 = \begin{pmatrix} 1 & 2 & 3 & 4 \\ 2 & 1 & 4 & 3 \end{pmatrix}$$

a nasledovný spôsob šifrovania. Pôvodnú 8-bitovú správu $x \in M$ *rozdeľme na dva 4-bitové bloky* $x = (m_0, m_1)$. *Potom vytvoríme novú 8-bitovú správu* $c_1 = (m_1, m_2)$, *kde*[3] $m_2 = m_0 + f_1(m_1)$ *a tento postup ešte raz zopakujeme, t.j. vytvoríme* $c_2 = (m_2, m_3)$, *kde* $m_3 = m_1 + f_2(m_2)$. *Určite* c_2 *pre* $x = (10011101)$.

Príklad
Ukážte, že dešifrovanie je možné realizovať opačným postupom, t.j. využitím vzťahov: $m_1 = m_3 + f_2(m_2)$ *a* $m_0 = m_2 + f_1(m_1)$, *pričom dešifrovaná hodnota je* $x = (m_0, m_1)$.

[1] Tento prístup má široké využitie aj v iných oblastiach. Napr. v oblasti zabezpečovacích kódov sú široko využívane tzv. zreťazené kódy (concatenated codes), ktoré z dvoch relatívne jednoduchých štandardných zabezpečovacích kódov vytvárajú výkonné zabezpečovacie kódy (napr. zreťazenie konvolučných a RS kódov, turbo kódy, ...).

[2] Zápis funkcie f_1 znamená, že prvý bit sa presunie na tretiu pozíciu, druhý bit na prvú pozíciu, tretí bit na druhú pozíciu, atď.

[3] Operácia $+$ reprezentuje sčítanie modulo 2.

Z predchádzajúcich príkladov je zrejmé, že funkcie f_1 a f_2 **nemusia byť prosté** (pretože nie je potrebné počítať ich inverzie). Všetkých možných funkcií $f: \ V_4 \to V_4$ je

$$\left(2^4\right)^{2^4} = 16^{16} \doteq 1{,}85 * 10^{19}$$

Príklad
Ukážte, že nie všetky voľby funkcií $f_1(m)$ a $f_2(m)$ sú pre šifrovanie vhodné (pomôcka: skúste $f_1(m) = 1000$ a $f_2(m) = 0001$).

Uvedené spôsoby šifrovania sú špeciálnym prípadom tzv. Feistelovho kryptosystému.

Definícia
Nech množina správ M pozostáva zo všetkých možných $2n$-tíc V_{2n}. Nech priestor kľúčov K tvoria všetky možné h-tice funkcií $k = (f_1, f_2, \ldots, f_h)$, $f_i: \ V_n \to V_n$ pre každé $i = 1, 2, \ldots, h$ a priestor zašifrovaných textov $C = V_{2n}$. Zobrazenie $T_k: \ K \times V_{2n} \to V_{2n}$ definované rekurentne vzťahmi

$$x = (m_0, m_1) \in M \tag{1.1}$$

$$m_{i+1} = m_{i-1} + f_i(m_i)$$

$$T_k(x) = (m_h, m_{h+1})$$

$$i = 1, 2, \ldots, h$$

definuje Feistelov kryptosystém.

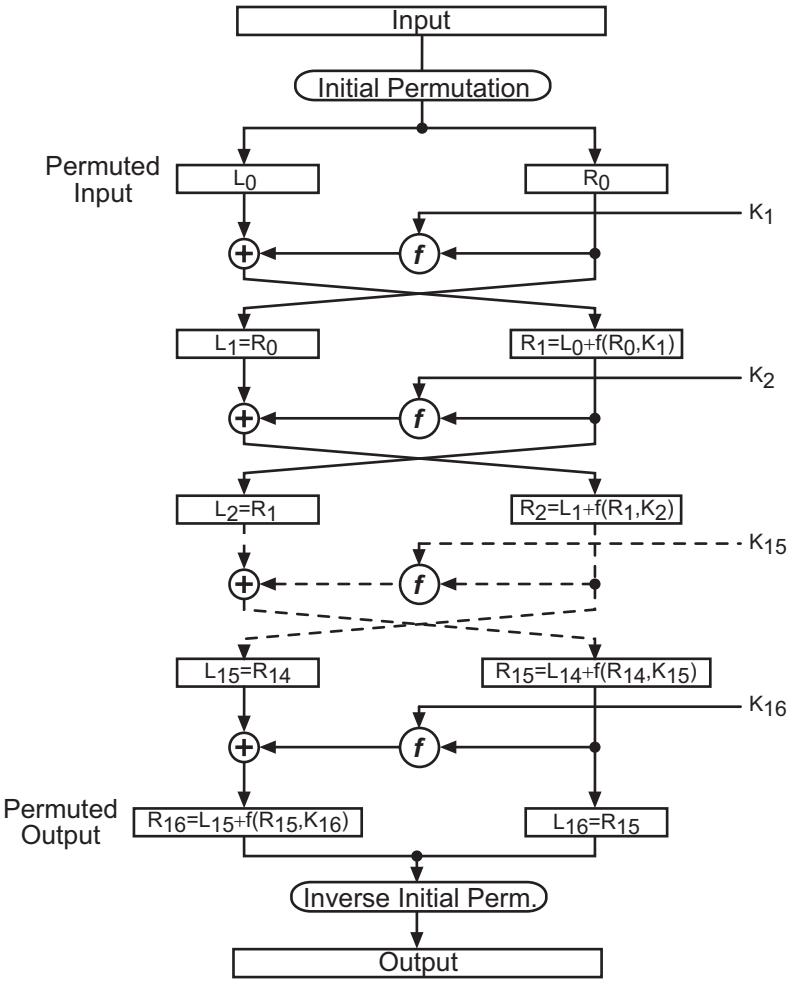
1.3 ŠIFROVACÍ ALGORITMUS DES A 3DES

Algoritmus DES bol vyvinutý organizáciou NBS (National Bureau of Standards) v polovici 70-tych rokov. V zmysle predchádzajúcej definície je dĺžka kľúča $h = 16$[4] a dĺžka bloku je $2n = 64$. Tento systém vyhovuje norme FIPS z roku 1976 (FIBS publication 46, "**Data Encryption Standard**", Federal Information Processing Standard, National Bureau of Standards, U.S. Department of Commerce, Washington D.C., January 1997) a tiež norme ANSI z roku 1980 (ANSI X3,92-1981,"**Data Encryption Algorithm**", American National Standards Institute, New York, December 31, 1980). Preto je niekedy citovaný aj ako systém **DEA**.

[4] Funkcie $(f_1, f_2, \ldots, f_{16})$ sú odvodené z 56 bitového reťazca a **kľúč má teda 56 bitov**.

DES bol v rokoch 1983, 1988, 1993 a 1999 doplňovaný a v súčasnosti (marec 2000) je najnovšia verzia štandard FIPS PUB 46-3 [4].

Využitie Feistelovej štruktúry nelineárnych posuvných registrov v algoritme DES je zrejmé z nasledujúceho obrázku.



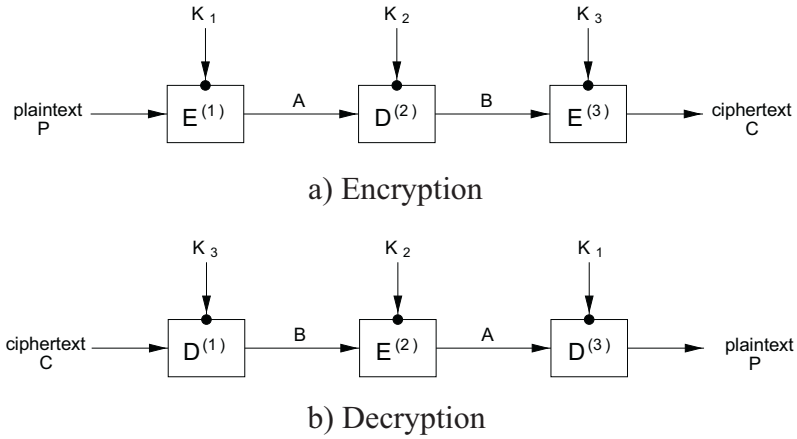
Obr.1 Feistelova štruktúra v algoritme DES

V norme [4] je špecifikované aj využitie algoritmu **3DES** (v norme označovaný ako **TDEA**), ktorý by podľa normy mal po určitú dobu existovať spoločne s algoritmom **AES**[5] (Advanced Encryption Standard) [6] a umožniť v budúcnosti postupný plynulý prechod k štandardu AES.

Základný princíp šifrovania a dešifrovania pomocou algoritmu 3DES je znázornený na nasledujúcich obrázkoch, pričom norma [4] špecifikuje tieto základné módy:

[5] AES je nový symetrický šifrovací štandard, ktorý vyberala organizácia NIST vo verejnej súťaži v rokoch 1997 až 2000. Americký NIST vybral ako nový štandard pre šifrovanie algoritmus **Rijndael** [7] (číta se [rajndol]), autormi ktorého sú belgičania Joan Daemen a Vincent Rijmen. Algoritmus Rijndael goritmus by mal nahradiť dnes už málo bezpečný algoritmus DES.

- Mód 1 – kľúče sú rôzne ($K_1 \neq K_2 \neq K_3$),
- Mód 2 – dva kľúče sú rôzne ($K_1 \neq K_2, \quad K_3 = K_1$),
- Mód 3 – kľúče sú rovnaké ($K_1 = K_2 = K_3$).



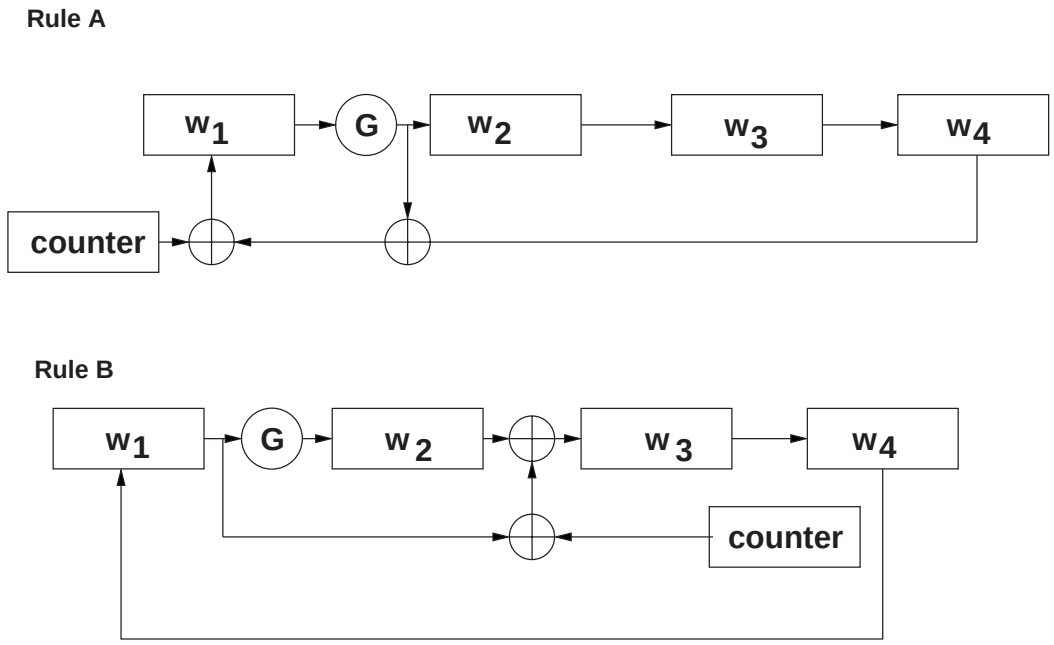
Obr.2 Princíp algoritmu 3DES: a) šifrovanie, b) dešifrovanie

1.4 ALGORITMUS SKIPJACK

23 júna 1998 ministerstvo obrany USA oznámilo, že **NSA** (National Security Agency) odtajnila algoritmy **Skipjack** (asymetrická šifra) a **KEA** (Key Exchange Algorithm – symetrická šifra). Algoritmy Skipjack a KEA mali byť pôvodne implementované len pomocou technických prostriedkov (napr. karta FORTEZA^{TM}, …) a tak ich zverejnenie bolo pre odbornú verejnosť veľkým prekvapením. Aj keď algoritmus Skipjack nebol prihlásený (a teda nebol ani kandidátom) do iniciatívy AES, na základe jeho architektúry je možné nepriamo odhadnúť, aké bezpečnostné princípy a hranice sú priateľné pre americkú tajnú službu, t.j. organizáciu, ktorá ho vytvorila a ktorá združuje najlepších matematikov, fyzikov a inžinierov. Algoritmus Skipjack je jedným z blokových šifrovacích algoritmov používaných v Internetovom protokole SSL (Secure Sockets Layer).

Detailný opis je verejne dostupný [5], pričom samozrejme existujú aj **referenčné implementácie** v jazyku C. Jedna z nich je uvedená v prílohe.

Skipjack je blokový algoritmus s **dĺžkou bloku 64** a kľúčom, ktorý má **80 bitov**. Základ algoritmu tvoria dve šifrovacie pravidlá (**Rule A**, **Rule B**) zobrazené na nasledujúcich obrázkoch.



Obr.3 Iteračné pravidlá v algoritme Skipjack

Rovnice pre šifrovanie:
Rule A:

$$w_1^{k+1} = G^k\left(w_1^k\right) \oplus w_4^k \oplus counter^k \tag{1.2}$$

$$w_2^{k+1} = G^k\left(w_1^k\right) \tag{1.3}$$

$$w_3^{k+1} = w_2^k \tag{1.4}$$

$$w_4^{k+1} = w_3^k \tag{1.5}$$

Rule B:

$$w_1^{k+1} = w_4^k \tag{1.6}$$

$$w_2^{k+1} = G^k\left(w_1^k\right) \tag{1.7}$$

$$w_3^{k+1} = w_1^k \oplus w_2^k \oplus counter^k \tag{1.8}$$

$$w_4^{k+1} = w_3^k \tag{1.9}$$

Rovnice pre dešifrovanie:
Rule inv A:

$$w_1^{k-1} = \left[G^{k-1} \right]^{-1} \left(w_2^k \right) \tag{1.10}$$

$$w_2^{k-1} = w_3^k \tag{1.11}$$

$$w_3^{k-1} = w_4^k \tag{1.12}$$

$$w_4^{k-1} = w_1^k \oplus w_2^k \oplus counter^{k-1} \tag{1.13}$$

Rule inv B:

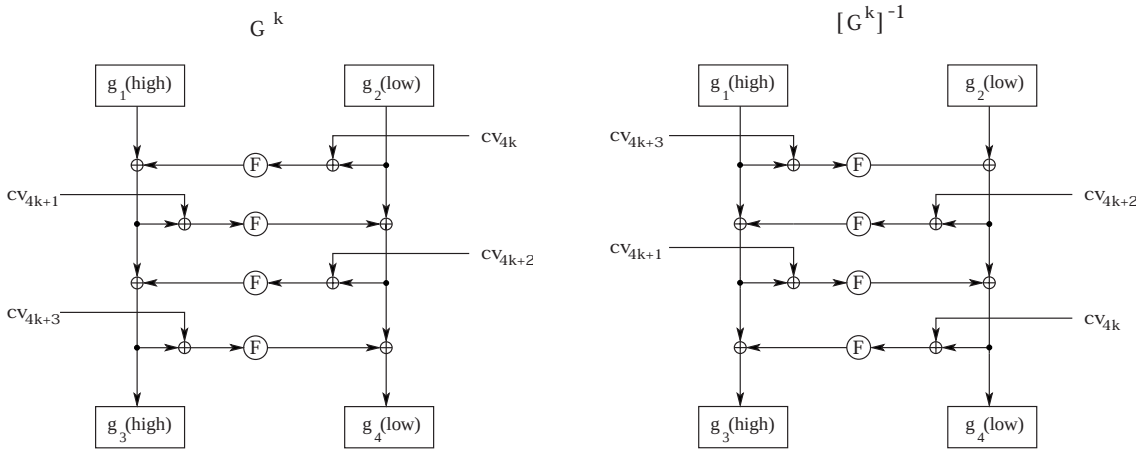
$$w_1^{k-1} = \left[G^{k-1} \right]^{-1} \left(w_2^k \right) \tag{1.14}$$

$$w_2^{k-1} = \left[G^{k-1} \right]^{-1} \left(w_2^k \right) \oplus w_3^k \oplus counter^{k-1} \tag{1.15}$$

$$w_3^{k-1} = w_4^k \tag{1.16}$$

$$w_4^{k-1} = w_1^k \tag{1.17}$$

pričom **nelineárna permutácia G** je realizovaná Feistelovou štruktúrou, ktorá je znázornená na nasledujúcom obrázku, pričom **256 bajtová tabuľka F** je v prílohe.



Obr.4 Feistelova štruktúra v algoritme Skipjack

Uvedené vzťahy, obrázky a programový kód budú **počas cvičenia** využité na **vysvetlenie činnosti** algoritmu Skipjack.

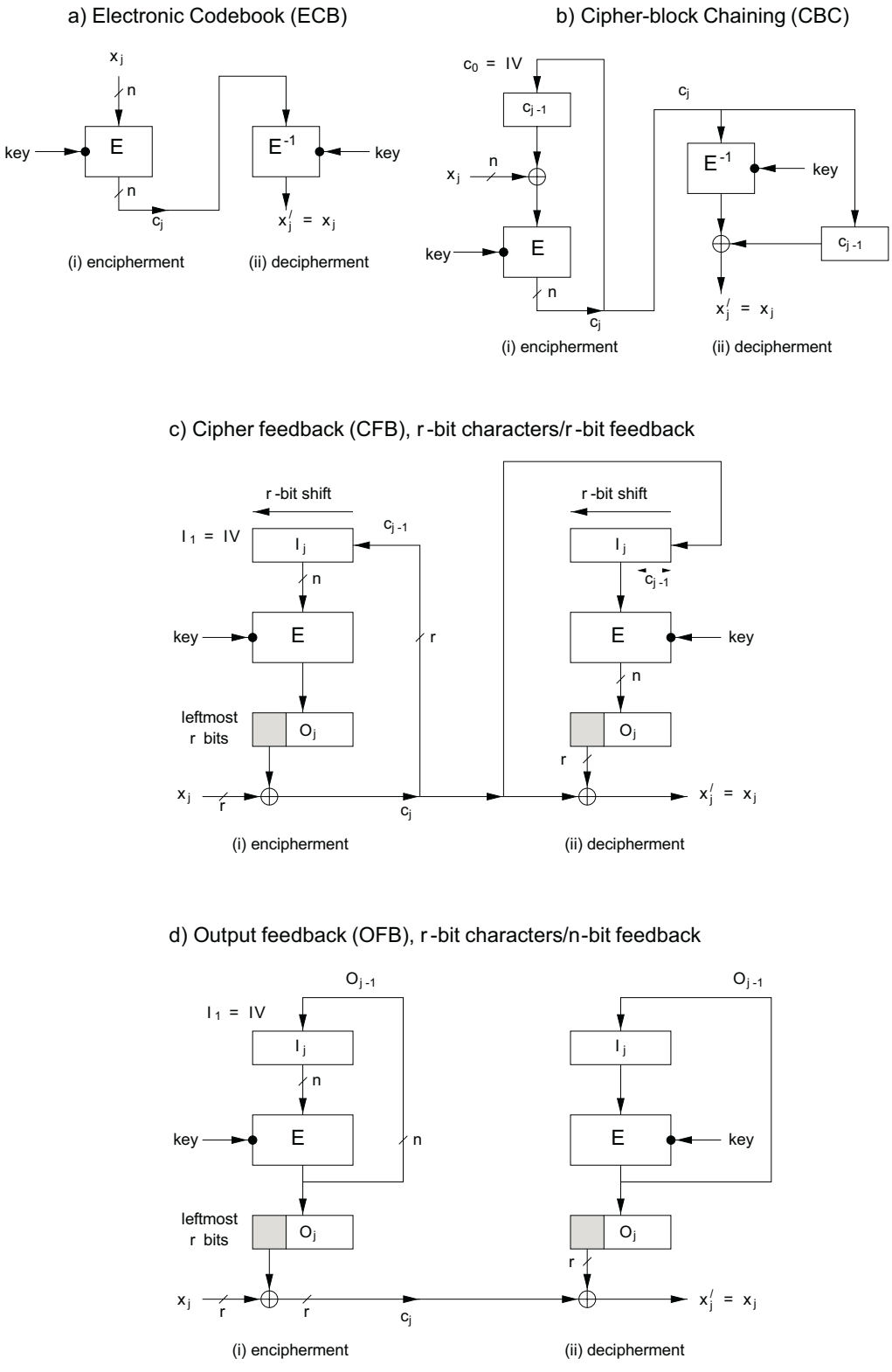
1.5 SPÔSOBY ZAPOJENIA BLOKOVÝCH ŠIFIER

Aj keď kvalita použitej blokovej šifry je jedným z najdôležitejším faktorov ovplyvňujúcich bezpečnosť celého systému, jej nevhodné použitie môže celkovú bezpečnosť systému výrazne ovplyvniť. Existuje niekoľko základných spôsobov zapojenia blokových šifier, ktoré sa v praxi používajú v závislosti na type aplikácie. Medzi základné módy patria:

- ECB mód,
- CBC mód,
- CFB mód,
- OCB mód.

pričom základné spôsoby ich zapojenia do celkového systému sú znázornené na obr.5.

Príklad
Vysvetlite princíp činnosti základných zapojení blokových šifier (ECB, CBC, CFB a OFB) znázornených na nasledujúcom obrázku a opíšte ich výhody, nevýhody a oblasť ich využitia.



Obr.5 Základné spôsoby (módy) zapojenia blokových šifier

LITERATÚRA

[1] Přibil, J. – Kodl, J.: Ochrana dat v iformatice. Vydavatelství ČVUT, Praha 1996, ISBN.

[2] Grošek, O. – Porubský, Š.: Šifrovanie – algoritmy, metódy, prax. Grada, 1992, ISBN 80-85424-62-2.

[3] Menezes, J.A. – Oorschot, P.C. – Vanstone, S.A.: Applied Cryptography. CRC Press, New York, 1997, ISBN 0-8493-8523-7.

[4] Data Encryption Standard - FIBS PUB 46-3, Federal Information Processing Standards Publications, Reaffirmed 1999 October 25. U.S. Department of Commerce/National Institute of Standards and Technology. Dostupné v elektronickej forme **– FIPS46-3.pdf**.

[5] Skipjack and KEA Algorithm Specifications. Version 2.0, 29 May 1998. Dostupné v elektronickej forme **– Skipjack_KEA.pdf**

[6] Advanced Encryption Standard, http://www.nist.gov/aes/

[7] J.Daemen, V.Rijmen, "The Rijndael Block Cipher", AES Proposal, Version 2, September 1999, http://www.nist.gov/aes. Dostupné v elektronickej forme **– Rijndael.pdf**.

PRÍLOHA

```
24 June 1998

------------------------------------------------------------------------
/*
  Skipjack, as defined in the document "Skipjack and KEA Algorithm
  dated 29 May 1998, and located at
          http://csrc.nist.gov/encryption/skipjack-1.pdf
          http://csrc.nist.gov/encryption/skipjack-2.pdf
  Note that there is no author, copyright, or other attribution on the PDF file.
  This implementation seems to successfully fit the test vectors
  listed in the documents. In fact, this version of the code simply
  prints the test vector out.
  The code is not optimized at all. It is written to be obvious, not
  necessarily to be fast.
  This code is hereby placed in the public domain. I would _like_ it
  if you gave me some credit, but you are under no obligation to do  so.
  Perry E. Metzger, Piermont Information Systems Inc., 24 June 1998
  You can reach me via email at perry@piermont.com
*/

#include <stdio.h>
#include <sys/types.h>

#define PRINTDUMP
/* #define TIMING */

typedef u_int8_t byte;
typedef u_int16_t word;
byte key[10];

const byte ftable[256] = {
/*        x0   x1   x2   x3   x4   x5   x6   x7   x8   x9   xA   xB   xC   xD   xE   xF*/
/* 0x */ 0xa3, 0xd7, 0x09, 0x83, 0xf8, 0x48, 0xf6, 0xf4, 0xb3, 0x21, 0x15, 0x78, 0x99, 0xb1, 0xaf, 0xf9,
/* 1x */ 0xe7, 0x2d, 0x4d, 0x8a, 0xce, 0x4c, 0xca, 0x2e, 0x52, 0x95, 0xd9, 0x1e, 0x4e, 0x38, 0x44, 0x28,
/* 2x */ 0x0a, 0xdf, 0x02, 0xa0, 0x17, 0xf1, 0x60, 0x68, 0x12, 0xb7, 0x7a, 0xc3, 0xe9, 0xfa, 0x3d, 0x53,
/* 3x */ 0x96, 0x84, 0x6b, 0xba, 0xf2, 0x63, 0x9a, 0x19, 0x7c, 0xae, 0xe5, 0xf5, 0xf7, 0x16, 0x6a, 0xa2,
/* 4x */ 0x39, 0xb6, 0x7b, 0x0f, 0xc1, 0x93, 0x81, 0x1b, 0xee, 0xb4, 0x1a, 0xea, 0xd0, 0x91, 0x2f, 0xb8,
/* 5x */ 0x55, 0xb9, 0xda, 0x85, 0x3f, 0x41, 0xbf, 0xe0, 0x5a, 0x58, 0x80, 0x5f, 0x66, 0x0b, 0xd8, 0x90,
/* 6x */ 0x35, 0xd5, 0xc0, 0xa7, 0x33, 0x06, 0x65, 0x69, 0x45, 0x00, 0x94, 0x56, 0x6d, 0x98, 0x9b, 0x76,
/* 7x */ 0x97, 0xfc, 0xb2, 0xc2, 0xb0, 0xfe, 0xdb, 0x20, 0xe1, 0xeb, 0xd6, 0xe4, 0xdd, 0x47, 0x4a, 0x1d,
/* 8x */ 0x42, 0xed, 0x9e, 0x6e, 0x49, 0x3c, 0xcd, 0x43, 0x27, 0xd2, 0x07, 0xd4, 0xde, 0xc7, 0x67, 0x18,
/* 9x */ 0x89, 0xcb, 0x30, 0x1f, 0x8d, 0xc6, 0x8f, 0xaa, 0xc8, 0x74, 0xdc, 0xc9, 0x5d, 0x5c, 0x31, 0xa4,
/* Ax */ 0x70, 0x88, 0x61, 0x2c, 0x9f, 0x0d, 0x2b, 0x87, 0x50, 0x82, 0x54, 0x64, 0x26, 0x7d, 0x03, 0x40,
/* Bx */ 0x34, 0x4b, 0x1c, 0x73, 0xd1, 0xc4, 0xfd, 0x3b, 0xcc, 0xfb, 0x7f, 0xab, 0xe6, 0x3e, 0x5b, 0xa5,
/* Cx */ 0xad, 0x04, 0x23, 0x9c, 0x14, 0x51, 0x22, 0xf0, 0x29, 0x79, 0x71, 0x7e, 0xff, 0x8c, 0x0e, 0xe2,
/* Dx */ 0x0c, 0xef, 0xbc, 0x72, 0x75, 0x6f, 0x37, 0xa1, 0xec, 0xd3, 0x8e, 0x62, 0x8b, 0x86, 0x10, 0xe8,
/* Ex */ 0x08, 0x77, 0x11, 0xbe, 0x92, 0x4f, 0x24, 0xc5, 0x32, 0x36, 0x9d, 0xcf, 0xf3, 0xa6, 0xbb, 0xac,
/* Fx */ 0x5e, 0x6c, 0xa9, 0x13, 0x57, 0x25, 0xb5, 0xe3, 0xbd, 0xa8, 0x3a, 0x01, 0x05, 0x59, 0x2a, 0x46
};

#define HIGH(x)      (((x) >> 8) & 0xff)
#define LOW(x)       ((x) & 0xff)
#define CONCAT(h, l) ((((word)(h)) << 8) | ((word)(l)))
#define CV(x) (key[((x) % 10)])
#define f(x) (ftable[(x)])

static word g(int k, word w)
{
          byte g1, g2, g3, g4, g5, g6;
          word ret;

          g1 = HIGH(w);
          g2 = LOW(w);

          g3 = f(g2 ^ CV(4*k  )) ^ g1;
          g4 = f(g3 ^ CV(4*k+1)) ^ g2;
```

```
	g5 = f(g4 ^ CV(4*k+2)) ^ g3;
	g6 = f(g5 ^ CV(4*k+3)) ^ g4;

	ret = CONCAT(g5, g6);

	return(ret);
}

static word inv_g(int k, word w)
{
	byte g1, g2, g3, g4, g5, g6;
	word ret;

	g6 = LOW(w);
	g5 = HIGH(w);
	g4 = f(g5 ^ CV(4*k+3)) ^ g6;
	g3 = f(g4 ^ CV(4*k+2)) ^ g5;
	g2 = f(g3 ^ CV(4*k+1)) ^ g4;
	g1 = f(g2 ^ CV(4*k  )) ^ g3;

	ret = CONCAT(g1, g2);
	return(ret);
}

static void dump(int i, word w1, word w2, word w3, word w4)
{
	printf("round %2d: %4.4x %4.4x %4.4x %4.4x\n", i, w1, w2, w3, w4);
}

static void ruleA(word *w1, word *w2, word *w3, word *w4, int k)
{
	word c, t1, t2, t3, t4, tmp;

	c = k + 1;

	t1 = *w1;
	t2 = *w2;
	t3 = *w3;
	t4 = *w4;

	tmp = g(k, t1);
	*w1 = tmp ^ t4 ^ c;
	*w2 = tmp;
	*w3 = t2;
	*w4 = t3;
}

static void ruleB(word *w1, word *w2, word *w3, word *w4, int k)
{
	word c, t1, t2, t3, t4;

	c = k + 1;

	t1 = *w1;
	t2 = *w2;
	t3 = *w3;
	t4 = *w4;

	*w1 = t4;
	*w2 = g(k, t1);
	*w3 = t1 ^ t2 ^ c;
	*w4 = t3;
}

static void inv_ruleA(word *w1, word *w2, word *w3, word *w4, int k)
{
	word c, t1, t2, t3, t4;

	c = k;

	t1 = *w1;
	t2 = *w2;
	t3 = *w3;
	t4 = *w4;
```

```
        *w1 = inv_g((k-1), t2);
        *w2 = t3;
        *w3 = t4;
        *w4 = t1 ^ t2 ^ c;
}

static void inv_ruleB(word *w1, word *w2, word *w3, word *w4, int k)
{
        word c, t1, t2, t3, t4, tmp;

        c = k;

        t1 = *w1;
        t2 = *w2;
        t3 = *w3;
        t4 = *w4;

        tmp = inv_g((k-1), t2);
        *w1 = tmp;
        *w2 = tmp ^ t3 ^ c;
        *w3 = t4;
        *w4 = t1;
}

void encrypt(word *w1, word *w2, word *w3, word *w4)
{
        int i, k;

        k = 0;

        for (i = 0; i < 8; i++) {
#ifdef PRINTDUMP
                dump(k, *w1, *w2, *w3, *w4);
#endif
                ruleA(w1, w2, w3, w4, k++);
        }

        for (i = 0; i < 8; i++) {
#ifdef PRINTDUMP
                dump(k, *w1, *w2, *w3, *w4);
#endif
                ruleB(w1, w2, w3, w4, k++);
        }

        for (i = 0; i < 8; i++) {
#ifdef PRINTDUMP
                dump(k, *w1, *w2, *w3, *w4);
#endif
                ruleA(w1, w2, w3, w4, k++);
        }

        for (i = 0; i < 8; i++) {
#ifdef PRINTDUMP
                dump(k, *w1, *w2, *w3, *w4);
#endif
                ruleB(w1, w2, w3, w4, k++);
        }

}

void decrypt(word *w1, word *w2, word *w3, word *w4)
{
        int i, k;

        k = 32;

        for (i = 0; i < 8; i++) {
#ifdef PRINTDUMP
                dump(k, *w1, *w2, *w3, *w4);
#endif
                inv_ruleB(w1, w2, w3, w4, k--);
        }
```

```
        for (i = 0; i < 8; i++) {
#ifdef PRINTDUMP
                dump(k, *w1, *w2, *w3, *w4);
#endif
                inv_ruleA(w1, w2, w3, w4, k--);
        }

        for (i = 0; i < 8; i++) {
#ifdef PRINTDUMP
                dump(k, *w1, *w2, *w3, *w4);
#endif
                inv_ruleB(w1, w2, w3, w4, k--);
        }

        for (i = 0; i < 8; i++) {
#ifdef PRINTDUMP
                dump(k, *w1, *w2, *w3, *w4);
#endif
                inv_ruleA(w1, w2, w3, w4, k--);
        }

}

int main(int argc, char **argv)
{
        int i;
        word w1, w2, w3, w4;

        /* plaintext 33221100ddccbbaa */
        w1=0x3322;
        w2=0x1100;
        w3=0xddcc;
        w4=0xbbaa;

        /* key      00998877665544332211 */
        key[0] = 0x00;
        key[1] = 0x99;
        key[2] = 0x88;
        key[3] = 0x77;
        key[4] = 0x66;
        key[5] = 0x55;
        key[6] = 0x44;
        key[7] = 0x33;
        key[8] = 0x22;
        key[9] = 0x11;

#ifdef TIMING
        for (i = 0; i < 65536; i++)
#endif
                encrypt(&w1, &w2, &w3, &w4);

        dump(32, w1, w2, w3, w4);

#ifdef TIMING
        for (i = 0; i < 65536; i++)
#endif
                decrypt(&w1, &w2, &w3, &w4);

        dump(00, w1, w2, w3, w4);
}
```