

B. Appendix: Data Encryption Standard.

(B.1). ‘*Data Encryption Standard*’, også kaldet DES, er en amerikansk standard for kryptering af data. En kort beskrivelse af DES er medtaget her, fordi DES faktisk i øjeblikket har anvendelser i praksis sammen med RSA. Bemærk, at DES ikke definerer et ‘public key’ system. Hemmeligholdelse i DES afhænger af at den benyttede nøgle holdes hemmelig.

(B.2). Et væsentligt element i DES er en algoritme, ‘*Data Encryption Algorithm*’ kaldet DEA, der med en valgt nøgle k krypterer 64-bit blokke, dvs elementer i \mathbb{F}_2^{64} . Krypteringstransformationen er således en afbildning $E_k: \mathbb{F}_2^{64} \rightarrow \mathbb{F}_2^{64}$. Nøglen k fastlægger tillige den inverse transformation D_k . Standarden DES kan anvendes i forskellige versioner (‘modes’). Fælles for disse versioner er, at klarteksten opbrydes i en følge af 64-bit blokke; afhængig af hvilken ‘mode’ der anvendes produceres her ud fra en ny følge af 64-bit blokke, og hver af de fremkomne blokke krypteres med algoritmen DEA (med en fast valgt nøgle k), hvorved krypteksten fremkommer som en ny følge af 64-bit blokke.

Når DES anvende i ‘*electronic codebook mode*’ (ECB) krypteres blot hver af klartekstens blokke t_1, t_2, \dots med transformationen E_k . I ‘*cipher block chaining mode*’ (CBC) vælges en fast initialiserings vektor $v \in \mathcal{T} = \mathbb{F}_2^{64}$. Ud fra klartekstens blokke t_1, t_2, \dots produceres nu blokken $v_1 := v + t_1$, og induktivt, blokkene $v_i := v_{i-1} + t_i$. Hver blok v_i krypteres med E_k til $h_i = E_k(v_i)$. Ved dekryptering anvendes først D_k på den modtagne følge h_1, h_2, \dots . Herved fremkommer følgen v_1, v_2, \dots , og klarteksten kan nu genfindes som $t_1 = v + v_1$ og $t_i = v_i + v_{i-1}$. Bemærk en vigtig egenskab ved ‘chaining’: en nok så lille ændring/fejl i en enkelt af klartekstens blokke vil påvirke krypteringen af alle de følgende blokke.

(B.3). De mulige blokke udgør mængden $\mathcal{T} = \{0, 1\}^{64}$ med 2^{64} elementer, og svarende til en valgt nøgle k definerer algoritmen DEA altså en bijektiv afbildning $E_k: \mathcal{T} \rightarrow \mathcal{T}$. Elementerne i \mathcal{T} kan opfattes på mangfoldige måder: som tallene fra 0 til $2^{64} - 1$ (binær repræsentation), som elementerne i restklasseringen $\mathbf{Z}/2^{64}$, som vektorerne i vektorrummet \mathbb{F}_2^{64} over legemet \mathbb{F}_2 , som ord bestående af 8 tegn (8 bytes a 8 bits), osv. De bijektive afbildninger af \mathcal{T} på sig selv udgør den symmetriske gruppe af orden $(2^{64})!$. I DES er der 2^{56} mulige nøgler k , så krypteringstransformationerne E_k frembragt af algoritmen DEA udgør en forsvindende brøkdel af samtlige mulige. At nøglerummet er for lille er en afgørende indvending mod DES; det er ikke umuligt at forestille sig en dekryptering forsøgt med samtlige nøgler. Krypteringstransformationerne E_k frembragt af DEA vil blive beskrevet nærmere nedenfor.

(B.4) Beskrivelse af algoritmen. Som nævnt definerer algoritmen DEA en familie af transformationer E_k , der afhænger af en valgt nøgle k . Hver transformation er en bijektiv afbildning af mængden \mathcal{T} på sig selv. Der er 2^{56} mulige nøgler k , og hver nøgle kan naturligt opfattes som en vektor i \mathbb{F}_2^{56} . Transformationen E_k defineres som en sammensætning af 33 transformationer,

$$\mathbb{F}_2^{64} \xrightarrow{IP} \mathbb{F}_2^{64} \xrightarrow{\varphi_{K_1}} \mathbb{F}_2^{64} \xrightarrow{\tau} \mathbb{F}_2^{64} \xrightarrow{\varphi_{K_2}} \mathbb{F}_2^{64} \xrightarrow{\tau} \dots$$

$$\dots \xrightarrow{\tau} \mathbb{F}_2^{64} \xrightarrow{\varphi_{K_{15}}} \mathbb{F}_2^{64} \xrightarrow{\tau} \mathbb{F}_2^{64} \xrightarrow{\varphi_{K_{16}}} \mathbb{F}_2^{64} \xrightarrow{FP} \mathbb{F}_2^{64} .$$

Transformationen τ , der optræder 15 gange, består blot i en simpel ombytning af de to halvdele af en blok: de første 32 bits i blokken ombyttes med de sidste 32 bits. Også transformationerne (FP) ('final permutation') og IP ('initial permutation') er faste (dvs uafhængige af den valgte nøgle) og udføres ved permutationer af bit'ene i en blok, og $(FP) = (IP)^{-1}$. Endelig er φ_K en transformation, der afhænger af en vektor K i \mathbb{F}_2^{48} , og K_1, \dots, K_{16} en følge af sådanne vektorer, bestemt ud fra den givne nøgle k . Definitionen af FP og IP og φ_K , og bestemmelsen af K_i 'erne ud fra k , beskrives i det følgende.

(B.5) Transformationerne FP og IP . Den faste transformation $IP: \mathcal{T} \rightarrow \mathcal{T}$ er en permutation af de 64 bits, som bestemmer elementerne i \mathcal{T} , og FP er den inverse. De svarer altså til to permutation IP og FP af tallene $1, 2, \dots, 64$, explicit bestemt ved følgen af de 64 billeder. Følgen af de 64 billeder kan kompakt opskrives „foldet“ i en 8×8 -matrix. Det er nok nemmest at se mønsteret i FP , som er

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

Efter denne beskrivelse transformerer FP en blok af 64 bits b_i således:

$$(b_1, b_2, \dots, b_{63}, b_{64}) \mapsto (b_{40}, b_8, \dots, b_{57}, b_{25}).$$

Den tilsvarende explicite angivelse af IP , såvel som en angivelse af de øvrige permutationer, der indgår i beskrivelsen herunder, kan umiddelbart læses ud af appendixet, som indeholder en C-implementering af algoritmen.

(B.6) Transformationerne φ_K . Transformationerne $\varphi_K: \mathcal{T} \rightarrow \mathcal{T}$, der afhænger af en vektor K i \mathbb{F}_2^{48} , defineres således: Idet \mathcal{T} kan identificeres med produktet $\mathbb{F}_2^{32} \times \mathbb{F}_2^{32}$ svarer blokke t i \mathcal{T} til par $t = (L, R)$ af vektorer L, R i \mathbb{F}_2^{32} . Med denne identification defineres φ_K ved

$$(L, R) \mapsto (L + f(R, K), R),$$

hvor afbildningen $R \mapsto f(R, K)$ for hver fast vektor K i \mathbb{F}_2^{48} er en (ikke-lineær) afbildning $\mathbb{F}_2^{32} \rightarrow \mathbb{F}_2^{32}$, der vil blive beskrevet nedenfor.

Bemærk, at φ_K er involution, dvs opfylder, at φ_K^2 er den identiske afbildning. Dette følger af at additionen ovenfor foregår i et vektorrum over \mathbb{F}_2 , hvor jo $f + f = 0$. Bemærk videre, at når man i praksis vil beregne sammensætningen af de afbildninger, hvoraf E_k består, så er det bekvemt i stedet for φ_K at betragte afbildningen $\tau\varphi_K$, der jo er givet ved

$$(L, R) \mapsto (R, L + f(R, K)).$$

Transformationen E_k er så produktet af først IP , dernæst 16 afbildninger af formen $\tau\varphi_K$ for $K = K_1, \dots, K_{16}$, dernæst τ (her udnyttes, at $\tau(\tau\varphi_{K_{16}}) = \varphi_{K_{16}}$), og endelig til sidst FP .

(B.7) Afbildningerne $f(R, K)$. Afbildningerne $R \mapsto f(R, K)$, der for hver fast vektor K i \mathbb{F}_2^{48} er afbildninger $\mathbb{F}_2^{32} \rightarrow \mathbb{F}_2^{32}$, er sammensætninger af 4 afbildninger:

$$\mathbb{F}_2^{32} \xrightarrow{E} \mathbb{F}_2^{48} \xrightarrow{+K} \mathbb{F}_2^{48} \xrightarrow{S} \mathbb{F}_2^{32} \xrightarrow{P} \mathbb{F}_2^{32}.$$

Afbildningen $+K : \mathbb{F}_2^{48} \rightarrow \mathbb{F}_2^{48}$ er den bijektive afbildning bestemt ved $W \mapsto W + K$, som til vektoren W adderer vektoren K .

De øvrige afbildninger E , S og P er faste, dvs uafhængige af k . Afbildningen $E : \mathbb{F}_2^{32} \rightarrow \mathbb{F}_2^{48}$ (' E bit-selection function') en fast (injektiv, lineær) afbildning, der essentielt blot består i en dublering af 16 udvalgte bits. Mere præcist, de 48 koordinater i billedet $E(R)$ af en vektor R i \mathbb{F}_2^{32} er bestemt ved at angive følgen af koordinatnumre. Denne følge er („foldet“):

$$E = \begin{matrix} & 32 & 1 & 2 & 3 & 4 & 5 \\ & 4 & 5 & 6 & 7 & 8 & 9 \\ & 8 & 9 & 10 & 11 & 12 & 13 \\ E = & 12 & 13 & 14 & 15 & 16 & 17 \\ & 16 & 17 & 18 & 19 & 20 & 21 \\ & 20 & 21 & 22 & 23 & 24 & 25 \\ & 24 & 25 & 26 & 27 & 28 & 29 \\ & 28 & 29 & 30 & 31 & 32 & 1 \end{matrix}$$

(se også appendixet: E bit-selection table). Afbildningen E afbilder altså en vektor med 32 koordinater (bits) b_i således:

$$(b_1, b_2, \dots, b_{32}) \mapsto (b_{32}, b_1, \dots, b_4, b_5, b_4, b_5, b_6, \dots, b_{32}, b_1).$$

Den afsluttende afbildning $P : \mathbb{F}_2^{32} \rightarrow \mathbb{F}_2^{32}$ er en fast permutation af de 32 koordinater (bits) bestemt ved en explicit angivelse af den permuterede rækkefølge (foldet i en 8×4 matrix), se appendixet.

Endelig er afbildningen $S : \mathbb{F}_2^{48} \rightarrow \mathbb{F}_2^{32}$ (' S -selection function' eller ' S -boxene') en fast afbildning, der beskrives herunder.

(B.8) S -boxene. Afbildningen S er en afbildning $\mathbb{F}_2^{48} \rightarrow \mathbb{F}_2^{32}$. Idet vi identificerer $\mathbb{F}_2^{48} = (\mathbb{F}_2^6)^8$ og $\mathbb{F}_2^{32} = (\mathbb{F}_2^4)^8$, kan S beskrives ved 8 afbildninger S_1, \dots, S_8 (de såkaldt S -boxe). Hver afbildning S_i er en afbildning

$$S_i : \mathbb{F}_2^6 \rightarrow \mathbb{F}_2^4.$$

Ud fra disse afbildninger er S givet ved

$$x = (x_1, \dots, x_8) \mapsto (S_1 x_1, \dots, S_8 x_8),$$

hvor vektoren $x \in \mathbb{F}_2^{48}$ med de 8 „koordinater“ $x_i \in \mathbb{F}_2^6$ afbildes i vektoren $Sx \in \mathbb{F}_2^{32}$ med de 8 „koordinater“ $S_i x_i \in \mathbb{F}_2^4$. Hver afbildning S_i defineres ud fra en 4×16 matrix, hvis koefficienter består af tallene $0, \dots, 15$ (hvor hvert tal optræder 4 gange i matricen). Ved definitionen

af den til matricen hørende afbildning regnes række-numrene (der er 4 rækker) som elementer i \mathbb{F}_2^2 , søjlenumrene (der er 16 søjler) regnes som elementer i \mathbb{F}_2^4 , og matricens elementer (det var udelukkende tallene $0, \dots, 15$) regnes som elementer i \mathbb{F}_2^4 . Med denne konvention er afbildningen hørende til matricen defineret således: vektoren $(b_1, b_2, b_3, b_4, b_5, b_6)$ i \mathbb{F}_2^6 afbildes på det element i \mathbb{F}_2^4 , der står i matricen i række nummer (b_1, b_6) og søjle nummer b_2, b_3, b_4, b_5 .

Matricen for S_1 er følgende:

$$S_1 = \begin{matrix} & 14 & 4 & 13 & 1 & 2 & 15 & 11 & 8 & 3 & 10 & 6 & 12 & 5 & 9 & 0 & 7 \\ & 0 & 15 & 7 & 4 & 14 & 2 & 13 & 1 & 10 & 6 & 12 & 11 & 9 & 5 & 3 & 8 \\ S_1 = & 4 & 1 & 14 & 8 & 13 & 6 & 2 & 11 & 15 & 12 & 9 & 7 & 3 & 10 & 5 & 0 \\ & 15 & 12 & 8 & 2 & 4 & 9 & 1 & 7 & 5 & 11 & 3 & 14 & 10 & 0 & 6 & 13 \end{matrix}$$

For den tilhørende afbildning $S_1: \mathbb{F}_2^6 \rightarrow \mathbb{F}_2^4$ bestemmes værdien på fx vektoren $V = (110111)$ således: Første og sidste bit giver rækkenummeret $1 \cdot 2 + 1 \cdot 1 = 3$, de fire bits i midten giver søjle nummeret $1 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2 + 1 \cdot 1 = 11$. På denne plads i matricen står $14 = 1 \cdot 2^3 + 1 \cdot 2^2 + 1 \cdot 2 + 0 \cdot 1$. Følgelig er $S_1(V) = (1110)$.

De tilsvarende matricer for S_2, \dots, S_8 fremgår af appendixet.

(B.9) Frembringelse af de 16 nøgler K_i . Som nævnt kan en nøgle k opfattes som en vektor k i \mathbb{F}_2^{56} . Ud fra denne vektor frembringes de 16 delnøgler K_i i \mathbb{F}_2^{48} ved på 16 forskellige (explicit angivne) måder at udvælge og permutere 48 af de 56 bits i k . En præcis beskrivelse anføres i det følgende.

Nøglerne i DES opfattes som de vektorer k i \mathbb{F}_2^{64} , for hvilke hvert 8'ende bit er et paritetsbit (dvs har samme paritet som summen af de 7 foregående). Ud fra en sådan nøgle k i \mathbb{F}_2^{64} frembringes først med en fast afbildning $PC_1: \mathbb{F}_2^{64} \rightarrow \mathbb{F}_2^{56}$ ('permuted choice 1') en vektor $M_0 := PC_1(k)$ i \mathbb{F}_2^{56} . Essentielt er PC_1 blot en permutation af de betydende bits i k : PC_1 glemmer paritetsbit'ene, og og permuterer de øvrige bits. Idet vi identificerer $\mathbb{F}_2^{56} = \mathbb{F}_2^{28} \times \mathbb{F}_2^{28}$, kan vektorer M i \mathbb{F}_2^{56} opfattes som bestående af 2 halvdele, $M = (C, D)$, hvor C, D er vektorer i \mathbb{F}_2^{28} . Afbildningen PC_1 angives ved en rækkefølge af hvordan koordinaterne i billedet M skal udtages fra k . Denne rækkefølge, eller rettere sagt de to rækkefølger for henholdsvis C og D , fremgår af appendixet.

I bestemmelsen af K_i 'erne indgår yderligere en fast afbildning $PC_2: \mathbb{F}_2^{56} \rightarrow \mathbb{F}_2^{48}$ ('permuted choice 2'), og en følge af 16 simple permutationer af \mathbb{F}_2^{56} , de såkaldte 'left shifts'. Afbildningen PC_2 er essentielt en projektionsafbildning; den glemmer 8 af 56 koordinater (det er faktisk numrene 9, 18, 22, 25, 35, 38, 43, 54), og permuterer de 48 resterende. Den præcise bestemmelse af PC_2 fremgår af appendixet.

Hvert af de såkaldte 'left shifts' LS består af en cyklisk forskydning (mod venstre) af bits'ene med samme forskydningstal i hver af de to halvdele C, D af en vektor M i \mathbb{F}_2^{56} . De 16 'left shifts' LS_i , der anvendes har i rækkefølge forskydningstallene

$$1, 1, 2, 2, 2, 2, 2, 2, 1, 2, 2, 2, 2, 2, 2, 1.$$

Med ovenstående definitioner kan frembringelsen af nøglerne K_i defineres induktivt ud fra $M_0 = PC_1(k)$ ved

$$M_i := LS_i(M_{i-1}), \quad K_i := PC_2(M_i),$$

for $i = 1, \dots, 16$.

(B.10). Hermed er beskrivelsen af algoritmen DEA fuldført. Dekryptering, dvs bestemmelse af den inverse D_k til krypteringstransformationen E_k fra (B.4), er essentielt den samme algoritme, idet

$$E_k^{-1} = (FP) \varphi_{K_1} \tau \varphi_{K_2} \cdots \tau \varphi_{K_1} \tau \varphi_{K_{16}} (IP).$$

Dette følger af at afbildningerne φ_K , som bemærket i (B.6), og τ er involutioner. Heraf ses, at dekryptering blot foregår ved at anvende algoritmen med K_i 'erne i den omvendte rækkefølge.

```
/*
 * This program implements the
 * Proposed Federal Information Processing
 * Data Encryption Standard.
 * See Federal Register, March 17, 1975 (40FR12134)
 */

/*
 * Initial permutation,
 */
static char IP[] = {
    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,
};

/*
 * Final permutation, FP = IP^(-1)
 */
static char FP[] = {
    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,
};
```

```
/*
 * Permuted-choice 1 from the key bits
 * to yield C and D.
 * Note that bits 8,16... are left out:
 * They are intended for a parity check.
 */
static char PC1_C[] = {
    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,
};
static char PC1_D[] = {
    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,
};

/*
 * Sequence of shifts used for the key schedule.
 */
static char shifts[] = {
    1,1,2,2,2,2,2,2,1,2,2,2,2,2,2,1,
};

/*
 * Permuted-choice 2, to pick out the bits from
 * the CD array that generate the key schedule.
 */
static char PC2_C[] = {
    14,17,11,24, 1, 5,
    3,28,15, 6,21,10,
    23,19,12, 4,26, 8,
    16, 7,27,20,13, 2,
};
static char PC2_D[] = {
    41,52,31,37,47,55,
    30,40,51,45,33,48,
    44,49,39,56,34,53,
    46,42,50,36,29,32,
};

/*
 * The C and D arrays used to calculate the key schedule.
 */
static char C[28];
static char D[28];

/*
 * The key schedule.
 */
```

```
    * Generated from the key.
    */
static char    KS[16][48];

/*
 * The E bit-selection table.
 */
static char    E[48];
static char    e[] = {
    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,
};

/*
 * Set up the key schedule from the key.
 */

setkey(key)
char *key;
{
    register i, j, k;
    int t;

    /*
     * First, generate C and D by permuting
     * the key.  The low order bit of each
     * 8-bit char is not used, so C and D are only 28
     * bits apiece.
     */
    for (i=0; i<28; i++) {
        C[i] = key[PC1_C[i]-1];
        D[i] = key[PC1_D[i]-1];
    }
    /*
     * To generate Ki, rotate C and D according
     * to schedule and pick up a permutation
     * using PC2.
     */
    for (i=0; i<16; i++) {
        /*
         * rotate.
         */
        for (k=0; k<shifts[i]; k++) {
            t = C[0];
            for (j=0; j<28-1; j++)
```

```
        C[j] = C[j+1];
        C[27] = t;
        t = D[0];
        for (j=0; j<28-1; j++)
            D[j] = D[j+1];
        D[27] = t;
    }
    /*
    * get Ki. Note C and D are concatenated.
    */
    for (j=0; j<24; j++) {
        KS[i][j] = C[PC2_C[j]-1];
        KS[i][j+24] = D[PC2_D[j]-28-1];
    }
}

for(i=0;i<48;i++)
    E[i] = e[i];
}

/*
* The 8 selection functions.
* For some reason, they give a 0-origin
* index, unlike everything else.
*/
static char S[8][64] = {
    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,

    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,

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

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

    2,12, 4, 1, 7,10,11, 6, 8, 5, 3,15,13, 0,14, 9,
    14,11, 2,12, 4, 7,13, 1, 5, 0,15,10, 3, 9, 8, 6,
    4, 2, 1,11,10,13, 7, 8,15, 9,12, 5, 6, 3, 0,14,
    11, 8,12, 7, 1,14, 2,13, 6,15, 0, 9,10, 4, 5, 3,
```

```
12, 1,10,15, 9, 2, 6, 8, 0,13, 3, 4,14, 7, 5,11,
10,15, 4, 2, 7,12, 9, 5, 6, 1,13,14, 0,11, 3, 8,
 9,14,15, 5, 2, 8,12, 3, 7, 0, 4,10, 1,13,11, 6,
 4, 3, 2,12, 9, 5,15,10,11,14, 1, 7, 6, 0, 8,13,

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

13, 2, 8, 4, 6,15,11, 1,10, 9, 3,14, 5, 0,12, 7,
 1,15,13, 8,10, 3, 7, 4,12, 5, 6,11, 0,14, 9, 2,
 7,11, 4, 1, 9,12,14, 2, 0, 6,10,13,15, 3, 5, 8,
 2, 1,14, 7, 4,10, 8,13,15,12, 9, 0, 3, 5, 6,11,
};

/*
 * P is a permutation on the selected combination
 * of the current L and key.
 */

static char P[] = {
    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,
};

/*
 * The current block, divided into 2 halves.
 */
/* static char L[32], R[32]; */
static char L[64];
#define R (&L[32])
static char tempL[32];
static char f[32];

/*
 * The combination of the key and the input, before selection.
 */
static char preS[48];

/*
 * The payoff: encrypt a block.
 */
encrypt(block, edflag)
char *block;
```

```
{  
    int i, ii;  
    register t, j, k;  
    /*  
     * First, permute the bits in the input  
     */  
    for (j=0; j<64; j++)  
        L[j] = block[IP[j]-1];  
    /*  
     * Perform an encryption operation 16 times.  
     */  
    for (ii=0; ii<16; ii++) {  
        /*  
         * Only encrypt for now.  
         */  
        i = ii;  
        /*  
         * Save the R array,  
         * which will be the new L.  
         */  
        for (j=0; j<32; j++)  
            tempL[j] = R[j];  
        /*  
         * Expand R to 48 bits using the E selector;  
         * exclusive-or with the current key bits.  
         */  
        for (j=0; j<48; j++)  
            preS[j] = R[E[j]-1] ^ KS[i][j];  
        /*  
         * The pre-select bits are now considered  
         * in 8 groups of 6 bits each.  
         * The 8 selection functions map these  
         * 6-bit quantities into 4-bit quantities  
         * and the results permuted  
         * to make an f(R, K).  
         * The indexing into the selection functions  
         * is peculiar; it could be simplified by  
         * rewriting the tables.  
         */  
        for (j=0; j<8; j++) {  
            t = 6*j;  
            k = S[j][(preS[t+0]<<5)+  
                    (preS[t+1]<<3)+  
                    (preS[t+2]<<2)+  
                    (preS[t+3]<<1)+  
                    (preS[t+4]<<0)+  
                    (preS[t+5]<<4)];  
            t = 4*j;  
            f[t+0] = (k>>3)&01;  
            f[t+1] = (k>>2)&01;  
            f[t+2] = (k>>1)&01;
```

```
        f[t+3] = (k>>0)&01;
    }
    /*
    * The new R is L ^ f(R, K).
    * The f here has to be permuted first, though.
    */
    for (j=0; j<32; j++)
        R[j] = L[j] ^ f[P[j]-1];
    /*
    * Finally, the new L (the original R)
    * is copied back.
    */
    for (j=0; j<32; j++)
        L[j] = tempL[j];
}
/*
* The output L and R are reversed.
*/
for (j=0; j<32; j++) {
    t = L[j];
    L[j] = R[j];
    R[j] = t;
}
/*
* The final output
* gets the inverse permutation of the very original.
*/
for (j=0; j<64; j++)
    block[j] = L[FP[j]-1];
}

char *
crypt(pw,salt)
char *pw;
char *salt;
{
    register i, j, c;
    int temp;
    static char block[66], iobuf[16];

    for(i=0; i<66; i++)
        block[i] = 0;
    for(i=0; (c= *pw) && i<64; pw++){
        for(j=0; j<7; j++, i++)
            block[i] = (c>>(6-j)) & 01;
        i++;
    }

    setkey(block);

    for(i=0; i<66; i++)
```

```
        block[i] = 0;

for(i=0;i<2;i++){
    c = *salt++;
    iobuf[i] = c;
    if(c>'Z') c -= 6;
    if(c>'9') c -= 7;
    c -= '.';
    for(j=0;j<6;j++){
        if((c>>j) & 01){
            temp = E[6*i+j];
            E[6*i+j] = E[6*i+j+24];
            E[6*i+j+24] = temp;
        }
    }
}

for(i=0; i<25; i++)
    encrypt(block,0);

for(i=0; i<11; i++){
    c = 0;
    for(j=0; j<6; j++){
        c <<= 1;
        c |= block[6*i+j];
    }
    c += '.';
    if(c>'9') c += 7;
    if(c>'Z') c += 6;
    iobuf[i+2] = c;
}
iobuf[i+2] = 0;
if(iobuf[1]==0)
    iobuf[1] = iobuf[0];
return(iobuf);
}
```