30. STRUCTURI DE DATE ÎN SECURIZAREA INFORMAŢIEI

30.1 Structuri şi algoritmi utilizaţi în securizarea informaţiei

Structurile de date folosite în securizarea informaţiilor de obicei sunt structuri de date "neconvenţionale" şi se implementează la diferite niveluri ale aplicaţiei. Se observă, pentru a securiza informaţia, structurile de date se implementează la orice nivel al aplicaţiei. Se întâlnesc patru niveluri importante de implementare a structurilor de date:

1. la nivelul operaţiilor matematice pe care le implementează algoritmul – structuri de date algebrice implementate prin programare cu ajutorul claselor şi obiectelor (inele algebrice în câmp finit – Galois Field (GF), inele polinomiale sau inelele claselor de resturi);
2. la nivelul transformărilor de prelucrare a datelor, sau schema algoritmului (folosirea reţelelor Feistel, sau a unor reţele proprii şi combinate sau transformări proprii);
3. la nivelul modului de folosire a algoritmilor simetrici în practică (de obicei se întâlnesc două moduri principale de utilizare – cifrarea bloc şi cifrarea secvenţială);
4. nivelul protocoalelor de transmisie a datelor şi al memorării certificatelor publice (de obicei aceste structuri a pachetelor de date sau a certificatelor se descriu în ASN.1/DER – Abstract Syntax Notation 1 /Definite Encoding Rules).

În criptografie din punct de vedere matematic se întâlnesc următoarele tipuri de algoritmi folosiţi pentru:

- funcţii de dispersie ("digerarea" mesajului) – MD2, MD4, MD5 (Message Digest), SHA-1 (Secure Hash Algoritm);
- algoritmi care fundamentează sisteme criptografice simetrice (cele ce operează cu blocuri pe 64 biţi – DES, 3DES, FEAL, LOKI şi cele noi din standardul AES (Advanced Encryption Standard) – AES-Rijndael, Twofish, Blowfish, Serpent, RC6, DEAL, MARS, FROG, LOKI-97);
- algoritmi ce fundamentează sisteme criptografice asimetrice pentru criptare sau semnătură electronică (RSA, El Gammal, DSA).

30.2 Structuri de date la nivelul operaţiilor matematice

La primul nivel – al operaţiilor matematice – se generează structuri de date specifice, care se folosesc în algoritmi criptografici. Pentru o înţelegere mai bună se prezintă succint operaţia de adunare şi înmulţire în inelul algebric cu mulţimea numerelor în câmp finit GF(2^3), Galois Field (256).

În GF(2^3) se reprezintă numere de la 0 la 255, adică 256 de numere. Valoarea maximă care se poate reprezenta de un byte (octet) fără semn este 255 (toţi cei 8 biţi cu valoarea 1 => $2^0+2^1+2^2+2^3+2^4+2^5+2^6+2^7 = 2^8$-

1 = 255). Pe de o parte se va opera cu biţi, iar pe de altă parte se va opera cu polinoame matematice.

De exemplu valoare 7 în binar se reprezintă 0000 0111 pe un octet sau în forma polinomială este: $b(x) = 0*x^7+0*x^6+0*x^5+0*x^4+0*x^3+1*x^2+1*x^1+1*x^0$. Dacă x=2 înseamnă că b(2) = 7. Cu alte cuvinte, orice număr din intervalul 0..255, adică un octet *b* cu biţii b_7 b_6 b_5 b_4 b_3 b_2 b_1 b_0 se reprezintă ca un polinom algebric cu coeficienţii b_i în mulţimea {0, 1}:

$$b_7 x^7 + b_6 x^6 + b_5 x^5 + b_4 x^4 + b_3 x^3 + b_2 x^2 + b_1 x + b_0 \qquad (30.1)$$

a) adunarea este echivalentă cu XOR (SAU exclusiv) pe biţi sau adunarea modulo 2. De exemplu, dacă se adună următoarele numere în hexazecimal: '57'+'83' = 'D4' sau (fiecare cifră în hexazecimal ocupă 4 biţi) '01010111' + '10000011' = '11010100' sau în reprezentare polinomială $(x^6+x^4+x^2+x+1) + (x^7+x+1) = x^7+x^6+x^4+x^2$. Această operaţie se implementează la nivel de octet foarte uşor în C/C++, folosind XOR pe biţi (operatorul ^). Mulţimea {0, 1, …, 255} împreună cu operaţia XOR formează grup abelian (operaţia este internă, asociativă, comutativă, există elementul neutru '00', există elementul invers – însuşi elementul este inversul lui);
b) înmulţirea nu mai are echivalenţă cu o operaţie pe biţi existentă la procesoarele actuale. În reprezentarea polinomială, multiplicarea – înmulţirea în $GF(2^8)$ corespunde cu înmulţirea a două polinoame modulo un polinom ireductibil de grad 8. Un polinom ireductibil înseamnă un polinom care nu are alţi divizori în afară de 1 şi el însuşi. De exemplu pentru Rijndael (algoritmul acceptat ca standard AES – Advanced Encryption Standard în S.U.A), polinomul de grad 8 ireductibil este de numit m(x) şi are forma: $m(x) = x^8+x^4+x^3+x+1$, adică '11B' în reprezentare hexazecimală. De exemplu: '57'*'83'='C1', în hexazecimal sau polinomial:

$$((x^6+x^4+x^2+x+1)*(x^7+x+1)) = x^{13}+x^{11}+x^9+x^8+x^7+x^7+x^5+x^3+x^2+x+x^6+x^4+x^2+x+1 = x^{13}+x^{11}+x^9+x^8+x^6+x^5+x^4+x^3+1\ x^{13}+x^{11}+x^9+x^8+x^6+x^5+x^4+x^3+1 \text{ modulo } m(x) = x^7+x^6+1 \qquad (30.2)$$

În programare înmulţirea a două numere din $GF(2^8)$ se face ca sumă exponenţială a 2 logaritmi în bază număr prim în $GF(2^8)$. Concret, se generează într-un vector *alog[]* de mărime 256 toate valorile posibile a unui număr prim din GF(28). Orice număr prim din GF(28) ridicat la putere cu toate celelalte numere mod '11B' generează tot câmpul finit. Înmulţirea cu 3 adică x+1 înseamnă b(x)*(x+1) = b(x)*x+b(x), deoarece GF(28) este grup abelian şi faţă de a doua operaţie. Mai mult b(x)*x înseamnă deplasarea la stânga cu un bit şi în caz că apare transport pe ultimul bit se face XOR cu '11B'.

Codul sursă este următorul:

```
#include <stdio.h>
class inelGF {
public:
            unsigned char val;//1 octet fara semn adica 8 biti
            int alog[256];//functia exponentiala f(y) = 3y = (x+1)y
                        // f(4) = (x+1)*(x+1)*(x+1)*(x+1)
            int log[256];//functia logaritm inversa exponentialei

            //constructorii
inelGF(int b=0);
            inelGF(unsigned char b);

            //metodele folosite
            void generareALOGSiLog();
            void setVal(unsigned char b);
            unsigned char getVal();

//adunare
            inelGF operator+ (inelGF &);
            //inmultire
            inelGF operator* (inelGF &);
            //atribuire
            inelGF operator= (inelGF &);
};
inelGF::inelGF(int b) {
this->val = (unsigned char)b;
this->generareALOGSiLog();
}
inelGF::inelGF(unsigned char b) {
this->val = b;
this->generareALOGSiLog();
}
void inelGF::generareALOGSiLog() {
      alog[0]=1;
      int i=0,j=0;
      int ROOT = 0x11B;
      for(i=1;i<256;i++) {
            j=(alog[i-1] << 1)^alog[i-1];
                  if((j&0x100)!=0)j=j^ROOT;
                              alog[i]=j;
             }
      for (i = 1; i < 256; i++) log[alog[i]] = i;
}
void inelGF::setVal(unsigned char b) {
      this->val = b;
}
unsigned char inelGF::getVal() {
            return this->val;
}
inelGF inelGF::operator+(inelGF &igf2) {
      inelGF temp;
      temp.val = this->val^igf2.val;
      return temp;
}
inelGF inelGF::operator*(inelGF &igf2) {
      inelGF temp;
      int t1 = (int) temp.val;
      int t2 = (int) this->val;
      int t3 = (int) igf2.val;
      (t2 != 0 && t3 != 0) ?t1 = this->alog[(log[t2 & 0xFF] + log[t3 &
0xFF]) % 255] : t1 = 0;
```

```
            //adica  7*5  =  alog[log[7]+log[5]];//in  mod  normal
logaritmul //pastreaza toate propietatile in acest inel algebric ca
peste //numerele reale
      temp.val = (unsigned char)t1;
      return temp;
}
inelGF inelGF::operator= (inelGF &igf2) {
      this->val = igf2.val;
      return *this;
}
void main()
{
      inelGF a(87);
      inelGF b(131);
      inelGF rez1, rez2;
      rez1 =(a + b);
      rez2 =(a * b);
      printf(" Afisez rezultat adunare: %d\n",rez1.val);
      printf(" Afisez rezultat inmultire: %d\n",rez2.val);
}
```

Acest exemplu este o clasă creată în C++, unde sunt implementate cele două operaţii şi împreună cu câmpul Galois formează un inel algebric.

În realitate pentru reutilizarea codului se elaborează biblioteci cu clase foarte puternice în limbaje de asamblare pentru diferite procesoare şi în limbaje evoluate precum C/C++, C# şi Java. Clase foarte folositoare din practică se dovedesc cele pentru numere întregi foarte mari, *BigInt* pentru algoritmi cu cheii publice, şi clase pentru inele şi corpuri algebrice în câmp finit, şi clase pentru matrice, vectori şi liste cu informaţii utile în $GF(2^8)$.

30.3 Structuri de date la nivelul etapelor algoritmului de criptare/decriptare

La al doilea nivel, adică al operaţiilor de ansamblu al algoritmului sau schema algoritmului (folosirea reţelelor Feistel, sau a unor reţele proprii şi combinate sau transformări proprii), apar cele mai diverse structuri de date.

În figurile 30.1, 30.2 şi 30.3 se prezintă modelul general al unui sistem criptografic simetric "era 128 de biţi" (algoritmii care au fost acceptaţi finalişti pentru AES – Advanced Encryption Standard), standard cerut de NIST – National Institute of Standards and Technology.

Algoritmul de criptare câştigător la AES a fost cel care are schema prezentată în figura 30.1. Algoritmul a fost creat scris pe 2 Octombrie 2000, şi a fost proiectat de Vincent Rijmen şi Joan Rijndael. Aceasta este o structură de date "neconvenţională".

Structura propriu-zisă nu reţine date, în schimb preia date din alte structuri (fişier sau masiv de octeţi) şi pune date în alte structuri. Descrierea cifrului este relativ simplă dar nu face obiectul acestui capitol.

Această structură de date preia date sub formă de octeţi (128 de biţi) şi transmite criptaţi în funcţie de cheie utilizator tot 128 de biţi. În schema din figura 30.1 este prezentată o singură rundă, iar de exemplu pentru o cheie de 128 biţi şi blocuri de criptare de 128 biţi algoritmul se execută în 9 runde, adică figura 30.1 pusă cap la cap de 9 ori.

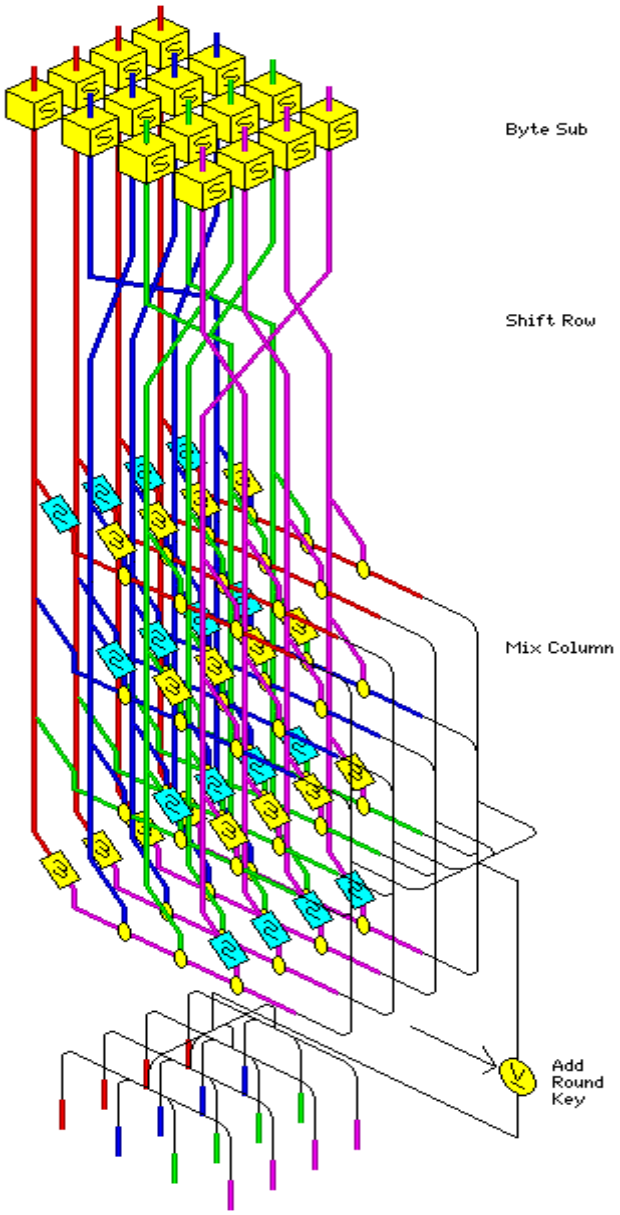

Figura 30.1 *Schema de criptare a algoritmului Rijndael AES [Copyright Sava00]*

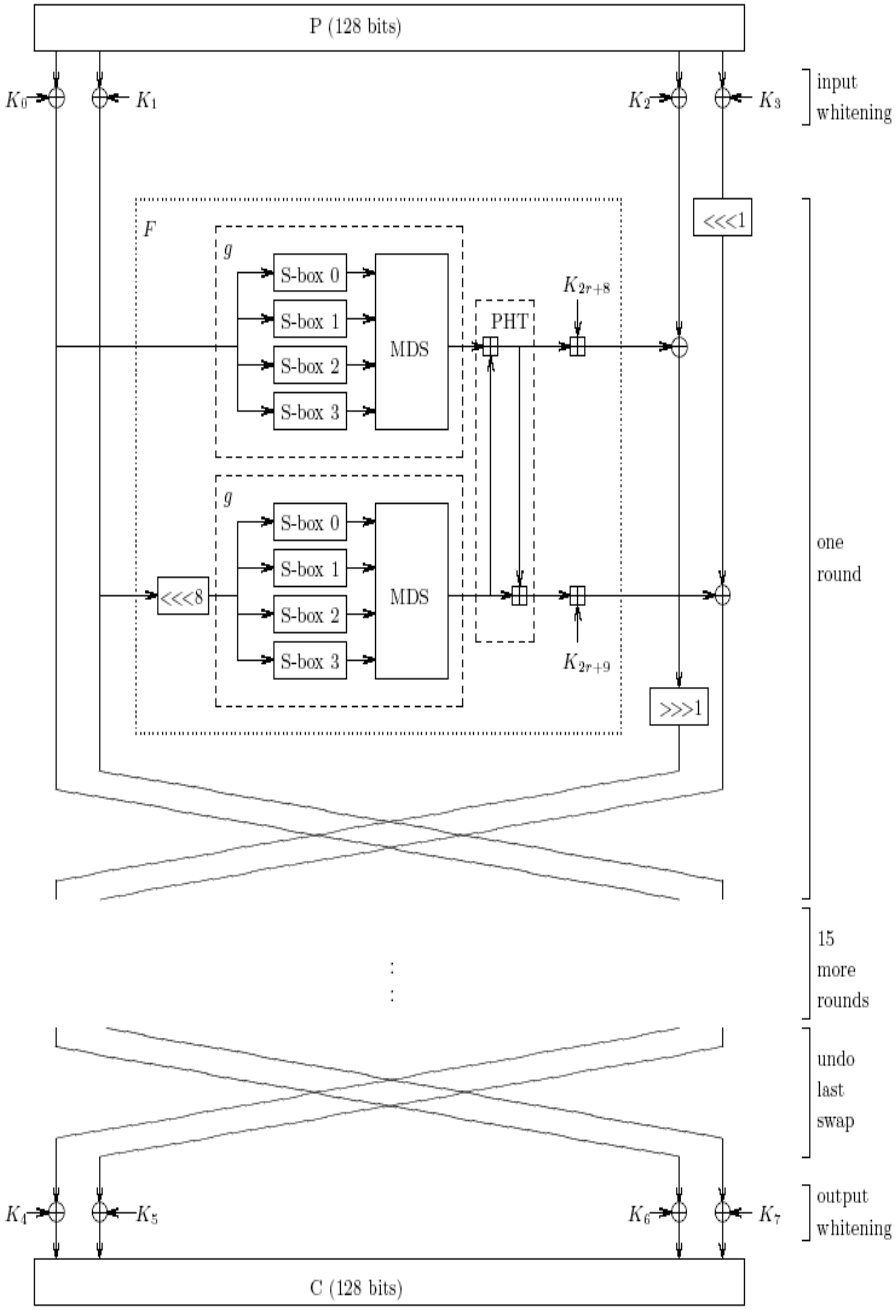

Figura 30.2 *Schema de criptare al algoritmului Twofish [Copyright Twof00]*

Algoritmul Twofish are 16 runde ce se bazează pe nişte "cutii S" care iau ca intrare 8 biţi şi produc 8 biţi prin aplicarea unei funcţii F bijective, în timp ce Rijndael se bazează în cutiile S pe o transformare neliniară în $GF(2^8)$).

Cifrul Twofish se mai bazează pe matrice 4x4 de distanţe maximum separabile în câmp $GF(2^8)$ – MDS (în Rijndael se foloseşte ShiftRow – deplasare pe biţi după anumite reguli asupra blocului de informaţii), aplicarea de transformări pseudo-Hadamard (în Rijndael se foloseşte

MixColumn ca în figura 30.1, dar în implementare prin programare înseamnă multiplicare de matrice în GF(2^8)), şi un mod de expandare al cheilor la fel de eficient ca în Rijndael.

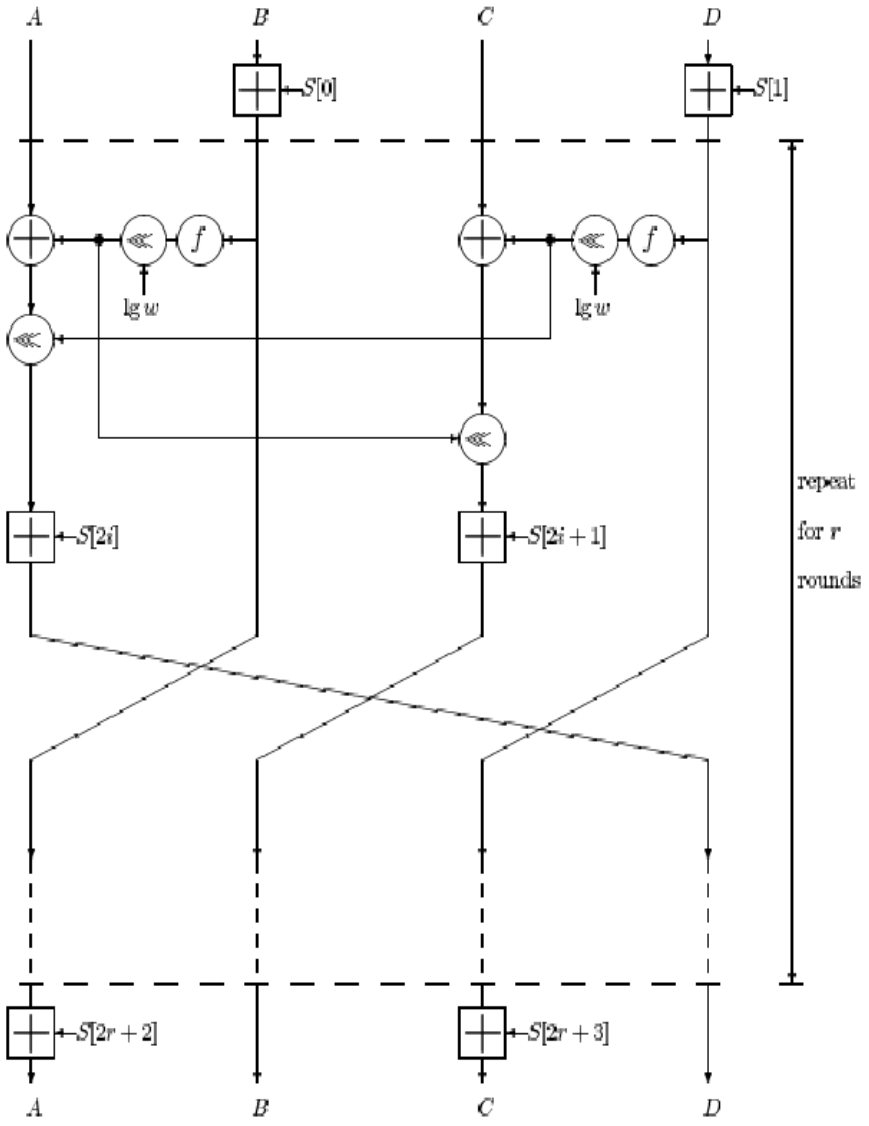

Figura 30.3 *Schema cifrului RC6 – Rivest Cipher 6 de la MIT [Copyright Rive00]*

Ideea principală este că şi în sistemele criptografice se folosesc structuri de date care sunt programate structuri ca atare sau ca simple secvenţe de transformări. Nu este necesar ca săgeţile din cifruri să fie pointeri către funcţii sau atribute. De fapt, ele reprezintă fluxurile de date.

În practică se pot implementa structuri de liste circulare ce păstrează rezultatele intermediare din fiecare rundă şi le aplică altor blocuri de date astfel încât descifrarea mesajului să fie imposibilă.

Fiecare algoritm are propria schemă de criptare şi propriile operaţii algebrice.

Nu numai algoritmii de criptare se pot folosi în structuri de date dar şi funcţiile de dispersie (*hash*) pot crea avalanşe de modificări imposibil de produs dacă se aleg structuri de date complicate.

Aceste lucruri se fac pentru producerea unor avalanşe de modificări greu de rezolvat fără cheia cifrului.

30.4 Structuri de date la nivelul de cifru al algoritmului

La nivelul modului de utilizare a algoritmilor simetrici se întâlnesc în practică două tipuri de cifrări: *cifrarea bloc* şi *cifrarea şir de caractere (secvenţială)*.

Cifrarea bloc operează cu blocuri de date în clar şi cifrate – de regulă 64 şi 128 biţi dar, uneori, şi mai mari.

Cifrarea secvenţială operează cu secvenţe de date în clar şi cifrate de mărime un bit sau octet, dar, uneori şi cu date de 32 de biţi.

În cazul cifrării bloc, acelaşi bloc de date în clar va fi cifrat de fiecare dată în acelaşi bloc de date cifrat, folosind aceeaşi cheie. În cazul cifrării secvenţiale, secvenţe similare de date în clar vor fi cifrate diferit, în cazul unor cifrări repetate.

Modurile de criptare constituie combinaţii ale celor două tipuri de bază, unele folosind metode feedback, altele realizând simple operaţii. Aceste operaţii sunt simple deoarece securitatea este atributul cifrării şi nu a modului în care se realizează schema de cifrare. Mai mult decât atât, modul de realizare a cifrării nu duce la compromiterea securităţii date de algoritmul de bază.

Cifruri bloc

Cifrarea ECB (Electronic Codebook). ECB este cea mai obişnuită cale de cifrare bloc, prin care un bloc de date sau text clar se transformă într-un bloc cifrat. Din moment ce acelaşi bloc de date se cifrează în acelaşi bloc cifrat, teoretic este posibilă crearea unei cărţi de coduri în care să se facă asocierea date în clar – date cifrate. Însă, pentru blocuri de 64 de biţi rezultă un număr de 264 intrări în cartea de coduri – mărime prea mare pentru a permite memorarea şi manipularea datelor. În plus fiecare cheie necesită propria carte de coduri.

Acesta este cel mai simplu mod de lucru, în care fiecare bloc de date în clar este cifrat independent. Nu trebuie ca fişierul care se criptează să intre în cifrare liniar, de la început până la sfârşit. Criptarea se face luând aleator blocuri din cadrul fişierului. Acest lucru este important pentru fişierele criptate care sunt accesate aleator, ca, de exemplu, în cazul bazelor de date. Schema de lucru arată ca în figura 30.4.

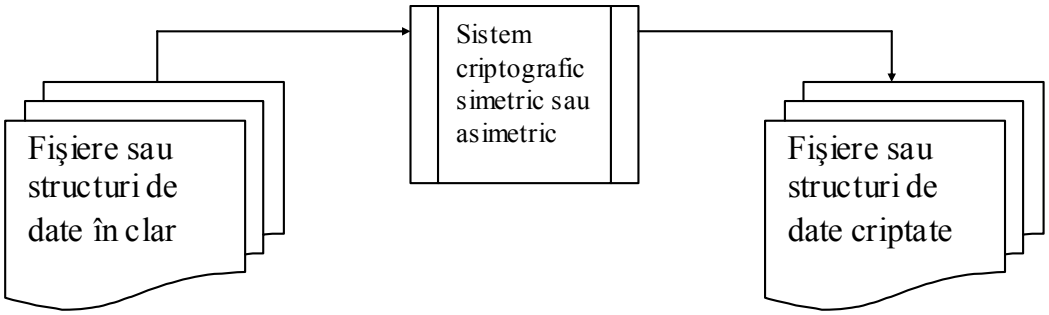

Figura 30.4 *Structură de date pentru cifrarea ECB*

Problema ECB-ului este că dacă un criptanalist, care deţine blocul de date în clar şi blocul de date cifrat echivalent pentru câteva mesaje, poate realiza o carte de coduri fără a cunoaşte cheia. În exprimarea curentă sunt fragmente de mesaje care tind să se repete. Mesajele pot avea structuri redundante sau şiruri lungi de spaţii sau zerouri. Dacă criptanalistul realizează că mesajul în clar '5ffa6ba1' se criptează în mesajul '778e342b', el poate decripta imediat mesajul respectiv acolo unde îl întâlneşte.

Cifrarea bloc cu înlantuire CBC (Cipher Block Chaining) adaugă mecanismului de criptare un bloc cu reacţie. Rezultatul criptării unui bloc anterior revine prin buclă şi intervine în criptarea blocului curent. În felul acesta, datele cifrate nu mai depind doar de datele în clar, ci şi de modul de cifrare a blocului anterior.

În CBC, datele în clar, înainte de a intra în blocul decriptare propriu-zis, sunt însumate modulo 2 (XOR) cu blocul de date cifrat anterior. Figura 30.5 reprezintă modul de criptare CBC:

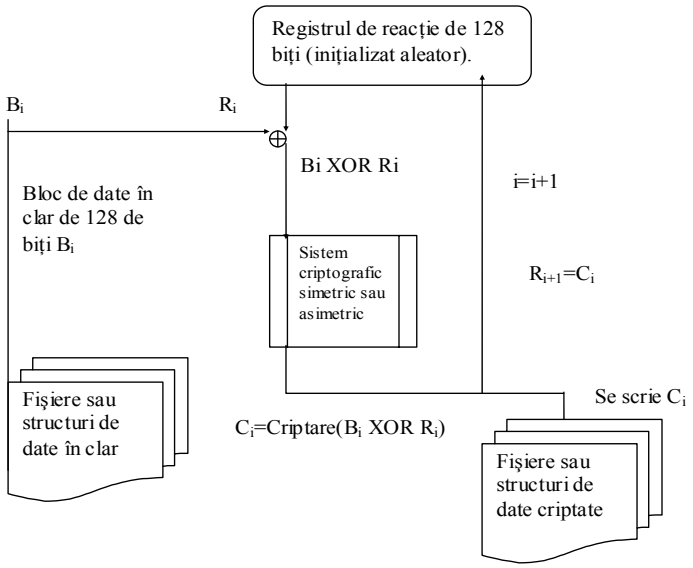

Figura 30.5 *Structura de date ce realizează criptarea CBC*

Paşii pe scurt sunt următorii:

- se iniţializează registrul de reacţie cu o funcţie hash de dispersie MD5, care produce rezumatul unei parole;
- pentru *i* de la 0 la numărul de blocuri a fişierului sau structurii de date -1 se execută *XOR* între blocul citit din fişier şi blocul de date din registrul de reacţie;
- se scrie blocul criptat în fişier;
- se atribuie registrului de reacţie blocul de biţi criptat;

- se incrementează *i* şi se reia procesul.

Aceasta e o structură de date compusă care implică două structuri de date de tip fişier şi o structură de tip masiv sau după necesităţi poate fi una dinamică, listă de liste de octeţi.

CBC face ca acelaşi bloc de date să se transforme în blocuri de date diferite, deoarece la diferite rulări valoarea de iniţializare a registrului de reacţie poate fi diferită. Dacă valoarea iniţială a registrului de reacţie rămâne neschimbată între rulări, atunci două mesaje identice folosind aceeaşi cheie se vor transforma în acelaşi mesaj criptat.

Vectorul de iniţializare, valoarea iniţială a registrului de reacţie, nu trebuie neapărat să fie secret.

Chiar dacă acest lucru pare greşit, şi anume de a nu ţine secret valoarea iniţială, nu este deoarece, oricum prin canal circulă blocurile criptate dar nu şi cheia. Deci, cineva care ar dori să spargă cifrul va trebui să cunoască ce structură de date s-a folosit şi ce algoritm şi mai mult să ştie protocolul de transmisie a datelor.

O posibilă descriere în C/C++ a acestui tip de structură este:

```
struct CBC {
      FILE *foriginal;
      FILE *fcriptat;
      unsigned char registruReactie[16];//16 octeți = 128 biți
      unsigned char buffer[16];
      AlgoritmCriptare ob;     /*obiectul care realizeaza criptarea ce
primeste parametrii blocul de date in clar si parola si "scoate"
blocul criptat"*/
};
```

O altă problemă este că fişierele sau alte structuri de date nu se împart exact la 128 de biţi, ceea ce înseamnă că se completează cu "0" până se ajunge la lungimea dorită multiplu de 128 de biţi.

Cifarea CBC cu propagare (PCBC – Propagation Cipher Block Chaining) este similar cu CBC, cu excepţia faptului că atât blocul de date anterior în clar, cât şi cel cifrat anterior sunt făcute *XOR* cu blocul curent de date în clar înainte sau după criptare ca în figura 30.6.

PCBC a fost utilizat în Kerberos versiunea 4 pentru a realiza în acelaşi timp secretizarea cât şi testul de integritate. O eroare în blocul de date cifrat, va determina decriptarea incorectă a tuturor blocurilor următoare. Aceasta înseamnă că este necesară transmiterea în finalul mesajului a unui bloc standard pentru a se asigura integritatea mesajului.

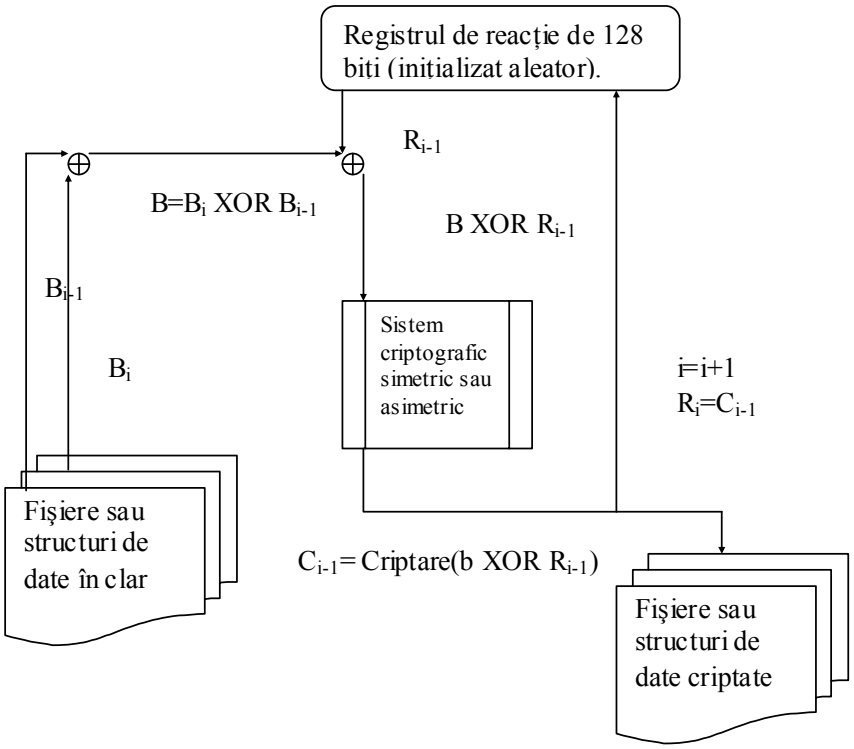

Figura 30.6 *Structură de date care realizează PCBC*

Paşii algoritmului cât şi iniţializările sunt cele de la CBC. În continuare se va prezenta codul sursă în C++ pentru codificarea acestei structuri. Descriere în totalitate a claselor *AlgoritmCriptare* şi *FunctieHash* nu face obiectul acestui capitol, iar completarea fişierului cu lungimea multiplu de 128 de biţi nu este prezentată.

```
#include <stdio.h>
#include <string.h>
#include <io.h>
class AlgoritmCriptare {
private:
      char *tipulAlg;
      char *parola;
      int   lungCheie;
      char *tipHashGenerareParola;
public:
      AlgoritmCriptare(char*,char*,int,char*);
      ~AlgoritmCriptare();
      unsigned char* doFinalEncryption(unsigned char*);
      unsigned char* doFinalDecryption(unsigned char*);
};
AlgoritmCriptare::AlgoritmCriptare(char* tipAlg,char* pass,int lCheie,
char* tipHash4Pass) {
      strcpy(this->tipulAlg,tipAlg);
      strcpy(this->parola,pass);
      this->lungCheie = lCheie;
      strcpy(this->tipHashGenerareParola,tipHash4Pass);
}
AlgoritmCriptare::~AlgoritmCriptare() {
      strcpy(this->tipulAlg,"0");
      strcpy(this->parola,"0");
      this->lungCheie = 0;
      strcpy(this->tipHashGenerareParola,"0");
```

```
}
unsigned char* AlgoritmCriptare::doFinalEncryption(unsigned char
 *message4Encryption) {
      if(stricmp(this->tipulAlg,"Rijndael")==0) {
      //se fac pasii algoritmului Rijndael sau al altui algoritm
            //la sfarsit se intoarce rezultatul care este in
            //principiu de 128 de biti

      //criptare
      unsigned char* rezultat = NULL;
      for(int i=0;i<128;i++)rezultat[i] = message4Encryption[i];
      //criptarea in functie de parola data
      return (unsigned char*)rezultat;
             }
      else return NULL;
}
unsigned char* AlgoritmCriptare::doFinalDecryption(unsigned char
 *message4Decryption) {
      if(stricmp(this->tipulAlg,"Rijndael")==0) {
      //se fac pasii algoritmului Rijndael sau al altui algoritm
      //la sfarsit se intoarce rezultatul care este in principiu
            //de 128 de biti

      //decriptare
      unsigned char* rezultat = NULL;
      for(int i=0;i<128;i++)rezultat[i] = message4Decryption[i];
      //decriptarea in functie de parola data
      return (unsigned char*)rezultat;
      }
      else return NULL;
}
class FunctieHash {
private:
      char *numeFctHash;
      unsigned char *mesajDeDigerat;
      int lmesaj;
public:
      FunctieHash(char*);
      ~FunctieHash();
      unsigned char * doFinalHash(unsigned char*);
      int lungime(unsigned char*);
};
FunctieHash::FunctieHash(char *nameFctHash) {
      strcpy(this->numeFctHash,nameFctHash);
}
FunctieHash::~FunctieHash() {
      strcpy(this->numeFctHash,"0");
      for(int i=0;i<this->lmesaj;i++)     this->mesajDeDigerat[i]='0';
      this->mesajDeDigerat = NULL;
}
int FunctieHash::lungime(unsigned char* cc) {
      int i=0;
      for(;((cc[i]!='\0')|(cc[i]!=NULL));i++);
      this->lmesaj = i;
      return i;
}
unsigned     char     *     FunctieHash::doFinalHash(unsigned     char*
message4Digest) {
      if(stricmp(this->numeFctHash,"MD5")==0) {
            this->mesajDeDigerat = message4Digest;
            this->lungime(this->mesajDeDigerat);
```

```
            unsigned char* rezultat=NULL;
                  for(int     i=0;i<this->lmesaj;i++)rezultat[i]     =
message4Digest[i];
            //se executa hashul
            return rezultat;
      }
      else return NULL;
}
class PCBC {
private:
      FILE *fo;
      FILE *fc;
      unsigned char* registruReactie;
           //de obicei are 16 octeți = 128 biți
      unsigned char *buffer[2];
      AlgoritmCriptare *obac;// obiectul care realizeaza criptarea
      // ce primeste parametrii blocul de date
           // in clar si parola si "scoate" blocul
           // criptat
          FunctieHash *obfh;     // obiect ce realizeaza functia de
           // dispersie si
                           // initializeaza registrul de reactie
      char *nfo;
      char *nfc;

public:
              PCBC(char  *nume_fis_orig,char  *nume_fis_criptat,  char*
nameAlg,
char*  passKeyAlg,  int  lenKey,  char*  nameHashF,  unsigned  char*
passregReact);
      ~PCBC();

             void doFinalPCBCEncryption();
};
PCBC::PCBC(char *nume_fis_orig,char *nume_fis_criptat, char* nameAlg,
char*  passKeyAlg,  int  lenKey,  char*  nameHashF,  unsigned  char*
passRegReact) {
      strcpy(this->nfo, nume_fis_orig);
      strcpy(this->nfc, nume_fis_criptat);
      this->fo = fopen(this->nfo,"rb");
      this->fc = fopen(this->nfo,"wb+");
      this->obac                          =                          new
AlgoritmCriptare(nameAlg,passKeyAlg,lenKey,nameHashF);
      this->obfh = new FunctieHash(nameHashF);
      this->registruReactie = this->obfh->doFinalHash(passRegReact);
}
void PCBC::doFinalPCBCEncryption() {
      unsigned char buff[16];
      unsigned char * rez;
      //inainte de a apela metoda se face "umplerea fisierului cu "0"
      //pana cand lungimea lui este divizibila cu 2*128 biti
      int j=0;
           while(!feof(this->fo)) {
      if(!feof(this->fo)){
      fread(this->buffer[1],sizeof(unsigned char[16]),1,this->fo);
      }
      if(!feof(this->fo)){
      fread(this->buffer[2],sizeof(unsigned char[16]),1,this->fo);
      }
long ll = ftell(this->fo);
```

```
fseek(this->fo,ll-sizeof(unsigned char[16]),SEEK_SET);
for(j=0;j<16;j++)            buff[j]  =  this->buffer[1][j]  ^  this-
>buffer[2][j];
for(j=0;j<16;j++)     buff[j] = buff[j] ^ this->registruReactie[j];
rez = this->obac->doFinalEncryption(buff);
fwrite(&rez,sizeof(char[16]),1,this->fc);
this->registruReactie = rez;

}
}
PCBC::~PCBC() {
      if(this->fo!=NULL)    fclose(fo);
      if(this->fc!=NULL)   fclose(fc);
}
```

Decriptarea PCBC se face la fel ca şi criptarea. Acest mod de criptare/decriptare se poate folosi cu orice tip de algoritm ce fundamentează un sistem criptografic simetric.

Cifruri secvenţiale

Prin *Cifrarea secvenţială (Stream Ciphers)* datele în clar se convertesc bit cu bit în text cifrat. Modelul general, structura de date care fundamentează modelul, este dat în figura 30.7.

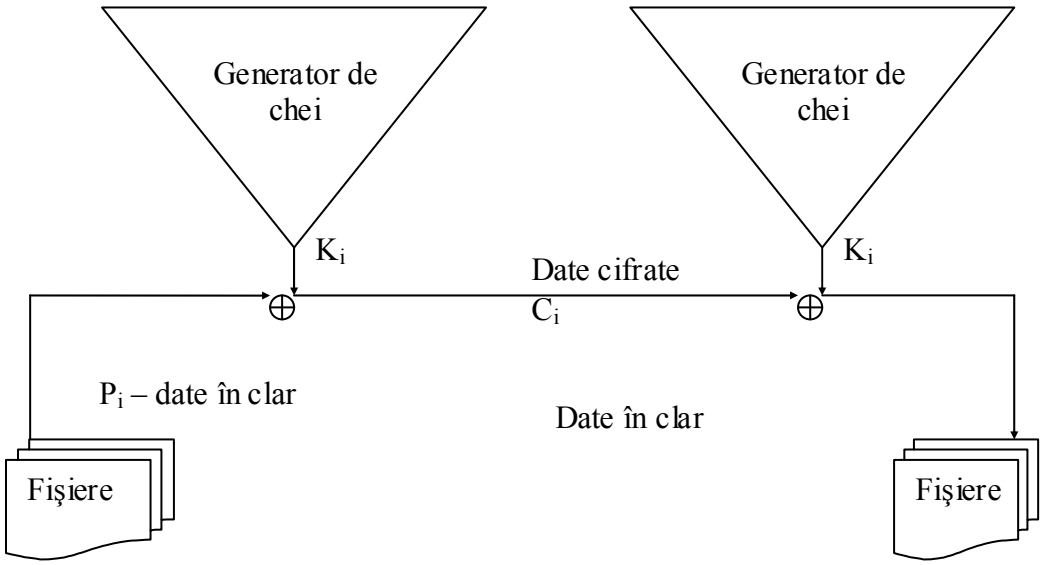

Figura 30.7 *Structură de date pentru cifrare secvenţială*

Există un generator de cheie care, în funcţie de timp şi, uneori parolă, generează un şir de biţi k_1, k_2, …, k_i. Cu acest şir cheie se face XOR cu şirul de biţi ai blocului de date în clar, p_1, p_2, …, p_i, pentru a produce un şir de date cifrate: $c_i = p_i \oplus k_i$.

La sfârşitul decriptării, rezultatul este obţinut din efectuarea unui XOR între criptogramă şi aceeaşi cheie curentă.

Securitatea sistemului depinde în întregime de generatorul de chei din această structură de date. Dacă această cheie generează acelaşi şir cheie, atunci securitatea sistemului nu este deosebită. Dacă, însă, acesta generează şiruri aleatoare, atunci există un înalt grad de securitate.

Acest tip este cel mai simplu mod de cifrare secvenţială. Mai există cifrarea secvenţială cu auto-sincronizare, cifrarea cu reacţie, cifrarea secvenţială sincronă şi cifrarea secvenţială cu reacţie la ieşire. Aceste tipuri de cifrări au tipuri proprii de structuri de date cu ajutorul cărora se realizează criptarea/decriptarea blocurilor de date.

30.5 Structuri de date la nivelul protocoalelor de transmisie a datelor şi al memorării certificatelor publice

Algoritmul dominant în sistemele oferite pe piaţa de software pentru sistemele de semnătură electronică îl reprezintă algoritmul RSA (Rivest – Shamir – Adleman), considerat un standard de facto în acest domeniu. RSA îşi bazează tăria criptografică pe imposibilitatea factorizării numerelor întregi foarte mari. Folosirea acestui algoritm în industrie se face conform unei suite de standarde, cunoscute sub denumirea de PKCS (Public-Key Cryptography Standards), realizate de proprietarul lui RSA, firma RSA Data Security Inc.'s: PKCS #3 – descrie metoda Diffie-Hellman de distribuire a cheilor criptografice simetrice:

- PKCS #1 – descrie metoda matematică de cifrare şi descifrare RSA, precum şi implementarea lor pentru realizarea două funcţii: semnarea electronică şi anveloparea digitală a cheilor criptografice simetrice (PKCS #1 include acum şi PKCS #2 şi PKCS #4);
- PKCS #3 – descrie metoda Diffie-Hellman de distribuire a cheilor criptografice simetrice;
- PKCS #5 – descrie metoda de implementare a cifrării simetrice DES-CBC, cu o cheie derivată din parolă;
- PKCS #6 – descrie standardul de certificat digital, supra-set al standardului X.509;
- PKCS #7 – descrie sintaxa generală a datelor ce urmează a fi criptate sau semnate;
- PKCS #8 – descrie sintaxa perechii private a cheilor RSA (cheie şi atribute);
- PKCS #9 – descrie atributele tipurilor definite în #6,#7,#8;
- PKCS #10 – descrie sintaxa standard pentru cererile de certificat;
- PKCS #11 – descrie interfaţa de program numită "Cryptoki";
- PKCS #12 – descrie sintaxa pentru memorarea în cadrul software-ului a unor informaţii criptografice, cum ar fi chei publice, chei secrete, certificate. Scopul îl constituie standardizarea unei structuri de fişier ce poate fi folosit de mai multe aplicaţii.

Forma electronică completă a acestor standarde se găseşte la adresa: http://www.rsa.com/rsalabs/pubs/PKCS/ sau se poate obţine prin e-mail de la adresa: pkcs@rsa.com . În ceea ce priveşte protocoalele implementate de diferite aplicaţii se întâlnesc structuri de date foarte diferite. De exemplu, în aplicaţia SSFTP descrisă în [Ivan02] chiar dacă se foloseşte un protocol de tip text în momentul de transmitere a cheilor publice se scriu în reţea obiecte serializate în format de certificat digital standard X509 descris în PKCS #6.

PKCS #6 versiunea 1.5 descrie sintaxa certificatelor extinse. Un certificat extins este format din certificatul unei chei publice, aşa cum este

el descris în standardul X509 şi un set de atribute; acest certificat extins este semnat de cel care îl emite. De aceea, autenticitatea certificatului poate fi verificată printr-o singură operaţie cu cheia publică şi oricând poate fi extras certificatul de tip X509.

Sintaxa certificatului extins în ASN1 este următoarea:

```
ExtendedCertificate::=SEQUENCE{
      extendedCertificateInfo ExtendedCertificateInfo,
      signatureAlgorithm SignatureAlgorithmIdentifier,
      signature Signature
}
      SignatureAlgorithmIdentifier::=AlgorithmIdentifier
      Signature::=BIT STRING
      ExtendedCertificateInfo::=SEQUENCE{
            version Version,
            certificate Certificate,
            attributes Attributes
      }
      Version::=INTEGER
      Attributes::=SET OF Attribute
```

Se observă că acestea sunt structuri în alte structuri de date şi sunt create pentru a asigura un standard şi o calitate superioară a aplicaţiilor informatice. Toate PKCS-urile conţin structuri în structuri de date şi sunt unanim acceptate în industria criptografică de către producători.

În orice tip de aplicaţie, dar mai ales în cele în care este necesară asigurarea unei protecţii superioare a informaţiilor, se folosesc structuri de date mai mult sau mai puţin convenţionale.

Structurile de date uşurează în general munca de programare. Structurile de date care se implementează nu ţin de limbajul de programare; se pot implementa în orice limbaj de programare universal: Pascal, C/C++, C#, Java. În criptografie se folosesc cu predilecţie astfel de structuri la toate nivelurile procesului implementat: de la nivelul aplicaţie al algoritmului până la protocolul de transmisie în reţea. Structurile alese ţin de natura aplicaţiei şi se o serie de cauze subiective (alegerea programatorului) şi obiective (restricţiile impuse de aplicaţie). Structurile de date mai sunt folosite şi în securitatea criptării simetrice pentru protecţia cheii. Managementul cheilor este vital în securitatea datelor şi cuprinde următoarele aspecte:

- generarea cheilor – se folosesc tabele de conversie şi structuri de date ce implementează funcţii de dispersie;
- distribuţia cheilor – transportul cheii secrete;
- memorarea cheilor.

Nu există domeniu al informaticii aplicate, şi cu atât mai mult în securizarea datelor, în care să nu se folosească structuri de date.