UNIVERSITATEA "TITU MAIORESCU" DIN BUCUREȘTI

FACULTATEA DE INFORMATICĂ

LUCRARE DE LICENȚĂ

COORDONATOR ȘTIINȚIFIC:

Prof.univ.dr.ing. Ciprian Răcuciu

ABSOLVENT:

Nicolae Jinga

SESIUNEA IUNIE

2017

UNIVERSITATEA "TITU MAIORESCU" DIN BUCUREȘTI

FACULTATEA DE INFORMATICĂ

LUCRARE DE LICENȚĂ

Aplicații bazate pe metode criptografice de tip bloc și de tip flux

COORDONATOR ȘTIINȚIFIC:

Prof.univ.dr.ing. Ciprian Răcuciu

ABSOLVENT:

Nicolae Jinga

SESIUNEA IUNIE

2017

Cuprins

Introducere

Motivarea alegerii temei și importanța domeniului criptografiei

Tema lucrării de licență aleasă se intitulează "Aplicații bazate pe metode criptografice de tip bloc și de tip flux" și este rezultatul curiozității mele intelectuale și al provocării de a implementa diverși algoritmi și cifruri de criptare cât și din dorința de documentare privitor la acest subiect.

Curiozitatea intelectuală s-a format și dezvoltat în cadrul cursurilor universitate de "Criptografie și securitatea informației" din cadrul Facultății de Informatică, Universitatea Titu Maiorescu. Cursurile din domeniul criptografiei, m-au făcut să aspir în această direcție, ulterior documentându-mă singur din dorința de a afla mai mult și de a progresa și specializa pe acest domeniu.

Dintotdeauna a fost nevoie de un mod de a prezerva informația atunci când este transmisă ca să nu poată fi accesată din exterior. Astfel a luat naștere "mesajul secret" cu diferite strategii de a ascunde informația. De la cele mai vechi trucuri antice, precum skytala, până în ziua de azi, la protejarea informației transmisă pe Internet, și accesarea bazelor de date cu ajutorul unor algoritmi bine puși la punct și standardizați. Apariția nevoilor de comunicare cât mai rapidă și cât mai sigură în cadrul rețelelor de comunicații, a direcționat domeniul criptografiei spre ideea de dezvoltare a acestor algoritmi ce pot asigura securitatea, confidențialitatea și integritatea datelor transmise. Este necesară implementarea acestora în aplicații software, cât și în echipamente hardware în scopul de a putea fi utilizați în criptografia modernă computațională.

Criptografia este un domeniu critic pentru dezvoltarea și viitorul societății umane, întrucât trebuie tot mereu să țină pasul cu dezvoltarea echipamentelor hardware cu putere din ce în ce mai mare de calcul, prin a actualiza și a dezvolta noi algoritmi care să asigure că informația rămâne sigură, confidențială și integrală. Datele sunt încărcate, stocare, transmise și prelucrate pe calculatore, astfel pe parcursul realizării acestor operații, trebuie avută în vedere protecția datelor.

Capitolul 1

Descierea domeniului criptografiei

Criptografia (din greacă, "kryptos" însemnând "secret", "ascuns", și "graphein" însemnând "scriere") este știința ce se ocupă cu studiul tehnicilor de a face comunicarea sigură în prezența terților, numindu-se adversari. Mai concret, criptografia se ocupă cu dezvoltarea și analizarea algoritmilor și protocoalelor ce previn terților sau publicului accesul la informația secretă transmisă. Diferite aspecte în securitatea informației, cum ar fi confidențialitatea, integritatea datelor, autentificarea sunt centrale pentru criptografia modernă. Criptografia modernă lucrează împreună cu alte științe reale cum ar fi matematica, informatica și ingineria pentru perfecționarea rezultatului la care se dorește să se ajungă. Aplicații ale criptografiei includ: comunicații, parole, ATM-uri, e-commerce etc.

Criptologia este considerată ca fiind o știință de foarte puțin timp. Aceasta cuprinde atât criptografia (scrierea secretizată), cât și criptanaliza. O parte a criptologiei, și anume criptografia, a fost utilizată încă de pe vremea lui Iulius Cezar (cifrul Cezar), însă a devenit o temă de cercetare academico-științifică abia începând cu anii 1970. Criptologia este legată de multe altele, de exemplu de teoria numerelor, algebră, teoria complexității, informatică.

Criptanaliza este studiul metodelor de obținere a înțelesului informațiilor criptate, fără a avea acces la informația secretă. De regulă, aceasta implică găsirea unei chei secrete, adică spargerea codului. Termenul de criptanaliză este folosit și cu referire la orice încercare de a ocoli mecanismele de securitate ale diferitelor tipuri de protocoale și algoritmi criptografici în general, și nu doar al criptării informației. Totuși, criptanaliza de regulă exclude metode de atac care nu țintesc slăbiciunile conceptuale ale criptografiei, cum ar fi mita, extragerea informației prin constrângeri fizice, intrarea prin efracție, logarea tastelor apăsate și ingineria socială; deși aceste tipuri de atac sunt o problemă importantă, sunt adesea mai eficiente decât criptanaliza tradițională. Deși scopul a rămas același, tehnicile și metodele de criptanaliză s-au schimbat drastic de-a lungul istoriei criptografiei, adaptându-se la creșterea complexității criptografice, de la metodele cu

creionul și hârtia din trecut, la mașini ca Enigma din al doilea război mondial, până la schemele criptografice computerizate din zilele noastre. Rezultatele criptanalizei au variat și ele, nefiind posibil să se obțină succese nelimitate în spargerea codurilor și există o clasificare ierarhică a ceea ce constituie un atac practic. La jumătatea anilor 1970, s-a introdus o nouă clasă criptografică: criptografia asimetrică. Metodele de spargere a acestor criptosisteme sunt radical diferite de cele anterioare, și implică de regulă rezolvarea unor probleme construite cu grijă în matematică, cea mai celebră dintre acestea fiind factorizarea întregilor.

Criptografia modernă computațională utilizează ca mod de lucru aceeaiași algoritmi ca și criptografia tradițională, și anume transpoziția și substituția, dar acest tip de criptografie computațională utilizează algoritmi de o complexitate ridicată. În criptografia modernă se bazează pe dezvoltarea unor algoritmi de o complexitatea ridicată astfel ca în situația interceptării unei cantități mari de text cifrat, criptanalistul să nu aibe posibilitatea extragerii unor informații relevante din textul clar sau despre cifru fără a utiliza și cheia secretă.

Ideal, se dorește dezvoltarea unor algoritmi ce previn în totalitate spargerea acestora de către orice adversar. Teoretic, este posibil spargerea aproape oricărui algoritm (One-Time-Pad este un algoritm ce este imposibil de spart și teoretic, dar este foarte greu de pus în practică), însă practic, este imposibil, datorită lipsei puterii suficiente de calcul. Astfel, acești algoritmi sunt considerați, siguri din punct de vedere computațional.

Până în era modernă, criptografia se referea exclusiv la criptare, ce reprezintă procesul de a converti informația din clar (numindu-se și plaintext) în informație inteligibilă (numindu-se și ciphertext). Decriptarea este procesul invers, și anume convetirea din cipher text în plaintext. Un cifru este o pereche de algoritmi (criptare, decriptare). Operațiile sofisticate ale unui cifru sunt controlate de un parametru, numit “cheie”. Cheia este secretă (doar părțile ce doresc comunicarea au acces la cheie), deobicei formată dintr-o secvență de caractere, ce face posibilă aplicatarea funcției de decriptare pentru a “traduce” din informația cifrată, în informație clară. Deci *decriptarea* și *criptanaliza* au același scop, și anume aflare textului clar. Diferența constă în faptul că în criptanaliză, textul clar trebuie aflat fără a se cunoaște cheia de decriptare.

În criptografia modernă, un sistem criptografic (criptosistem) este reprezentat de cinci parametrii: S = (P, C, K, E, D). P reprezintă mulțimea tuturor textelor clare, C reprezintă mulțimea

tuturor textelor cifrate, K reprezintă mulțimea tuturor cheilor posibile, E și D reprezintă algoritmii de criptare și respectiv decriptare, ce corespund fiecărei chei în parte.

Care este scopul de a avea o cheie în primul rând care trebuie ținută secretă, înloc de a ține secret întreg algoritmul de criptare? Răspunsul este unul simplu. Cum am menționat și mai sus, cheile deobicei au o lungime scurtă de caractere, astfel e mai ușor ca această să rămână un secret, decât internele unui întreg algoritm complicat. Dar acesta nu este motivul principal pentru care se folosesc cheile pentru parametrizarea funcțiilor de criptare și decriptare. Motivul principal este că permite ca algoritmul să fie public și cel mai important, să poată fi liber testată securitatea acestuia de către toată lumea, în special de către criptanaliști pentru a avea o garanție sporită că algoritmul este sigur din punct de vedere criptografic. Astfel, toată lumea se poate folosi de algoritm, dar personalizându-l cu o cheie.

Sunt cinci funcții primare ce trebuie avute în vedere când discutăm despre criptografie:

- *Confidențialitatea*: proprietatea de a păstra secretul informației, pentru ca aceasta să fie folosită numai de persoanele autorizate.
- *Autentificarea*: proprietatea de a identifica identitatea unei entități.
- *Integritatea*: proprietatea de evitare a oricărei modificări a informației din surse neautorizate
- *Non-repudierea*: mecanism ce dovedește că expeditorul chiar a trimis mesajul.
- *Schimbul-de-chei*: metoda prin care cheile criptării sunt transmite între expeditor și receptor.

Pentru ca un sistem criptografic să fie considerat sigur sau viabil, trebuie să îndeplinească o serie de condiții atât pentru algoritmii de criptare și decriptare cât și pentru cheia utilizată. Claude Shannon a menționat în lucrarea sa, “Communication Theory of Secrecy Systems” că următoarele sunt cele mai importante aspecte ale unui sistem criptografic sigur: cantitatea de secretizare, mărimea cheii, complexitatea operațiilor de criptare și decriptare, propagarea erorilor, expansiunea mesajului. Pentru implementarea unui criptosistem, trebuie avut în vedere două principii foarte importante, enunțate de Claude Shannon, și anume *confuzia* și *difuzia*.

Prin confuzie se înțelege că face relația dintre textul clar, textul criptat și cheie una foarte complicată. Efectul acestei tehnici este de a face foarte grea găsirea cheii, chiar dacă cripanaliștii

au un număr mare de perechi de text clar – text criptat produse de aceeași cheie. Astfel, fiecare bit din textul criptat trebuie să depindă de toată cheia. În mod concret, schimbând un singur bit în cheie, ar trebui să se schimbe complet întreg textul criptat. Pentru aplicarea tehnicii de confuzie, se vor folosi tabele de substituție. Difuzia reprezintă influența a oricărui bit din textul clar asupra

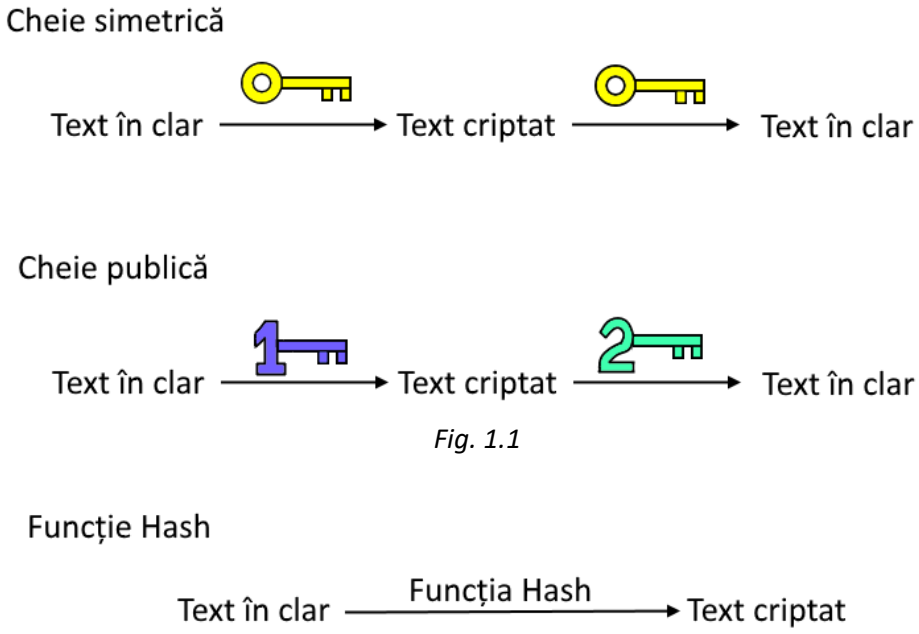

Fig. 1.1

multor alți biți în textul criptat. Efectul acestei tehnici este acela de a obliga criptanaliștii să intercepteze o cantitate mult mai mare de text criptat pentru a avea șanse teoretice să realizeze o criptanaliză statistică. Pentru aplicarea tehnicii de difuzie, se vor utiliza serii de permutări.

În decursul evoluției moderne a criptografiei, s-au dezvoltat trei tipuri de algoritmi criptografici (Figura 1.1): algoritmi criptografici simetrici, unde cheia se folosește atât la criptare, cât și la decriptare; algoritmi criptografici asimetrici, ce constau într-o cheie publică folosită pentru a face criptarea, în timp ce o altă cheie, secretă, este folosită pentru a face decriptarea. Această clasă de algoritmi asimetrici constituie Criptografia cu cheie publică. Și al treilea tip de algoritmi criptografici se numesc funcții hash, ce reprezintă transformarea matematică ireversibilă de criptare a informației, furnizată prin semnătură digitală. Este folosită în mod special pentru asigurarea integrității mesajului. Funcțiile hash nu au cheie deoarece mesajul în clar nu poate fi recuperat din mesajul criptat.

Schema generală a unui algoritm cu cheie simetrică este următoarea:

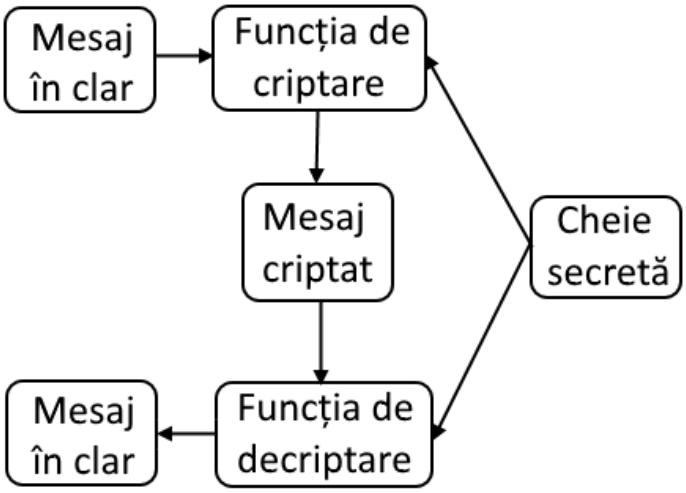

Fig. 1.1

După cum se poate observa în Figura 1.2, avem o cheie secretă, știută doar de cei care comunică, care este parametru pentru funcția de criptare și decriptare. Mesajul în clar este transformat în mesaj criptat prin funcția de criptare folosind cheia secretă, iar pentru a reveni înapoi la mesajul în clar, se va folosi funcția de decriptare, adică funcția inversă funcției de criptare, folosind aceeași cheie ca și la criptare.

Algoritmii cu cheie simetrică presupun că cel care trimite mesajul, cât și cel care îl primește, trebuie să aibe aceeași cheie secretă. Majoritatea vechilor sistemelor criptografice aveau nevoie ca unul dintre comunicanți să primească într-un fel o copie a acelei chei secrete printr-un canal fizic sigur. Aproape toate sistemele criptografice moderne, elimină nevoie de a transmite cheia printr-un canal fizic prin a folosi protocoale pentru chei publice, cum ar fi schimbul de chei Diffie-Hellman.

Capitolul 2

Metode criptografice.

Metodele criptografice reprezinta algoritmii desemnați siguri din punct de vedere criptografic. Criptarea cu cheie simetrică se poate face cu metode criptografice de tip bloc (block ciphers) sau cu metode criptografice de tip flux (stream ciphers). Deși cifrurile de tip bloc și de tip flux sunt foarte diferite, cifrul de tip bloc se poate implementa ca un cifru de tip flux, iar cifrul de tip flux se poate implementa ca un cifru de tip bloc. Cifrurile de tip bloc funcționează asupra unui bloc de date, în timp ce cifrurile de tip flux funcționează asupra unei singure entități de date la un moment dat. Diferențele dintre cele două tipuri de metode criptografice constau și în implementarea lor. Cifrurile de tip flux criptează și decriptează câte un bit la un moment dat, de aceea nu sunt prea eficiente pentru implementările software. Cifrurile de tip bloc sunt mai ușor de implementat software deoarece evită manipulările la nivel de bit (ce consumă foarte mult timp), și funcționează asupra blocuri de date. Pe de altă parte, cifrurile de tip flux sunt mult mai adecvate implementărilor hardware deoarece se pot implementa foarte eficient în silicon.

Definiție XOR

XOR vine din englezul Exclusive or, ce înseamnă "sau exclusiv" și este operația logică ce are ca dată de ieșire "adevărat" numai când datele de intrare diferă (adică una este adevărată, iar cealaltă falsă). Tabelul de adevăr al operației XOR, unde 0 înseamnă fals și 1 adevărat, este următorul:

Date de intrare		Date de ieșire
0	0	0
0	1	1
1	0	1
1	1	0

2.1. Metode criptografice de tip bloc

În criptografie, un algoritm de tip bloc este un algoritm deterministic operând pe o grupare de lungime fixă de biți, numită bloc. Cifrările de tip bloc sunt foarte des folosite pentru criptarea cantităților mari de date. Proiectarea modernă a cifrurilor de tip bloc stau la baza conceptului de cifru compus iterabil. Cifrul compus combină mai multe transformări simple cum ar fi substituția (S-Box), permutarea (P-Box), și aritmetică modulo. Iterațiile cifrului compus transmite informația criptată prin mai multe runde, fiecare rundă folosind o diferită "subcheie" derivată din cheia inițială. O implementare larg răspândită a acestui concept este *Rețeaua Feistel*, în special implementată în algoritmul DES. Multe alte implementări ale cifrurilor de tip bloc, cum ar fi AES, sunt clasificate ca *rețea de substituție-permutare*.

Un cifru de tip bloc constă într-o pereche de doi algoritmi, unul pentr criptare (E), și altul pentru decriptare (D). Ambii algoritmi acceptă două date de intrare: un bloc de mărime n biți și o cheie de mărime k biți; iar ambii au ca date de ieșire un bloc de n biți. Algoritmul de decriptare D este definit ca funcția inversă a algoritmului de criptare: $D = E^{-1}$.

Întrucât operația XOR stă la baza tuturor algoritmilor precizați mai jos, aceasta necesită o definiție în prealabil.

2.1.1. Data Encryption Standard (DES)

Data Encryption Standard (DES) este un algoritm cu cheie simetrică pentru criptarea datelor. Deși în ziua de azi este considerat nesigur din punct de vedere criptografic, a influențat foarte mult avansarea criptografiei moderne. Dezvoltat la începutul anilor 1970 în cadrul IBM și proiectat după rețeaua Feistel (de către Horst Feistel), algoritmul a fost prezentat la Biroul Național de Standarde (National Bureau of Standards), ca urmare a unei invitații pentru construirea unui sistem de criptare oficial. Publicarea sistemului de criptare standard, de către Agenția de Securitate Națională, a dus la adopția internațională a acestuia. Au apărut controverse în legătură cu elementele de proiectare clasificate, cheia cu o lungime relativ scurtă, și implicarea Agenției de Securitate Națională, creând suspiciunea că există o slăbiciune în algoritm de care doar arhitecții algoritmului știau. Cercetarea academică intensivă a algoritmului a condus în timp la înțelegerea modernă a cifrurilor de tip bloc și a criptanaliza acestora. DES este considerat nesigur pentru multe

aplicații. Aceasta datorită lungimii cheii de 56 de biți ca fiind prea mică; în Ianurie 1999, distributed.net împreună cu Electronic Frontier Fondation au colaborat pentru a sparge public algoritmul DES în 22 de ore și 15 minute. Algoritmul este considerat sigur din punct de vedere criptografic în forma triplu-DES (3DES). DES a fost retras ca și standard de către Institutul Național de Standarde și Tehnologie (National Institute of Standards and Technology – NIST), și înlocuit cu Advanced Encryption Standard (AES).

Descrierea algoritmului:

DES primește ca date de intrare o serie de biți de lungime fixă, iar prin o serie complicată de operații, transformă seria inițială de biți într-o altă serie de biți de aceeași lungime. În cazul algoritmului DES, lungimea blocului este de 64 de biți. DES folosește o cheie pentru personalizarea transformării astfel încât decriptarea să poată fi efectuată doar de către cei care cunosc cheia cu care s-a făcut criptarea. Lungimea cheii este de 64 de biți, însă doar 56 de biți sunt folosiți în algoritm. Ceilalți 8 biți sunt folosiți pentru verificarea parității, astfel se renunță la ei. Prin urmare, lungimea cheii efective este de 56 de biți. Cheia este stocată și transmisă ca 8 octeți. Un bit din fiecare octet al cheii se poate utiliza pentru detectarea erorilor în generarea cheilor, distribuire și stocare. Decriptarea folosește aceeași structură ca și criptarea dar cu cheile în ordine inversă. Aceasta permite o ușoară implementare a funcției de decriptare.

Structura generală a algoritmului este prezentată în Figura 2.1.

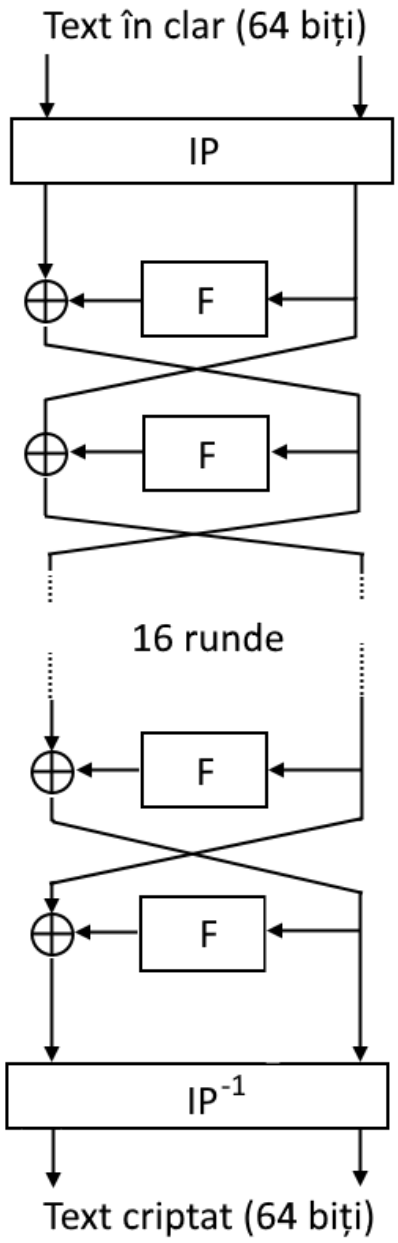

Fig. 2.1

Algoritmul cuprinde 3 etape:

1. Fie dat P textul clar inițial, de 64 biți. I se aplică o permutare IP (Initial Permutation) inițială fixată, obținându-se $P_0 = IP(P) = L_0R_0$. L_0 este format din primii 32 biți a lui P_0, iar R_0 din ultimii 32 biți.
2. Se efectuează 16 iterații (runde) ale unei funcții F.

La fiecare rundă se calculează L_iR_i ($1 \leq i \leq 16$) după regula:

$L_i = R_{i-1}$;

$R_i = L_{i-1} \oplus f(R_{i-1}, K_i)$

unde $\oplus$ este *sau-exclusiv* (XOR) a două secvențe binare. F este funcția care se va preciza, iar K_i sunt cheile K_1, K_2, .. , K_{16} sunt secvențe de 48 biți calculate din cheia de 56 biți K.

3. Blocului $R_{16}L_{16}$ i se aplică inversa permutării inițiale IP^{-1} (Inverse Permutation) pentru a obține textul criptat.

După cum se observă și în Figura 2.1, există 16 stagii identice, numite runde. De asemenea, este și o permutare inițială numită Initial Permutation sau IP, și o permutare finală numită Final Permutation sau Inverse Permutation, notată cu FP sau respectiv cu IP^{-1}. IP și IP^{-1} nu ai nicio semnificație din punct de vedere criptografic, dar au fost incluse la timpul respectiv pentru a facilita încărcarea blocurilor în 1970 pe hardware 8-bit.

Înainte de rundele principale, blocul de 64 de biți este împărțit în două blocuri de 32 de biți și sunt procesate alternativ, această încrucișare este cunoscută ca schema Feistel. Structura Feistel asigură că decriptarea și criptarea sunt procese foarte similare, singura diferență fiind că subcheile se aplică în ordine inversă când se face decriptarea. Simbolul $\oplus$ reprezintă operația pe biți XOR. Funcția F amestecă jumătate din blocul de 64 de biți cu o parte din cheie. Datele de ieșire din funcția F sunt apoi completate cu cealaltă jumătate de bloc, iar apoi jumătățile sunt interschimbate înainte de următoarea rundă. După finalul ultimei runde, jumătățile sunt interschimbate încă o dată; această trăsătură a structurii Feistel face ca operația de criptare și decriptare să fie procese similare.

Funcția Feistel (F)

Funcția Feistel operează asupra unei jumătăți de bloc (32 biți) la un moment dat și constă în 4 stagii. (Figura 2.2)

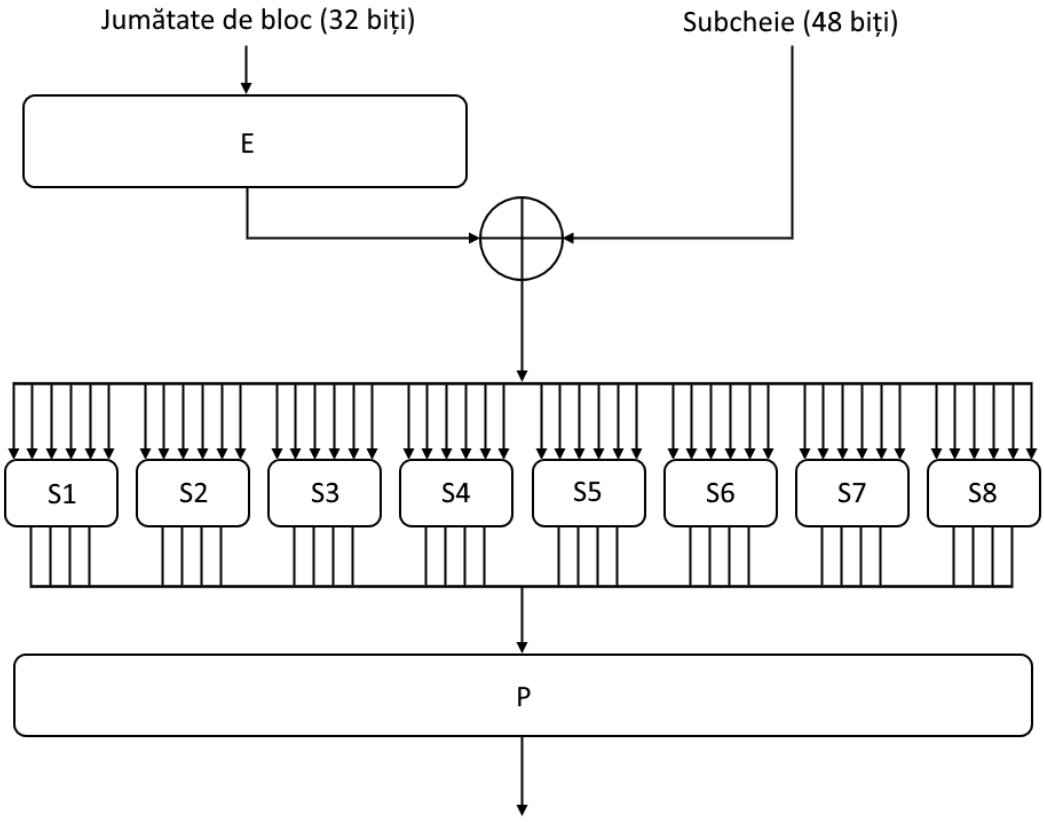

Fig. 2.2

1. Expansiune (Expansion Box): jumătatea de 32 biți este expandată la 48 biți folosing permutarea de expansiune, duplicând jumătate de biți. Datele de ieșire din cutia de expansiune constau în 8 serii de 6 biți ($8 \cdot 6 = 48$ biți), fiecare conținând o copie a celor 4 biți de intrare corespunzători, plus o copie imediat învecinată de ambele părți a datelor de intrare.
2. Amestecarea cheii (Key Mixing) este rezultatul combinării cu o subcheie folosind operația XOR. 16 subchei de 48 biți (câte una pentru fiecare rundă) sunt derivate din cheia principală folosing algoritmul de dispersare a cheii (Key Schedule).
3. Substituție (Substitution): după mixarea cheii, blocul este împărțit în 8 părți egale de 6 biți fiecare înainte de a fi procesate de către ”cutiile de substituție” (S-Boxes). Fiecare din cele

8 cutii de substituție înlocuiesc cei 6 biți de intrare cu 4 biți de ieșire (Figura 2.3) corespunzător cu o transformare neliniară dată sub forma unui tabel. Cutiile de substituție asigură aspectul principal de securitate al algoritmului DES; fără aceste cutii, cifrul ar fi liniar, și s-ar putea sparge ușor.

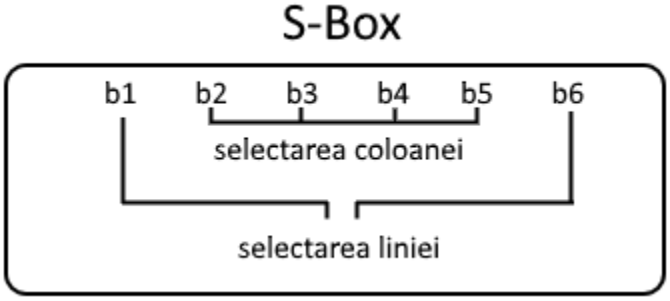

Fig. 2.3

4. Permutare (Permutation): în final, după ce obținem cei 32 de biți din cutiile de substituție, aceștia sunt rearanjați corespunzător unei permutări fixe. Scopul acestei operații este ca datele de ieșire a fiecărei cutii de substituție să fie împrăștiate peste 4 alte cutii de substituție din runda următoare. Cu alte cuvinte, are rol de a omogeniza datele ieșire din stagiul anterior.

Alternarea proceselor de substituție din S-Box-uri și permutarea biților cu operațiile de Permutare și Expansiune asigură procesele de *confuzie* și respectiv *difuzie*, concepte enunțate de Claude Shannon în 1940 ca fiind o condiție necesară pentru un algoritm criptografic sigur și practic.

Dispersarea cheii (Key Schedule)

În Figura 2.4 se observă algoritmul de dispersare a cheii pentru funcția de criptare. Inițial, 56 de biți din cheia de 64 de biți sunt selectați prin funcția PC-1 (Permuted Choice 1), restul de 8 biți sunt ori folosiți pentru verificarea parității biților, ori ignorați. Cei 56 de biți sunt împărțiți în două jumătăți a câte 28 biți fiecare. Fiecare jumătate este tratată individual. În fiecare rundă, ambele jumătăți sunt rotite în stânga cu un bit sau doi biți, specificat de numărul rundei curente. În cazul rundelor 1, 2, 9, 16, rotirea se face cu un bit, iar în restul rundelor se face cu 2 biți (desemnată prin simbolul "<<" în Figura 2.4). După aceea, 48 de biți sunt selectați cu funcția PC-2 (Permuted Choice 2), 24 de biți din jumătatea stânga și 24 de biți din jumătatea dreaptă. Rotația biților

înseamnă că un set diferit de biți este folosit în fiecare sub-cheie, fiecare bit fiind folosind în aproximativ 14 din cele 16 subchei. Dispersarea cheii pentru funcția de decriptare este identic cu cel al funcției de criptare cu excepția faptului că subcheile sunt folosite în ordine inversă.

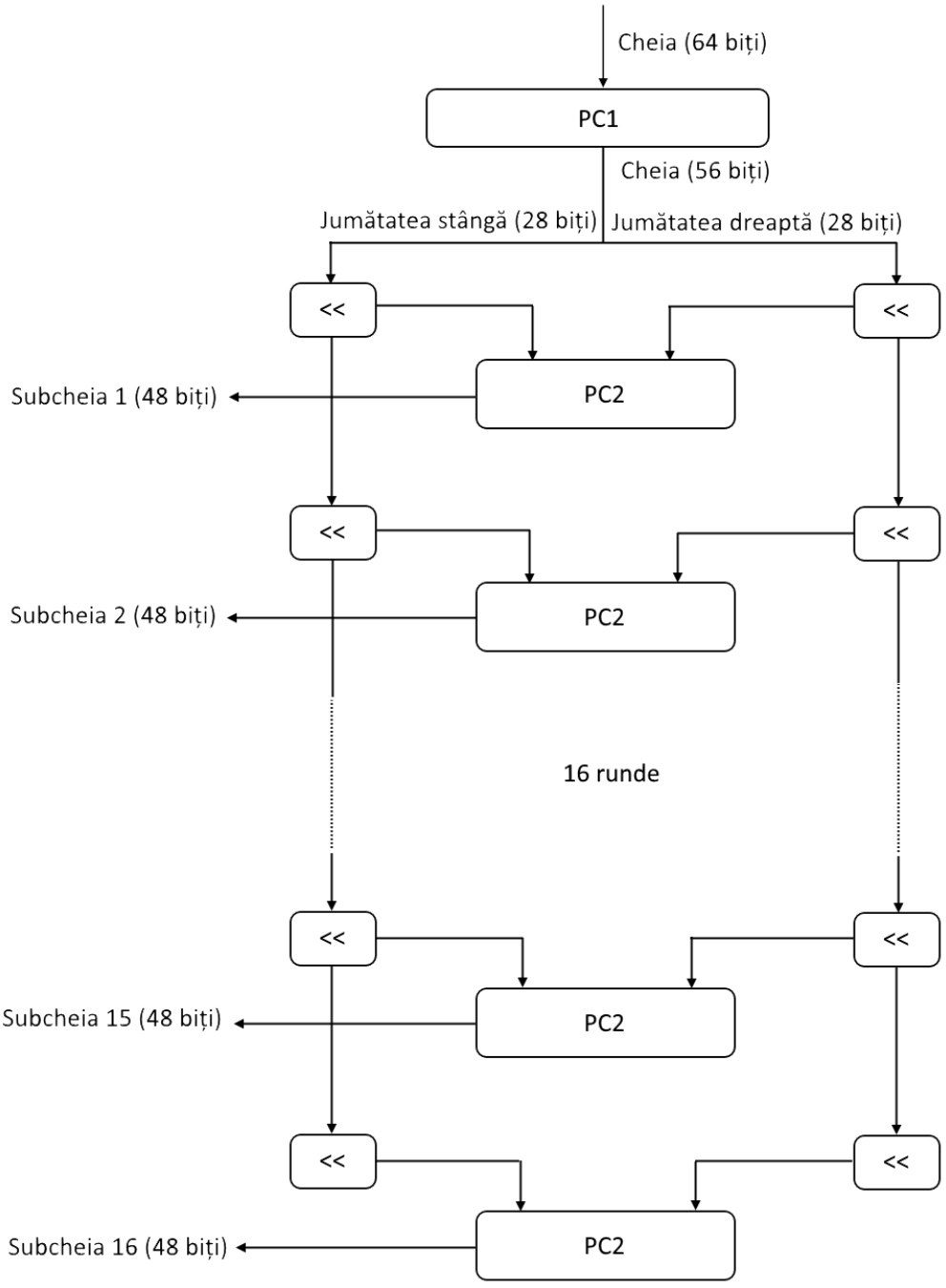

Fig. 2.4

Securitate și criptanaliză

Deși s-a descoperit mai multe informații în legătură cu criptanaliza cifrului DES decât oricare alt cifru de tip bloc, atacul cel mai practic asupra cifrului rămâne în continuare atacul prin forță brută (Brute force). Sunt cunoscute câteva proprietăți criptanalitice, și doar 3 atacuri teoretice sunt posibile, având o complexitate teoretică mai mică decât atacul prin forță brută, dar necesită un număr nerealist de corespondențe între textele clare și cele criptate, ceea ce nu este cazul în practică.

Pentru orice cifru, cea mai simplă metodă de atac este acea prin forță brută, însemnând încercarea fiecărei chei posibile. Lungimea cheii determină numărul posibil de chei. Pentru DES, s-au ridicat întrebări în legătura cu mărimea cheii, chiar înainte să fie adoptat ca și standard, și lungimea cheii a dictat necesitatea înlocuirii algoritmului. Vulnerabilitatea cifrului DES a fost practic demonstrată spre sfârșitul anilor 1990. În 1998, Electronic Frontier Foundation a construit o mașină special pentru spargerea cifrului DES. Motivația a fost ca să arate că algoritmul este vulnerabil atât în practică, cât și în teorie. Mașina a spart cifrul prin forță brută în mai puțin de 2 zile. Există 3 atacuri mai rapide din punct de vedere teoretic ce pot sparge DES cu o complexitate mai mică decât căutarea prin forță brută: criptanaliză diferențială, criptanaliză liniară și atacul Davies. Aceste atacuri sunt doar teoretice și sunt imposibil de pus în practică. Criptanaliza diferențială a fost redescoperită la sfârșitul anilor 1980 de Eli Biham și Adi Shamir; era cunoscută de IBM și NSA dar a fost ținută secret. Pentru a sparge cele 16 runde, criptanaliza diferențială necesită 2^{47} de corespondențe între textele clare și cele criptate. Cifrul DES a fost construit să fie rezistent la acest tip de atac. Criptanaliza liniară a fost descoperită de Mitsuru Matsui și are nevoie de 2^{43} de corespondențe texte clare-criptate. Atacul Davies, este un atac specific pentru DES, sugerat de Donald Davies în anii 1980. Cea mai puternică formă a atacului are nevoie de 2^{50} de corespodențe și are o complexitate de 2^{50} cu o rată de succes de 51%.

2.1.2. Triple Data Encryption Standard (3DES)

Triple Data Encryption Standard (3DES), oficial numit Triple Data Encryption Algorithm (TDEA sau Triple DEA), este un cifru cu cheie simetrică de tip bloc, ce aplică cifrul DES de 3 ori pentru fiecare bloc de date. Lungimea blocului este de 64 de biți. Lungimea cheii algoritmului DES este de 56 de biți și era suficient când acesta a fost proiectat, dar cu disponibilitatea în creșterea puterii de calcul, atacurile prin forță brută erau foarte practice. Triple DES furnizează o metodă simplă de creștere a mărimii cheii algoritmului DES pentru protejarea împotriva atacurilor prin forță brută, fără a fi nevoie de a proiecta un nou cifru de tip bloc. Lungimea cheii pentru algoritmul 3DES este de 168, 112, 56 biți respectiv opțiunii de cheie 1, 2, 3.

Algoritmul constă într-un pachet de chei K_1, K_2, K_3, fiecare cu 56 de biți (excluzând biții de paritate). Algoritmul de criptare (Fig. 2.5) este $C = E_{K3}(D_{K2}(E_{K1}(P)))$, însemnând criptare DES cu cheia K_1, decriptare DES cu cheia K_2, iar apoi criptare DES cu cheia K_3 asupra textului în clar P, obținând textul cifrat C. Operația de decriptare este invers operației de criptare: $P = D_{K1}(E_{K2}(D_{K3}(C)))$, și anume decriptare folosind cheia K_3, criptare folosind cheia K_2, decriptare folosind cheia K_1. Fiecare criptare triplă, criptează un bloc de 64 de biți de date. În ambele cazuri, criptare și decriptare, operația din mijloc anulează prima, respectiv ultima operație. Aceasta crește rezistența algoritmului când se folosește opțiunea de cheie 2, și furnizează compatibilitate înapoi cu DES folosind opțiunea de cheie 3.

Opțiunile de cheie

Standardul definește 3 opțiunii de cheie:

- Opțiunea de cheie 1: Toate cele 3 chei sunt independente.
- Opțiunea de cheie 2: Cheia K_1 și cheia K_2 sunt independente, iar $K_3 = K_1$.
- Opțiunea de cheie 3: Toate cele 3 chei sunt identice $K_1 = K_2 = K_3$.

Opțiunea de cheie 1 este cea mai sigură, deoarece are $3 \cdot 56 = 168$ biți independenți pentru cheie. Opțiunea de cheie 2 asigură mai puțină securitate având doar $2 \cdot 56 = 112$ biți pentru cheie. Această opțiune este mai sigură decât a cripta de 2 ori cu DES deoarece protejează împotriva atacurilor de tip “întâlnire-la-mijloc”. Opțiunea de cheie 3 este echivalentă cu DES, având doar 56 biți pentru

cheie. Furnizează compatibilitate înapoi cu DES deoarece prima și a doua operație DES se anulează între ele.

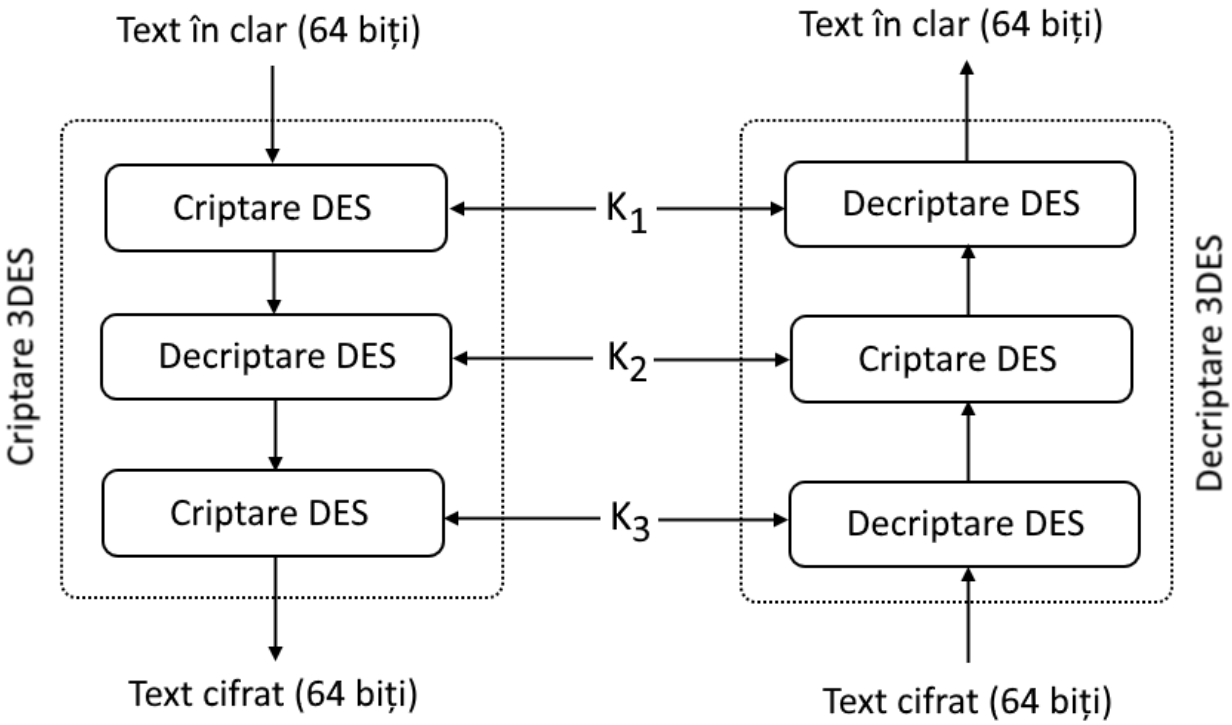

Fig. 2.5

Securitate

În general, 3DES cu trei chei independente (opțiunea de cheie 1) are o lungime de cheie de 168 biți (3 chei DES de 56 biți fiecare), dar datorită atacului de tip "întâlnire-la-mijloc", în practică oferă doar 112 biți de securitate. Opțiunea de cheie 2 reduce cheia efectivă la 112 biți (deoarece a treia și prima cheie sunt identice), dar această opțiune este predispusă la anumite corespondențe de text clar – text cifrat, astfel, a fost desemnat de către NIST că are doar 80 biți de securitate. Această opțiune este considerată spartă în ziua de azi, deoarece se poate face o căutare în tot spațiul cheilor 3DES foarte ușor și accesibil.

Cel mai bun atac asupra opțiunii de cheie 1, reprezentând și cea mai bună opțiune de cheie pentru 3DES, necesită 2^{32} corespondențe text clar – text cifrat, 2^{113} pași, 2^{90} criptări DES și 2^{88} memorie. Spargerea acestuia nu este încă practică, iar NIST consideră 3DES cu opțiunea de cheie 1 să reziste până în 2030.

2.1.3. Advanced Encryption Standard (AES)

Conform bibliografiei studiate [7], Advanced Encrytion Standard (AES), de asemenea cunoscut după numele său original Rijndael, este o specificare pentru criptarea datelor stabilită de NIST în 2001. AES este un subset al cifrurilor Rijndael, proiectat de doi criptografi belgiani, Vincent Rijmen și Joan Daemen, ce au prezentat acest proiect către NIST în timpul procesului de alegere a AES-ului. Rijndael este o familie de cifruri cu diferite mărimi pentru bloc și cheie. Pentru AES, NIST a ales 3 cifruri din familia de cifruri Rijndael, fiecare cu o mărime de bloc de 128 biți, dar cu lungime diferite de chei: 128, 192 și 256 biți. AES a fost adoptat de către guvernul american și acuma este folosit la scară globală. A înlocuit DES, care a fost publicat în 1977. Algoritmul descris de AES este un algoritm cu cheie simetrică, însemnând că aceeași cheie este folosită atât pentru criptarea cât și pentru decriptarea datelor.

AES are la bază principiul cunoscut ca "rețeaua de substituție-permutare", o combinare de operațiile de substituție și de permutare, și este rapid atât în implementările software cât și în implementările hardware. Spre deosebire de predecesorul său DES, AES nu are la bază o rețea Feistel. AES este o variantă a cifrului Rijndael ce are o lungime fixă a blocului de 128 de biți, și o lungime a cheii de 128, 192, și respectiv 256 biți. Familia de cifruri Rijndael este specificată cu o mărime de bloc și de cheie multiplu de 32 de biți, cu un minim de 128 biți, și un maxim de 256 biți.

AES lucrează pe o matrice pătratică de mărime 4, numită "stare" (State). Majoritatea calculelor pentru AES sunt făcute într-un anumit câmp finit. Octeții sunt așezați în matrice pe coloană în specificația AES, dar cum accesul la memorie se face pe linie în implementările software, pentru o performanță crescută este indicată așezarea pe linie a octeților. De exemplu, dacă avem 16 octeți b_0, b_1, ..., b_{15}, acești octeți sunt reprezentați în matrice conform specificației în felul următor:

$$\begin{bmatrix} b_0 & b_4 & b_8 & b_{12} \\ b_1 & b_5 & b_9 & b_{13} \\ b_2 & b_6 & b_{10} & b_{14} \\ b_3 & b_7 & b_{11} & b_{15} \end{bmatrix}$$

Mărimea cheii folosită pentru AES specifică numărul runde pe care acesta îl face pentru criptare, respectiv decriptare

- 10 runde pentru chei de 128 biți
- 12 runde pentru chei de 192 biți
- 14 runde pentru chei de 256 biți

Astfel, avem următorul tabel:

	Lungimea cheii (Nk cuvinte)	Lungimea blocului (Nb cuvinte)	Numărul de runde (Nr)
AES-128	4	4	10
AES-192	6	4	12
AES-256	8	4	14

Descrierea algoritmului

1. Key Expansion – cheile de rundă, sau subcheile, sunt derivate din cheia principală folosind algoritmul Rijndael de dispersare a cheii. AES necesită subchei de 128 de biți pentru fiecare rundă, plus încă una.
2. Runda Inițială
    a. AddRoundKey
3. Rundele intermediare
    a. SubBytes
    b. ShiftRows
    c. MixColumn
    d. AddRoundKey
4. Runda finală
    a. SubBytes
    b. ShiftRows
    c. AddRoundKey

Se observă că în ultima rundă nu se mai face operația MixColumns.

Atât pentru funcția de criptare, cât și pentru funcția de decriptare, algoritmul AES folosește o funcție pe rundă constând în patru operații la nivel de octet diferite:

1) SubBytes: substituție de octeți non-lineară folosind un tabel de substituție (S-Box) unde fiecare octet este înlocuit cu un altul corespunzător din tabel

2) ShiftRows: operație de transpoziție ce deplasează rândurile din matricea de stare cu diferite valori corespunzător numărului rundei curente

3) MixColumns: operație de combinare asupra coloanelor matricii de stare, combinând cei 4 octeți din fiecare coloană

4) AddRoundKey: adăugarea unei "chei de rundă" (Round Key) la matricea de stare însemnând că fiecare octet din matricea de stare este combinat cu un bloc din cheia de rundă folosind operația pe biți XOR.

Toți octeții în algoritmul AES sunt interpretați ca fiind elemente în câmpul finit Galois GF(2^8). Elementele câmpului pot fi adunate și înmulțite, însă aceste operații sunt diferite față de cele folosite la numere. Mai jos vom defini operațiile de adunare și înmulțire a elementelor din câmpul Galois GF(2^8).

Adunarea a două elemente într-un câmp finit este realizată prin adunarea coeficienților puterilor corespunzătoare polinoamelor a celor două elemente. Adunarea este efectuată cu operația XOR modulo 2, astfel ca $1 \oplus 1 = 0$, $1 \oplus 0 = 1$, $0 \oplus 1 = 1$, $0 \oplus 0 = 0$. Scădearea polinoamelor este identică cu adunarea polinoamelor. Alternativ, adunarea elementelor într-un câmp finit poate fi descris ca adunarea modulo 2 a biților corespunzători dintr-un octet. Pentru doi octeți $\{a_7a_6a_5a_4a_3a_2a_1a_0\}$ și $\{b_7b_6b_5b_4b_3b_2b_1b_0\}$, suma este $\{c_7c_6c_5c_4c_3c_2c_1c_0\}$, unde $c_i = a_i \oplus b_i$.

Înmulțirea în GF(2^8), în reprezentarea polinomială, corespunde cu înmulțirea polinoamelor modulo un **polinom ireductibil** de grad 8. Un polinom este ireductibil dacă divizorii săi sunt doar el însuși și unu. Pentru algoritmul AES, acest polinom ireductibil este:

$$m(x) = x^8 + x^4 + x^3 + x + 1$$

sau {01}{1b} în notație hexazecimală. Motivul pentru care facem modulo m(x) este acela de a asigura că rezultatul va fi un polinom binar de grad mai mic decât 8, astfel putând fi reprezentat pe un octet. Spre deosebire de adunare, nu există o operație simplă la nivel de octet corespunzătoare

acestei înmulțiri. Înmulțirea definită mai sus este asociativă, iar elementul {01} reprezintă elementul neutru.

Înmulțirea cu x reprezintă înmulțirea coeficientului al fiecărei puteri al unui polinom cu x. Înmulțind polinomul $b(x) = b_7x^7+b_6x^6+b_5x^5+b_4x^4+b_3x^3+b_2x^2+b_1x+b_0$ cu x va rezulta în $b_7x^8+b_6x^7+b_5x^6+b_4x^5+b_3x^4+b_2x^3+b_1x^2+b_0x$. Rezultatul $x \cdot b(x)$ este obținut prin reducerea rezultatului obținut mai devreme cu m(x), definit mai sus. Dacă $b_7 = 0$, atunci rezultatul este în formă redusă. Dacă $b_7 = 1$, reducerea este realizată prin aplicarea operației de XOR cu polinomul m(x). Înmulțirea cu x ({00000010} sau {02}) poate fi implementată la nivel de octet (Figura 2.6) ca o shiftare la stânga urmată de operația XOR cu {1b}. Această operație pe octeți este notată "xtime()". Înmulțirea cu puteri mai mari ale lui x poate fi implementată prin aplicări repetate ale funcției xtime().

```
unsigned char AES::xtime(unsigned char x)
{
    return (x << 1) ^ (x & 0x80 ? 0x1b : 0);
}
```

Fig. 2.6

Descrierea operațiilor interne ale algoritmului AES.

Operația SubBytes

În operația SubBytes, fiecare octet $a_{i,j}$ din matricea de stare este înlocuit cu o valoare corespunzătoare din tabelul de substituție. Această operație furnizează non-linearitatea cifrului. Tabelul de substituție este obținut prin inversa operației de înmulțire asupra câmpului $GF(2^8)$. Pentru a preveni atacurile bazate pe proprietăți algebrice simple, tabelul de substituție este construit combinând funcția inversă cu transformare afină inversabilă. Fiecare octet se va calcula prin relația: $b_i` = b_i \oplus b_{(i+4) \bmod 8} \oplus b_{(i+5) \bmod 8} \oplus b_{(i+6) \bmod 8} \oplus b_{(i+7) \bmod 8} \oplus c_i$, pentru $0 \leq i < 8$, unde b_i reprezintă bitul corespunzător poziției i din cadrul octetului, iar c_i reprezintă bitul corespunzător poziției i din octetul reprezentând valoarea hexazecimală.

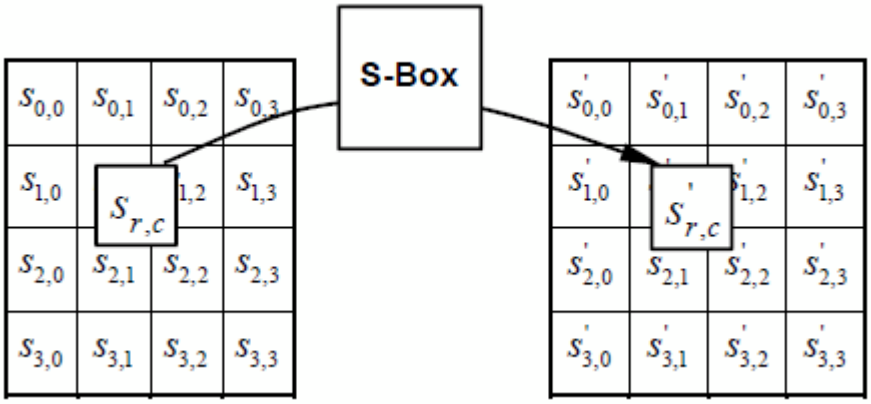

Fig. 2.7 [7]

Figura 2.7 ilustrează effectul operației SubBytes asupra matricii de stare.

Operația ShiftRows

În operația ShiftRows are loc deplasarea ciclică a octeților la nivel de rând al matricii de stare. Deplasarea ciclică are loc în următorul mod (Figura 2.8): Rândul unu nu va înregistra nici o deplasare, rândul doi va înregistra o deplasare în partea stângă cu o poziție, rândul trei va înregistra o deplasare în partea stânga cu două poziții, rândul patru va înregistra o deplasare în partea stângă cu trei poziții. În urma acestui pas, fiecare coloană a matricii de stare rezultat va fi compusă din octeți de pe fiecare coloană a matricii de stare inițiale.

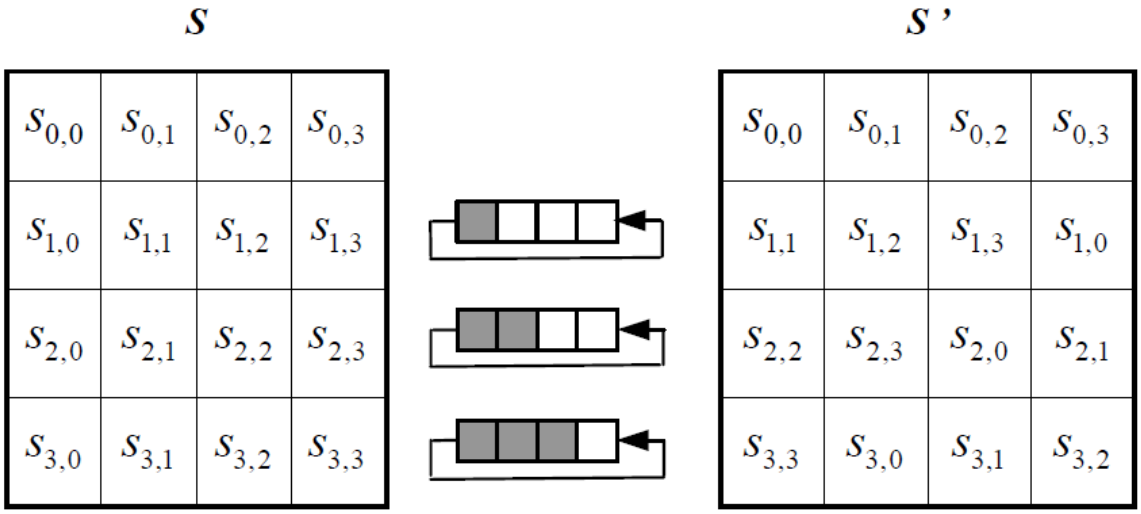

Fig. 2.8 [7]

Importanța acestei operații este de a evita să avem coloanele liniar independente, caz în care, algoritmul AES ar ajunge să fie patru cifruri de tip bloc independente.

Operația MixColumns

În operația MixColumns, 4 octeți din fiecare coloană a matricii de stare sunt combinați folosind o transformare lineară inversabilă. Funcția MixColumns primește ca date de intrare 4 octeți și furnizează ca date de ieșire 4 octeți, unde fiecare octet ca dată de intrare afectează toți cei 4 octeți ca date de ieșire. Împreună cu operația ShiftRows, MixColumns oferă proprietatea de difuzie în cifru. În timpul operației, fiecare coloană este transformată folosind o matrice fixă:

$$\begin{bmatrix} 2 & 3 & 1 & 1 \\ 1 & 2 & 3 & 1 \\ 1 & 1 & 2 & 3 \\ 3 & 1 & 1 & 2 \end{bmatrix}$$

Coloanele sunt considerate ca fiind polinoame de grad 4 peste câmpul finit GF(2^8) și înmulțite modulo $x^4 + 1$ cu un polinom fix a(x) dat de $a(x) = \{03\}x^3 + \{01\}x^2 + \{01\}x + \{02\}$. Aceasta se poate scrie ca o înmulțire de matrici $s`(x) = a(x) \otimes s(x)$:

$$\begin{bmatrix} s'_{0,c} \\ s'_{1,c} \\ s'_{2,c} \\ s'_{3,c} \end{bmatrix} = \begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix} \begin{bmatrix} s_{0,c} \\ s_{1,c} \\ s_{2,c} \\ s_{3,c} \end{bmatrix}, \text{ pentru } 0 \leq c < Nb$$

Ca rezultat al acestei înmulțiri, cei 4 octeți dintr-o coloană v-or fi înlocuiți cu următorii 4 octeți:

$$s`_{0,c} = (\{02\} \cdot s_{0,c}) \oplus (\{03\} \cdot s_{1,c}) \oplus s_{2,c} \oplus s_{3,c}$$

$$s`_{1,c} = s_{0,c} \oplus (\{02\} \cdot s_{1,c}) \oplus (\{03\} \cdot s_{2,c}) \oplus s_{3,c}$$

$$s`_{2,c} = s_{0,c} \oplus s_{1,c} \oplus (\{02\} \cdot s_{2,c}) \oplus (\{03\} \cdot s_{3,c})$$

$$s`_{3,c} = (\{03\} \cdot s_{0,c}) \oplus s_{1,c} \oplus s_{2,c} \oplus (\{02\} \cdot s_{3,c})$$

Figura 2.9 ilustrează operația MixColumns

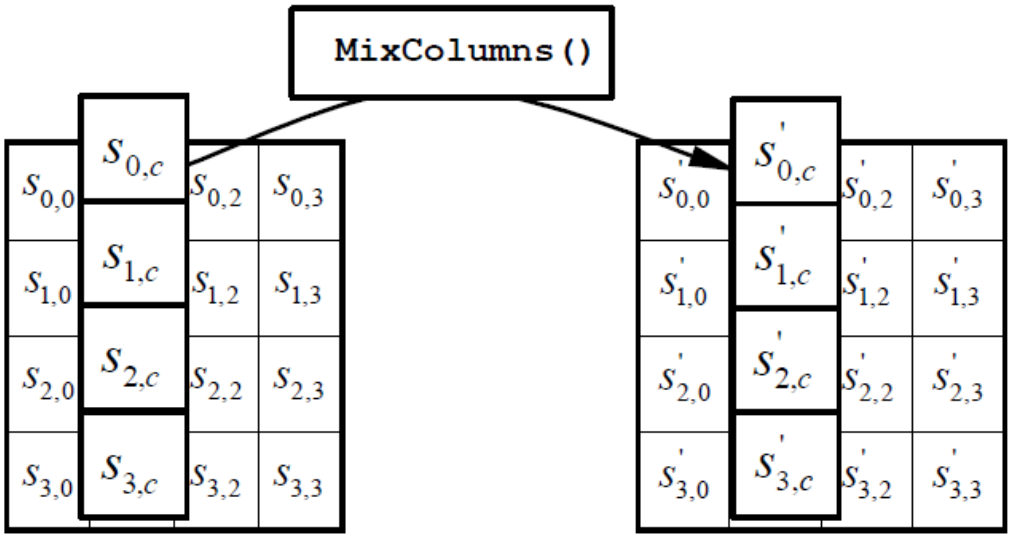

Fig. 2.9 [7]

Operația AddRoundKey

În operația AddRoundKey, o subcheie (sau o cheie de rundă) este adăugată la matricea de stare. Pentru fiecare rundă, o subcheie este derivată din cheia originală folosind algoritmul Rijndael de dispersare a cheii. Fiecare subcheie are aceeași mărime ca matricea de stare. Subcheia este adăugată la matricea de stare combinând fiecare octet al matricii de stare corespunzător fiecărui octet din subcheie folosind operația pe biți XOR. Fiecare subcheie constă în Nb cuvinte generate din algoritmul de dispersare al cheii (descris mai jos). Acele Nb cuvinte sunt adăugate în coloanele matricii de stare astfel încât

$$[s'_{0,c} \quad s'_{1,c} \quad s'_{2,c} \quad s'_{3,c}] = [s_{0,c} \quad s_{1,c} \quad s_{2,c} \quad s_{3,c}] \oplus [w_{rundă \cdot Nb+c}],\ 0 \le c < Nb,$$

unde $[w_i]$ sunt cuvintele determinate de algoritmul de dispersare al cheii, iar "rundă" este o valoare în intervalul $0 \le$ rundă $\le$ Nr. În algoritm, adunarea subcheii inițiale are loc când rundă = 0, înainte de prima aplicare a funcției de rundă. Aplicarea funcției AddRoundKey asupra rundelor are loc când cifru ajunge cu runda în intervalul $1 \le$ rundă $<$ Nr. Acțiunea acestei operații este ilustrată în Figura 2.10, unde l = rundă · Nb.

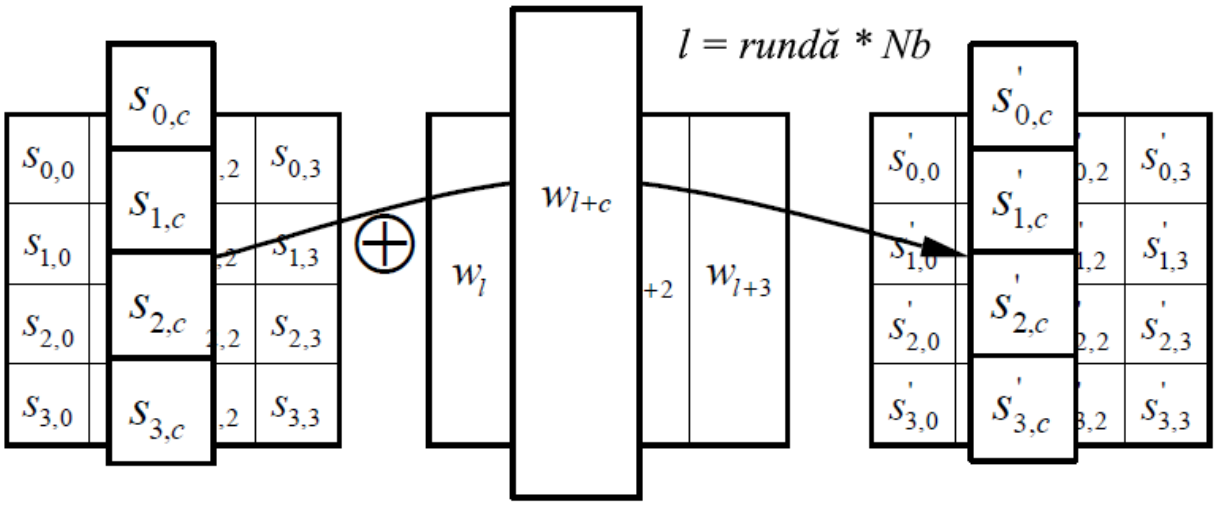

Fig. 2.10 [7]

Expansiunea cheii

Algoritmul AES, primește ca dată de intrare cheia K, și aplică o funcție asupra acestei chei pentru a genera subcheile necesare algoritmului. Algoritmul de dispersare a cheii generează în total Nb · (Nr + 1) cuvinte; algoritmul are nevoie de un set inițial Nb de cuvinte, și fiecare din cele Nr runde au nevoie de chei de lungime de Nb cuvinte. Rezultatul funcției de expansiune a cheii constă într-un vector de cuvinte de 4 octeți, notați cu [w_i], cu *i* în intervalul $0 \leq i < Nb(Nr + 1)$.

SubWord() este o funcție ce primește ca date de intrare 4 octeți, aplică tabelul de substituție S-box asupra fiecărui octet și produce ca date de ieșire 4 octeți (sau un cuvânt). Funcția RotWord() primește un cuvânt [a_0,a_1,a_2,a_3] ca date de intrare, aplică o permutare ciclică și returnează ca date de ieșire cuvântul [a_1,a_2,a_3,a_0]. Vectorul de rundă constant de cuvinte, Rcon[i], conține valorile date de [x^{i-1},{00},{00},{00}], cu x^{i-1} fiind puterile lui x (x este notat cu {02}) în GF(2^8); i pornește de la 1, și nu de la 0.

Din Figura 2.11 se observă că primele Nk cuvinte ale cheii expandate sunt inițializate cu cheia originală. Fiecare cuvânt următor, w[i], este egal cu rezultatul operației XOR aplicate asupra cuvântului anterior w[i-1] și cuvântului situat Nk poziții anterior, w[i-Nk]. Pentru cuvintele ce sunt în poziția cu numărul multiplu de Nk, o transformare este aplicată asupra w[i-1] înainte de aplicarea operației de XOR, urmată de XOR cu constanta de rundă, Rcon[i]. Această transformare constă în permutarea ciclică a octeților în cuvânt (RotWord()), urmată de aplicarea tabelului de

substituție asupra celor 4 octeți din cuvânt (SubWord()). Este important de precizat că algoritmul pentru chei de mărime de 256 biți este puțin diferit față algoritmul ce folosesc chei de lungimi 128, și respectiv 192 biți. Dacă Nk = 8 și i-4 este multiplu de Nk, atunci funcția SubWord() este aplicată cuvântului w[i-1] înaintea operației XOR.

```
void AES::KeyExpansion(unsigned char key[], unsigned int w[], int keysize)
{
    int Nb = 4, Nr, Nk;
    switch (keysize)
    {
    case 128:
    {
        Nr = 10;
        Nk = 4;
        break;
    }
    case 192:
    {
        Nr = 12;
        Nk = 6;
        break;
    }
    case 256:
    {
        Nr = 14;
        Nk = 8;
        break;
    }
    default:
    {
        return;
    }
    }
    unsigned int Rcon[] = { 0x01000000, 0x02000000, 0x04000000, 0x08000000, 0x10000000, 0x20000000, 0x40000000, 0x80000000,
                            0x1b000000, 0x36000000, 0x6c000000, 0xd8000000, 0xab000000, 0x4d000000, 0x9a000000 };
    unsigned int temp;
    for (int i = 0; i < Nk; ++i)
    {
        w[i] = ((key[4 * i]) << 24) | ((key[4 * i + 1]) << 16) | ((key[4 * i + 2]) << 8) | (key[4 * i + 3]);
    }
    for (int i = Nk; i < Nb*(Nr + 1); ++i)
    {
        temp = w[i - 1];
        if (i % Nk == 0)
        {
            temp = SubWord(RotWord(temp)) ^ Rcon[(i - 1) / Nk];
        }
        else if (Nk > 6 && (i % Nk == 4))
        {
            temp = SubWord(temp);
        }
        w[i] = w[i - Nk] ^ temp;
    }
}
```

Fig. 2.11

Decriptare

Decriptarea este funcția inversă criptării, iar în cazul AES, pentru implementarea acesteia, trebuie implementate inversele tuturor funcțiilor menționate mai sus: InvSubBytes() este funcția inversă funcției SubBytes(), InvShiftRows() este funcția inversă funcției ShiftRows(), InvMixColumns() este funcția inversă funcției MixColumns(). Funcția AddRoundKey() nu necesită o funcție inversă întrucât ea însăși este inversa deoarece implică doar aplicarea operației XOR. Două proprietăți sunt important de avut în vedere la implementarea funcției de decriptare:

1. Operațiile SubBytes() și ShiftRows() sunt comutative, aceata însemnând că o operație SubBytes() imediat urmată de operația ShiftRows() este echivalentă cu operația ShiftRows() imediat urmată de operația SubBytes(). Aceasta este adevărat și pentru inversele lor InvSubBytes() și InvShiftRows()
2. Operațiile MixColumns() și InvMixColumns() sunt lineare în concordanță cu datele de intrare ale unei coloane, asta însemnând că:

InvMixColumns(stare XOR subcheie) = InvMixColumns(stare) XOR InvMixColumns(subcheie)

Aceste două proprietăți permit ca ordinea operațiilor InvSubBytes() și InvShiftRows() să fie inversate. Ordinea operațiilor AddRoundKey() și InvMixColumns(), de asemenea pot fi inversate, aceasta numai dacă coloanele (cuvintele) funcției de expansiune a cheii este modificată folosind operația InvMixColumns(). Este indicată aplicarea acestor două proprietăți în implementare deoarece oferă o structură mai eficientă față de folosirea lor în ordinea în care s-a făcut și criptarea.

Securitate

Pentru criptografi, o "spargere criptografică" înseamnă orice este mai rapid decât un atac prin forță brută. O spargere poate consta în modalități care nu sunt posibile cu tehnologia curentă. Deși nu sunt practice, spargerile teoretice pot uneori să furnizeze informații privitoare la tiparele de vulnerabilitate. Până în prezent s-au făcut spargeri doar asupra unor algoritmi AES reduși, constând într-un număr mai mic de runde. În prezent, nu există nici o metodă practică pentru a permite unei entități neautorizate, fără a cunoaște cheia, să aibe acces la datele criptate de către algoritmul AES.

2.1.4. Moduri de operare

În criptografie, un mod de operare asupra unui algoritm criptografic de tip bloc este un algoritm ce folosește un cifru de tip bloc pentru a ascunde informațiile de către entitățile neautorizate sporind eficacitatea conceptelor de confidențialitate și autenticitate. Un cifru de tip bloc oferă o transformare criptografică sigură (criptare și decriptare) asupra unei lungimi fixe de

biți, numită bloc. Un mod de operare descrie cum se va aplica un cifru de tip bloc pentru a face operațiile de criptare, respectiv decriptare asupra datelor ce au lungime mai mare de un bloc.

Majoritatea modurilor au nevoie de o secvență binară, numită vector de inițializare, pentru fiecare operație de criptare. Vectorul de inițializare trebuie să nu se repete, iar pentru anumite moduri, trebuie să fie generat întâmplător (aleator). Vectorul de inițializare este folosit pentru a asigura că diferite texte criptate sunt produse chiar și atunci când același text în clar este criptat de mai multe ori independent folosind aceeași cheie. Cifrurile de tip bloc au unul sau mai multe lungimi de blocuri, dar în timpul unei transformări, lungimea blocului este tot mereu fixată. Moduri de operare asupra cifrurile de tip bloc necesită ca ultima parte a datelor de intrare să fie egalată (padding) cu lungimea blocului, dacă este mai mică decât lungimea blocului curent. Există de altfel moduri care nu necesită această egalare întrucât folosesc un cifru de tip bloc ca un cifru de tip flux.

Un vector de inițializare este o dată de intrare de lungime fixă pentru o primitivă criptografică ce necesită să fie aleator sau pseudo-aleator. Operația de randomizare este crucială pentru metodele de criptare pentru a obține securitate semantică, proprietate prin care folosirea repetată ale aceleiași metode cu aceeași cheie nu permite atacatorului să deducă concluzii privitor la relația dintre segmentele de mesaje criptate. Pentru cifrurile de tip bloc, folosirea unui vector de inițializare este descrisă de modul de operare. Anumite primitive criptografice necesită ca vectorul de inițializare să nu se mai fi repetat, iar generarea aleatoare necesară este calculată intern. Mărimea unui vector de inițializare este dependentă de primitiva criptografică folosită; pentru cifruri de tip bloc, deobicei este chiar de lungimea blocului cifrului.

Motivarea aplicării unui mod de operare: Un cifru de tip bloc este una din cele mai simple primitive în criptografie, și este foarte frecvent folosit pentru criptarea de date. Totuși, el însuși, poate cripta o bucată de date de o mărime predefinită, numită bloc. De exemplu, o singură invocare a algoritmului AES transformă 128 de biți de text clar în 128 de biți de text criptat. Cheia, ce este furnizată ca dată de intrare, definește relația dintre textul clar și cel criptat. Dacă un set de date de o lungime arbitrară trebuie criptată, o soluție simplă este de a împărți setul de date în blocuri egale cu mărimea blocului necesitat de algoritmul de criptare, și de a cripta fiecare bloc în parte cu aceeași cheie. Această metodă nu este sigură deoarece blocuri de texte clare egale sunt transformate în blocuri de texte criptate egale, iar o entitate neautorizată observând datele criptate

poate deduce mesajul clar chiar și dacă nu îi este cunoscută cheia de criptare. Pentru a ascunde tiparele de date criptate, evitând folosirea unei chei noi după fiecare invocare a algoritmului de tip bloc, o metodă este necesară pentru a "randomiza" datele de intrare. În 1980, NIST a publicat un document în care specifică modurile de operare asupra cifrurilor de tip bloc, fiecare descriind o soluție diferită pentru criptarea seturilor de date de intrare. Primul mod implementează soluția simplă specificată mai sus, și este numită electronic codebook (ECB). În contrast, fiecare din celelalte moduri descriu un proces prin care textul criptat dintr-un bloc de criptare este combinat cu datele pentru următorul pas de criptare. Pentru a iniția acest proces, o valoare inițială este necesară pentru a se combina cu primul bloc, această valoare inițială poartă numele de vector de inițializare. De exemplu modul cipher-block chaining (CBC) necesită o valoare arbitrară egală cu mărimea blocului algoritmului ca dată de intrare, și este adunată la primul bloc de text clar, înainte de criptările ulterioare. Astfel, textul criptat produs în primul pas de criptare este adăugat la al doilea text clar, și așa mai departe. Scopul acestei operații este de a obține securitate semantică: imposibilitatea de a trage concluzii în urma observării textului criptat.

Majoritatea metodelor de criptare de tip bloc funcționează asupra unor blocuri de lungime fixă, dar mesajul poate avea lungimi diverse. Astfel, anumite moduri (cum ar fi ECB și CBC) necesită ca ultimul bloc să fie egalat la lungimea de bloc necesară algoritmului (padding) înainte de criptare. Există diverse moduri de a face această operație. Cea mai simplă este de a introduce octeți nuli la textul clar până este adus la un multiplu de lungimea blocului necesar.

Algoritmul DES fiind cel mai cunoscut din categoria metodelor cu chei simetrice, modurile de criptare au fost special concepute pentru acest algoritm. Aceste moduri de criptare sunt folosiți și de către alți algoritmi de criptare simetrici.

Cele mai întâlnite metode de criptare sunt următoarele:

- Modul ECB (Electronic Codebook)
- Modul CBC (Cipher-Block Chaining)
- Modul CFB (Cipher Feedback)
- Modul OFB (Output Feedback)
- Modul CTR (Counter Mode)

Modul Electronic Codebook

Este cea mai simplă formă de criptare. Mesajul este împărțit în blocuri, iar fiecare bloc este criptat individual. Pentru decriptare este identic, fiecărui bloc de text criptat i se aplică funcția de decriptare.

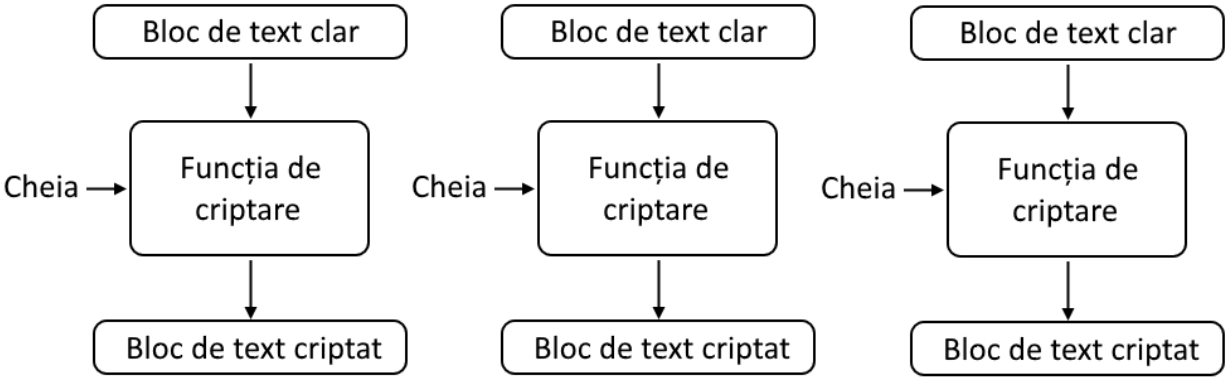

Fig. 2.12

Dezavantajul acestei metode este acela că blocurile identice de text în clar sunt criptate în blocuri identice de text criptat, astfel acest mod nu ascunde tiparele de date prea bine. Nu oferă confidențialitate, și nu este recomandată folosirea acestuia în sisteme criptografice. Un exemplu interesant pentru care modul ECB lasă tiparele de date vizibile este atunci când criptăm o imagine care are arii mari de culoare uniformă. Deși culoarea fiecărui pixel individual este criptată, imaginea per totală poate fi distinsă deoarece tiparele de pixeli colorați identici din imaginea originală, rămân și în imaginea criptată. (Figura 2.13)

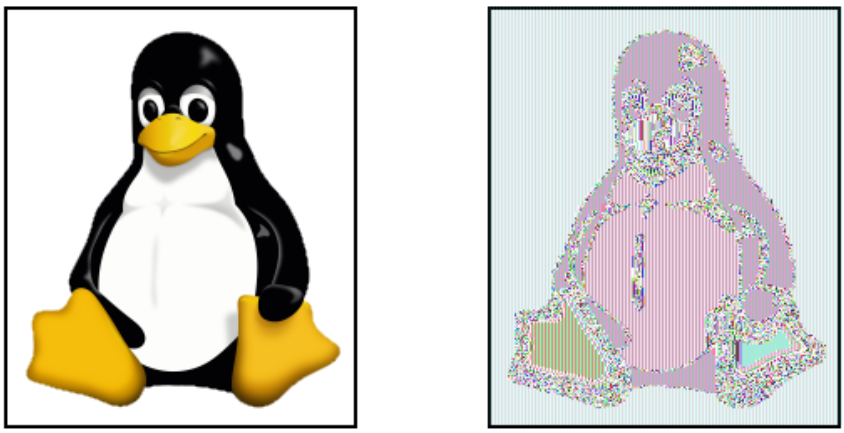

Fig. 2.13

Modul ECB poate deasemenea să facă protocoale fără protecție pentru integritate chiar mai susceptibile la atacurile de tip reluare (replay attacks), întrucât fiecare bloc este decriptat în exact același mod.

Modul Cipher Block Chaining

Modul Cipher Block Chaining (CBC) a fost un mod de operare inventat în 1976 de către Ehrsam, Meyer, Smith și Tuchman. În modul CBC, fiecărui bloc de text în clar îi este aplicată operația de XOR, cu blocul precedent de text criptat, înainte de a fi criptat. În felul acesta, fiecare bloc de text criptat depinde de toate celelalte blocuri de text în clar procesate până în acel punct. Pentru a face fiecare mesaj unic, un vector de inițializare trebuie folosit pentru primul bloc. (Figura 2.14)

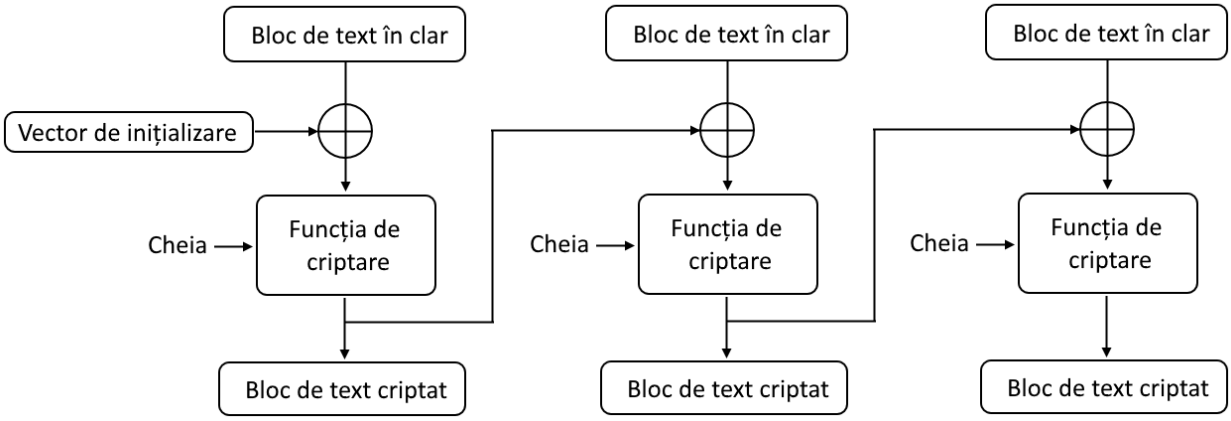

Fig. 2.14

Pentru procesul de decriptare, pașii sunt exact inverși: decriptăm primul bloc, aplicăm operația XOR pe blocul decriptat și vectorul de inițializare și va rezulta primul bloc de text în clar. La pasul următor se va aplica operația de XOR între blocul decriptat și blocul de text criptat înainte să i se aplice operația de decriptare din pasul anterior. (Figura 2.15)

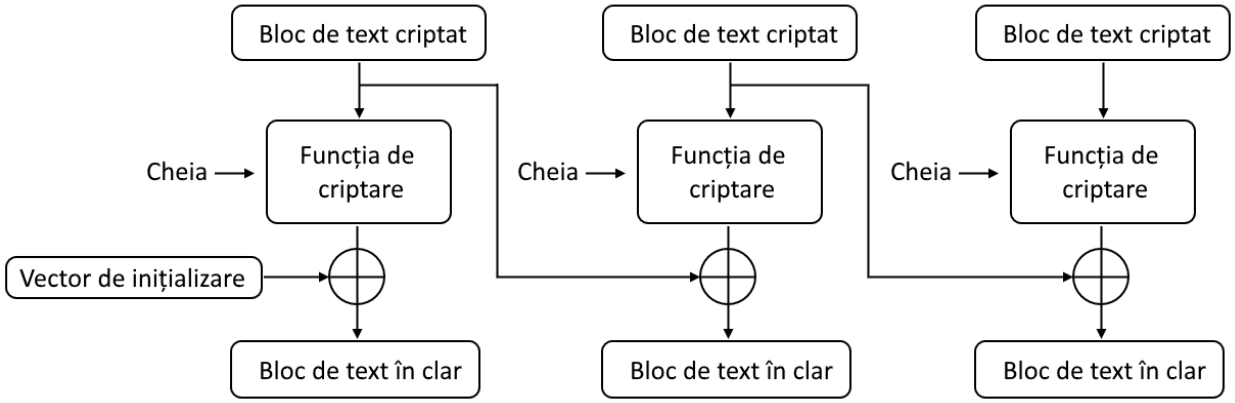

Fig. 2.16

Dacă primul bloc are indexul 1, atunci formula matematică pentru criptarea CBC este:

$C_i = E_K (P_i \oplus C_{i-1})$, $C_0 = IV$, unde C_i reprezintă blocul criptat curent, E_K este funcția de criptare, P_i este blocul în clar curent, și IV este vectorul de inițializare.

Păstrând aceleași notații, formula matematică pentru decriptarea CBC este:

$P_i = D_K (C_i) \oplus C_{i-1}$, $C_0 = IV$.

CBC a fost cel mai folosit mod de operare. Dezavantajele principale pe care îl are este acela că funcția de criptare este secvențială, adică nu poate fi paralelizată, și că mesajul trebuie adus la un multiplu de blocul cifrului. Decriptarea cu vectorul de inițializare incorect produce ca primul bloc de text în clar să fie corupt, dar blocurile ulterioare de text în clar vor fi corecte. Asta se întâmplă datorită faptului că fiecărui bloc îi este aplicată operația XOR cu textul criptat din blocul anterior, și nu acela al textului în clar, astfel nu trebuie decriptat blocul anterior ca acesta să poată fi folosit ca și vector de inițializare pentru funcția de decriptare al blocului curent. Aceasta înseamnă că un bloc de text în clar poate fi recuperat din două blocuri adiacente de text criptat. Ca și consecință, decriptarea poate fi paralelizată. De observat faptul că, schimbarea unui singur bit a textului criptat produce coruperea totală a blocului corespondent de text în clar, și inversează bitul corespunzător în următorul bloc de text în clar, dar restul blocurilor rămân intacte. Această particularitate a fost exploatată în diferite atacuri de tip “padding oracle attacks”, cum ar fi POODLE.

Modul Cipher Feedback

Modul Cipher Feedback (CFB) este asemănător modului CBC. Acesta transformă un cifru de tip bloc într-un cifru de tip flux cu auto-sincronizare. Operația este foarte similară; în particular, decriptarea CFB este aproape identică cu criptarea CBC efectuată în sens invers:

$$C_i = E_K(C_{i-1}) \oplus P_i, \quad P_i = E_K(C_{i-1}) \oplus C_i, \quad C_0 = IV.$$

În figurile 2.17 și 2.18 se observă criptarea, respectiv decriptarea modului CFB.

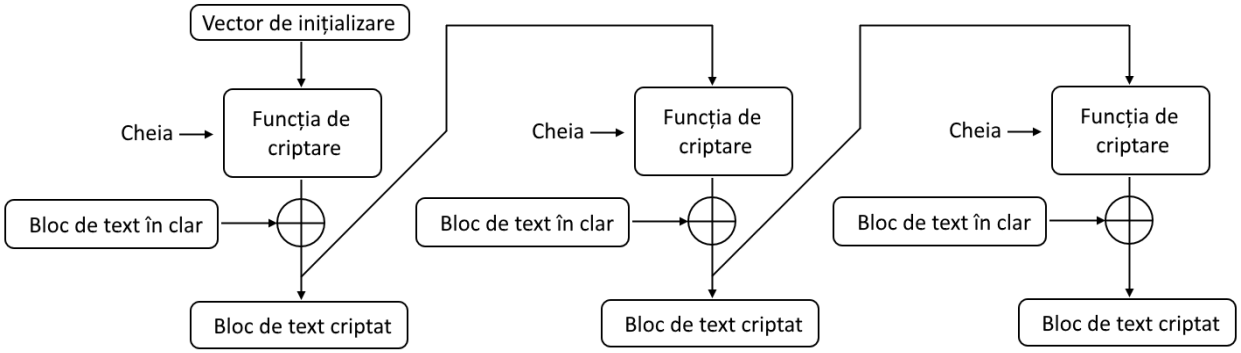

Fig. 2.17

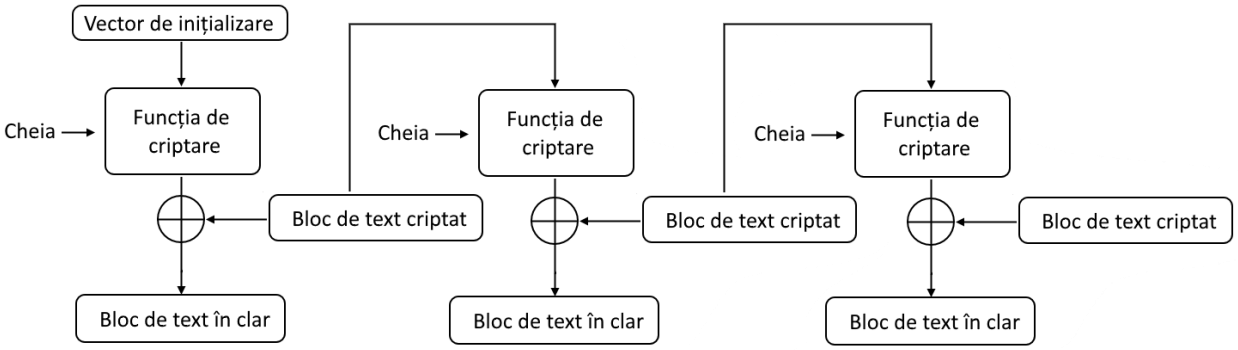

Fig. 2.18

Prin definiție cifrului cu auto-sincronizare, dacă o parte a textului criptat este pierdut (de exemplu datorită erorilor de transmisie), atunci receptorul va pierde doar o parte din mesajul original, și ar trebui să continue decriptarea în mod corect pentru restul blocurilor după ce s-a procesat o anumită cantitate de date de intrare. Dacă un întreg bloc de text criptat este pierdut, atunci ambele moduri, CBC și CFB se vor sincroniza, dar dacă se pierde un singur octet sau bit, atunci funcția de decriptare va fi permanent desincronizată. Pentru a sincroniza după pierderea unui singur octet sau bit, trebuie ca un singur octet sau bit să fie criptat la un moment dat. CFB poate fi folosit în acest

mod când este combinat cu registru de deplasare (shift register) ca și date de intrare pentru cifrul bloc.

Pentru a folosi CFB pentru a face un cifru de tip flux auto-sincronizator ce va sincroniza orice multiplu de x biți pierduți, trebuie inițializat registrul de deplasare de mărimea blocului cu vectorul de inițializare. Acesta va fi criptat cu cifrul de tip bloc și celor mai din stânga x biți ai rezultatului, li se aplică operația de XOR cu x biți din textul clar pentru a produce x biși în text cifrat. Acești x biți de ieșire sunt deplasați intr-un registru de deplasare, iar procesul se repetă cu următorii x biți de text în clar. Decriptarea este similară, se începe cu vectorul de inițializare, se criptează, și se aplică operația de XOR asupra biților celor mai din stânga ai rezultatului cu x biți din textul cifrat pentru a produce x biți de text în clar. După aceea se deplasează cei x biți de text cifrat în registrul de deplasare. Această procedură este cunoscută ca și CFB-8 sau CFB-1 (depinzând de mărimea deplasării).

Ca și notație, S_i este cea de-a i-a stare al registrului de deplasare, "$<< x$" reprezintă deplasarea la stânga cu x biți, "head(a, x)" reprezintă cei mai din stânga x biți a lui a și n reprezintă numărul de biți al vectorului de inițializare:

$$C_i = \text{head}\ (\ E_K\ (\ S_{i-1}\ ),\ x\ ) \oplus P_i$$

$$P_i = \text{head}\ (\ E_K\ (\ S_{i-1}\ ),\ x\ ) \oplus C_i$$

$$S_i = (\ (\ S_{i-1} << x\ ) + C_i\ ) \bmod 2^n$$

$$S_0 = IV$$

Dacă x biți sunt pierduți din textul criptat, cifrul va produce ca date de ieșire un text în clar incorect până când registrul de deplasare va fi din nou egal cu starea ce o deținea când se făcea criptarea, moment în care cifrul s-a resincronizat. Aceasta va rezulta în cel mult un bloc de date de ieșire pierdut.

Ca și modul CBC, schimbările ce au loc în textul clar se propagă la nesfârșit în textul criptat, și criptarea nu poate fi paralelizată. De asemenea ca și modul CBC, decriptarea poate fi paralelizată. Când se face decriptarea, o schimbare de un singur bit în textul criptat afectează două blocuri de text în clar: o singur bit schimbat în textul clar corespunzător, provoacă corupție totală asupra blocului următor de text clar. Blocurile ulterioare de text clar se vor decripta normal.

Modul CFB are două avantaje comune cu modul OFB și CTR asupra modului CBC: cifrul de tip bloc este folosit doar în direcția de criptare, și mesajul nu trebuie adus la un multiplu de blocul cifrului.

Modul Output Feedback

Modul Output Feedback (OFB) transformă un cifru de tip bloc într-un cifru de tip flux sincronic. Acesta generează blocuri de chei fluide, cărora li se aplică operația XOR cu blocurile cu text în clar pentru a obține textul criptat. Ca și cu alte cifruri de tip flux, schimbarea unui bit în textul criptat produce o schimbare a bitului în textul în clar, în aceeași locație. Această proprietate permite multor coduri de corecție a erorilor să funcționeze normal chiar și când sunt aplicate înainte de criptare. Datorită simetriei operației XOR, criptarea și decriptarea sunt identice:

$$C_j = P_j \oplus O_j$$

$$P_j = C_j \oplus O_j$$

$$O_j = E_K (I_j)$$

$$I_j = O_{j-1}$$

$$I_0 = IV$$

Fiecare dată de ieșire a modului OFB depinde de toate cele anterioare, astfel nu poate fi efectuat în paralel. Totuși, deoarece textul în clar și textul criptat sunt folosite doar pentru operația finală XOR, operațiile cifrului bloc pot fi efectuate în avans, permițând ca ultimul pas să fie efectuat în paralel de îndată ce textul în clar sau cel criptat sunt disponibile.

Este posibil de a obține o cheie fluidă pentru modul OFB folosind modul CBC cu un șir constant de zero-uri ca și date de intrare. Aceasta poate fi utilă deoarece permite folosirea hardware-urilor pentru implementările rapide ale modului CBC pentru funcția de criptare a modului OFB.

În figurile 2.19 și 2.20 se observă criptarea, respectiv decriptarea modului OFB.

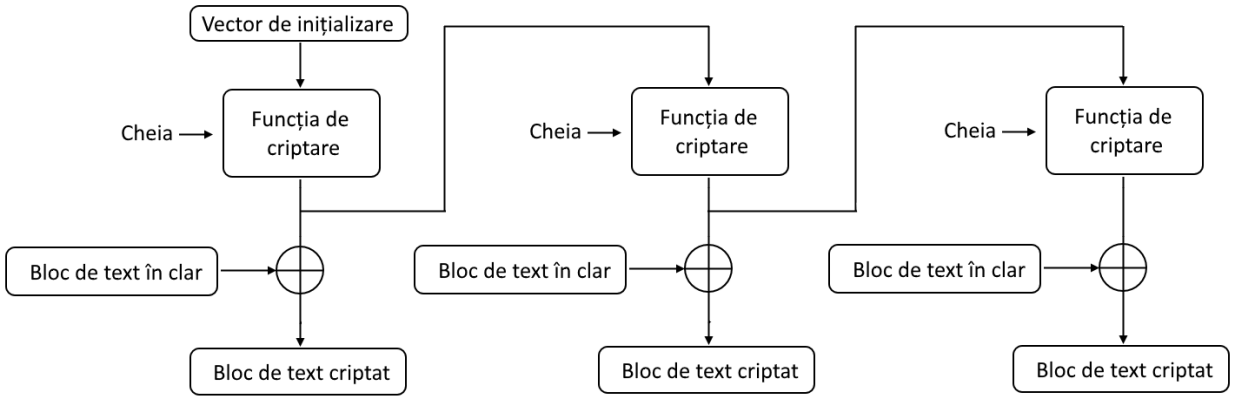

Fig. 2.19

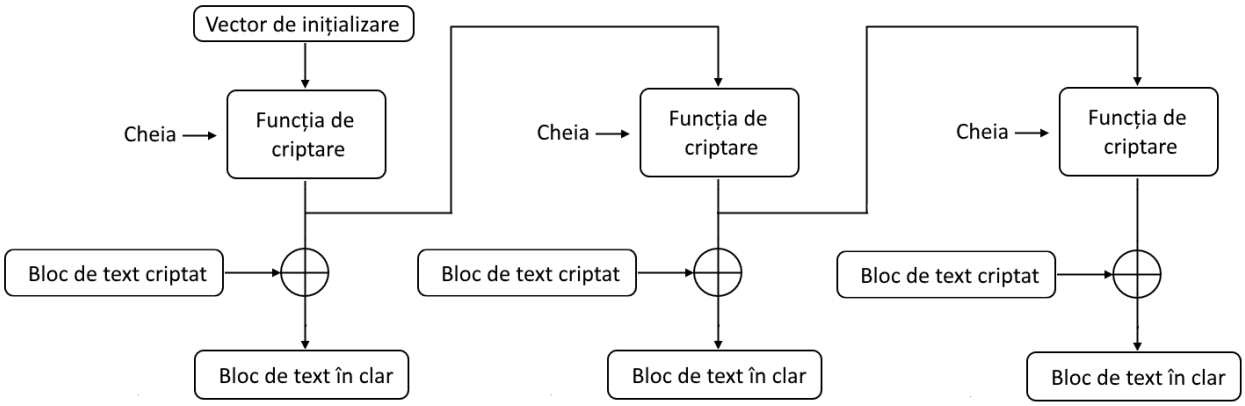

Fig. 2.20

Modul Counter (CTR)

Counter Mode (CM), este cunoscut și ca integer counter mode (ICM) și segmented integer counter (SIC).

Ca și modul OFB, CTR transformă un cifru de tip bloc într-unul de tip flux. Generează următorul bloc de cheie fluidă prin criptarea succesivă a valorilor unui "contor". Contorul poate fi orice funcție ce produce o secvență ce garantează să nu se mai repete, cu toate că un contor ce crește valoarea cu unu este cel mai simplu și cel mai popular. Folosirea unei funcții simplu deterministe obișnuia să fie controversată; criticii susțineau că "expunerea intenționată a unui criptosistem către date de intrare sistematice știute, reprezentau un risc inutil". În orice caz, astăzi, modul CTR este acceptat pe scară largă, iar orice probleme sunt considerate o slăbiciune a cifrului

bloc ce stă la baza modului, care este de așteptat să fie sigur indiferent de datele de intrare sistematice. Împreună cu CBC, modul CTR este unul din cele două moduri pentru cifrurile de tip bloc recomandate de Niels Ferguson și Bruce Schneier. Modul CTR a fost introdus de Whitfield Diffie și Martin Hellman în 1979.

Modul CTR are caracteristici similare cu modul OFB, dar de asemenea oferă proprietatea de acces direct la memorie pentru decriptare. Modul CTR este foarte potrivit pentru funcționarea sa pe o mașină cu mai mult de un procesor, unde blocurile pot fi criptare și decriptare în paralel.

Dacă vectorul de inițializare este arbitrar, atunci acesta poate fi combinat împreună cu contorul folosind operații fără pierderi (concatenarea, adunarea, operația XOR) pentru a produce contorul unic de bloc pentru criptare. În cazul unui vector de inițializare nearbitrar, acesta are trebui concatenat (de exemplu stocând vectorul în primii 64 de biți și contorul în ultimii 64 de biți dintr-un contor de bloc de 128 biți). Pur și simplu adăugând sau aplicând operația de XOR asupra vectorului de inițializare și a contorului pentru a obține o singură valoarea, va risca securitatea, sub unui atac de corespondență text clar – text criptat, în multe cazuri, datorită faptului că atacatorul poate manipula întreaga pereche vector-contor pentru a crea o coliziune. Odată ce atacatorul controlează perechea vector-contor și textul în clar, aplicând operația de XOR asupra textului criptat cu textul în clar știut va da o valoare care, atunci când îi este aplicată operația XOR cu textul criptat al blocului ce au în comun aceeași pereche vector-contor, va decripta acel bloc.

În figurile 2.21 și 2.22 se observă criptarea, respectiv decriptarea modului CTR:

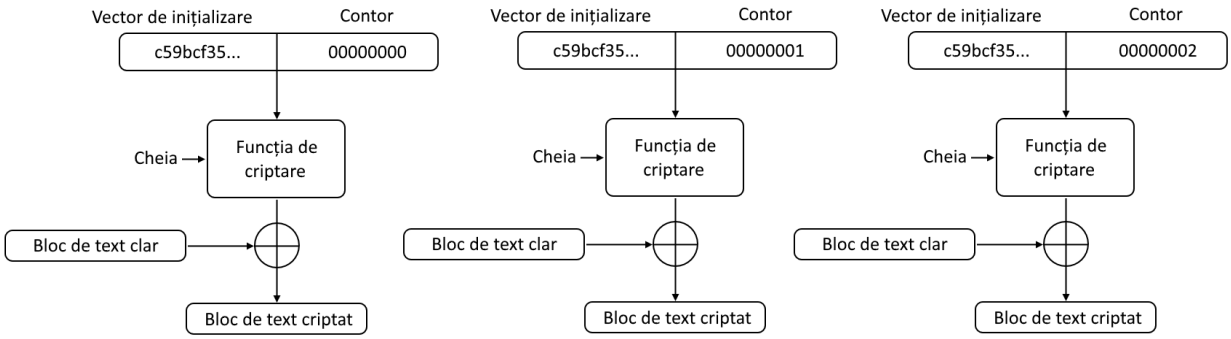

Fig. 2.21

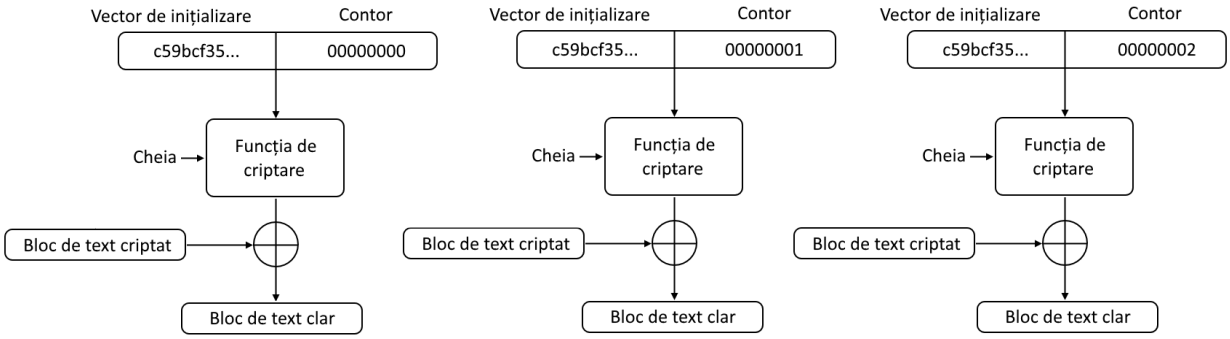

Fig. 2.22

În tabelul de mai jos este prezentat fiecare mod de operare cu proprietățile dacă pot operațiile de criptare, respectiv decriptare fi paralelizate și dacă citirea aleatorie este permisă.

	ECB	CBC	CFB	OFB	CTR
Criptarea paralelizabilă	✓	×	×	×	✓
Decriptarea paralelizabilă	✓	✓	✓	×	✓
Citire aleatorie	✓	✓	✓	×	✓

Propagarea erorilor

Înainte de răspândirea pe scară largă a codurilor pentru mesajele de autentificare și a criptărilor cu autentificare, era obișnuit să se analizeze propagarea erorilor ca proprietate ca și criteriu în selecția unui mod de operare. Putea fi observat, spre exemplu, că o eroare a unui bloc în textul cifrat va rezulta într-o eroare a unui bloc în textul în clar pentru modul ECB, în timp ce pentru modul CBC, o asemenea eroare afectează două blocuri. Unii considerau că o asemenea rezistență era de dorit în fața erorilor aleatoare, în timp ce alții argumentau că prin corectarea erorilor crește vizibilitatea atacatorilor de a interfera cu mesajul. În orice caz, când protecția adecvată pentru integritate este folosită, o asemenea eroare va rezulta în întreg mesajul să fie respins. Dacă este dorită rezistența împotriva erorilor aleatoare, coduri de corectare a erorilor ar trebui să fie aplicate textului de criptare înainte de transmisie.

2.2. Metode criptografice de tip flux

Un cifru de tip flux este un cifru cu cheie simetrică unde fiecare cifră al textului în clar este combinat cu o cheie fluidă (key stream) pseudo-aleatoare. Într-un cifru de tip flux, fiecare cifră din textul clar este criptat câte unul la un moment dat împreună cu cifra corespondentă din cheia fluidă, pentru a obține cifra corespondentă în textul criptat. Deoarece criptarea fiecărei cifre este dependentă de starea curentă a cifrului, cifrul mai este cunoscut și ca cifru de stare (state cipher). În practică, prin cifră ne referim la un bit, iar operația de combinare a textului în clar cu cheia este XOR.

Cheia fluidă pseudo-aleatoare este deobicei generată pornind de la un număr aleator (seed) folosind regiștrii de permutare. Valoarea seedului reprezintă cheia criptografică pentru decriptarea fluxului de text criptat. Cifrurile de tip flux reprezintă o abordare diferită pentru criptarea simetrică față de cifrurile de tip bloc. Cifrurile de tip bloc funcționează pe blocuri fixe de o lungime mare. Cu ajutorul anumitor moduri de operare, o primitivă de tip bloc poate fi folosită în așa mod încât să funcționeze ca un cifru de tip flux. Cifurile de tip flux deobicei se execută la viteze mult mai mari decât cifrurile de tip bloc și au o complexitate hardware mai scăzută. În orice caz, cifrurile flux pot fi susceptibile la probleme de securitate dacă sunt folosite incorect: în particular, același seed nu trebuie să fie vreodată folosit de două ori.

One-time pad

Este cunoscut și ca cifrul Vernam. One-time pad folosește o cheie fluidă formată din cifre complet aleatoare. Cheia este combinată cu cifrele textului clar, câte unul la un moment dat, pentru a forma textul criptat. Acest sistem a fost demonstrat să fie sigur din punct de vedere criptografic de către Claude Shannon în 1949. Însă, cheia fluidă trebuie să fie generată complet aleator cu mărimea de cel puțin egală cu mărimea mesajului ce se dorește a fi criptat *și* nu trebuie folosit mai mult de o singură dată. Aceasta face ca sistemul să fie foarte greu de implementat în multe aplicații practice, și prin urmare, one-time pad nu a fost folosit pe scară largă, cu excepția aplicațiilor celor mai critice. Generarea, distribuirea și administrarea cheii sunt critice pentru acele aplicații.

Un cifru de tip flux se folosește de o cheie de mărime mult mai mică și mai convenabil de pus în practică, cum ar fi o cheie de 128 biți. Pe baza acestei chei, se generează pseudo-aleator o cheie fluidă pentru a putea fi combinată cu textul clar într-un mod similar cu one-time pad. Totuși,

aceasta vine cu un cost, și anume cheia fluidă este pseudo-aleatoare și nu este cu adevărat aleatoare. Dovada de securitate asociată cu one-time pad nu mai este valabilă. Este foarte posibil ca un cifru flux să fie complet nesigur din punct de vedere criptografic.

Tipuri de cifruri flux

Un cifru flux generează elemente succesive ale cheii fluide bazate pe o stare internă. Această stare este actualizată în două moduri: dacă starea se schimbă independent de textul clar sau de textul criptat, cifrul este clasificat ca și cifru flux sincron. În contrast, cifruri flux auto-sincrone își actualizează starea bazându-se pe cifre ale textului criptat anterioare.

Cifru flux cu sincronizare

Într-un cifru flux sincron, un flux de cifre pseudo-aleatoare sunt generate independent de textul clar sau de textul cifrat, apoi sunt combinate cu textul clar (pentru criptare) sau textul criptat (pentru decriptare). În cea mai simplă formă, cifrele binare, și anume biții sunt folosiți, iar cheia fluidă este combinată cu textul în clar folosind operația XOR. Aceasta este denumit ca cifru flux aditiv binar.

Într-un cifru flux sincron, emițătorul și receptorul trebuie să fie exact coordonați pentru ca decriptarea să aibe succes. Dacă se adaugă sau se elimină biți din mesaj în timpul transmisiei, sincronizarea este pierdută. Pentru a restabili sincronizarea, diferite decalaje pot fi încercate sistematic pentru a obține decriptarea corectă. O altă abordare este aceea de a marca textul criptat în diferite puncte la datele de ieșire.

Dacă, în orice caz, un bit este corupt în timpul transmisiei, și nu adăugat sau eliminat, doar un singur bit în textul în clar va fi afectat, iar eroare nu se propagă la ceilalți biți din mesaj. Această proprietate este utilă când rate de eroare la transmisie este mare. În plus, datorită acestei proprietăți, cifrurile flux sincrone sunt foarte susceptibile la atacuri active: dacă o entitate neautorizată poate schimba un bit în textul clar, de asemenea poate face schimbări anticipate corespunzător bitului din textul clar; de exemplu, schimbând valoarea unui bit în textul criptat, provoacă schimbarea aceluiași bit în textul clar.

Cifru flux cu auto-sincronizare

O altă abordare este de a folosi câteva din cei *n* biți criptați pentru a produce cheia fluidă. Această metodă este cunoscută și ca cifruri flux auto-sincrone. Avantajul acesteia este că receptorul va sincroniza automat cu generatorul de cheie fluidă după primirea celor *n* biți de text criptat, făcând mult mai ușoară recuperarea biților ce au fost eliminați sau adăugați în fluxul de mesaj. Erorile de un singur bit au efect limitat, în sensul că afectează doar până la *n* biți de text clar. Un exemplu de cifru flux auto-sincron este un cifru bloc folosit în modul cipher feedback.

2.2.1. Rivest Cipher 4 (RC4)

Rivest Cipher 4 sau RC4 este un cifru de tip flux. Cu toate că este impresionant pentru simplicitatea și viteza pe care o oferă în software, multiple vulnerabilități au fost descoperite în acest cifru făcându-l nesigur din punct de vedere criptografic. Este vulnerabil în special atunci când începutul cheii fluide de ieșire nu este înlăturată, sau când se folosesc chei nealeatoare. Din 2015, există o speculație cum că anumite agenții posedă capabilitatea de a sparge RC4 când acesta este folosit în protocoale TLS. IEFT (Internet Engineering Task Force) a publicat documentul RFC 7465 pentru interzicerea folosirii cifrului RC4 în protocoale TLS; Mozilla și Microsoft au emis recomandări similare.

RC4 a fost proiectat de Ron Rivest din cadrul RSA Security în 1987. Inițial RC4 a fost un secret comercial, dar în Septembrie 1994 o descriere a sa a fost anonim publicată. RSA Security nu a emis niciodată algoritmul, însă Rivest a confirmat istoria cifrului RC4 și codul acestuia într-un articol din 2014 scris de el.

Descrierea algoritmului

RC4 generează un flux de biți pseudo-aleator. Ca oricare alt cifru de tip flux, acesta poate fi folosit pentru criptarea prin aplicarea operației XOR asupra biților de text clar cu biții din cheia fluidă. Decriptarea este efectuată în același mod, deoarece operația XOR este chiar însăși inversa. Acest procedeu este similar cu cifrul Vernam cu excepția faptului că biții din cheia fluidă sunt generați pseudo-aleator. Pentru a genera cheia fluidă, cifrul se folosește de o stare internă secretă ce constă în două părți:

1. O permutare a tuturor celor 256 octeți posibili (notată cu *S*)

2. Doi octeți, indicatori de index (notați cu *i* și *j*)

Permutarea este inițializată cu o variabilă de lungimea cheii, deobicei între 40 și 2048 biți, folosind algoritmul de dispersare al cheii (key-scheduling algorithm (KSA)). De îndată ce acesta s-a terminat, fluxul de biți este generat folosind algoritmul de generare pseudo-aleator (pseudo-random generation algorithm (PRGA)).

Algoritmul de dispersare al cheii este folosit pentru a inițializa permutarea în vectorul *S*. Lungimea cheii este definită ca fiind numărul de octeți pe care cheia îl are și poate fi în intervalul [1, 256], dar deobicei este în intervalul [5, 16] corespunzător unei lungimi de chei între 40 și 128 de biți. La început, vectorul S este inițializat cu permutarea identică. După aceea, *S* este procesat pentru 256 de iterații, într-un mod similar cu algoritmul de generare pseudo-aleator, dar combină și octeți din cheie în același timp.

Pentru numărul total de iterații necesare (egal cu lungimea mesajului), algoritmul de generare pseudo-aleator modifică starea și scoate ca dată de ieșire un octet al cheii fluide. În fiecare iterație, algoritmul crește valoarea lui *i* cu 1, caută al *i*-lea element din vectorul *S*, S[i], și adaugă această valoare la valoarea lui *j*, după care se interschimbă valorile S[i] cu S[j], iar apoi se folosește suma S[i] + S[j] modulo 256 ca index pentru a obține un al treilea element *K* (valoarea propriu-zisă a cheii fluide). În final se aplică operația XOR pe acesta cu următorul octet din mesaj pentru a produce următorul octet din textul în clar sau textul criptat. Fiecare element *S* este interschimbat cu un alt element de cel puțin o dată la fiecare 256 de iterații.

Figura 2.23 ilustrează modul de funcționare al algoritmului de generare pseudo-aleator.

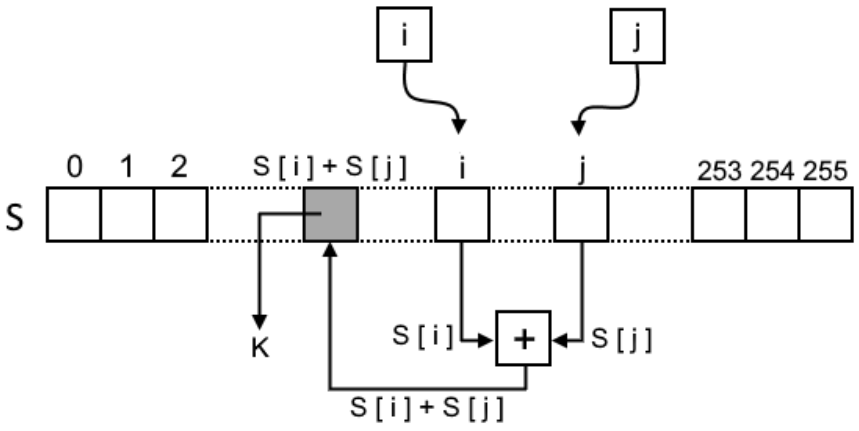

Fig. 2.23

Figura 2.24 ilustrează modul de funcționare a cifrului RC4, cât și unde intră în schemă algoritmii precizați mai sus.

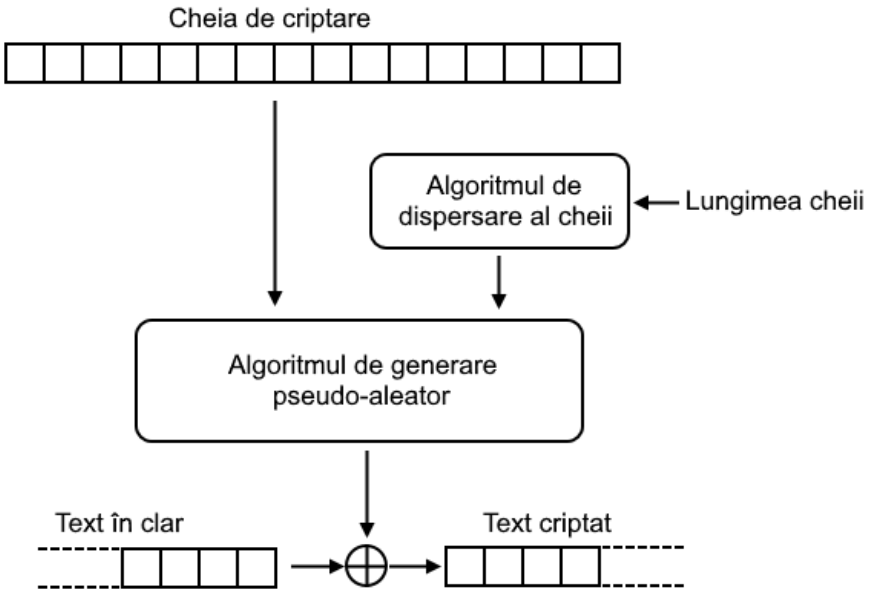

Fig. 2.24

Multe cifruri de tip flux sunt bazate pe regiștrii de permutare cu feedback liniar, care deși sunt eficienți în hardware, nu sunt eficienți și în software. RC4 a fost gândit să evite folosirea acestor regiștrii, și este ideal de implementat în software, deoarece necesită doar manipulări la nivel de octet. Folosește 256 octeți de memori pentru vectorul de stare, de la S[0] la S[255], *k* octeți de memori pentru cheie, de la k[0] la k[k-1], și variabilele de tip întreg i, j și K.

Securitate

Spre deosebire de cifrurile de tip flux moderne, RC4 nu primește ca dată de intrare un vector de inițializare împreună cu cheia. Aceasta înseamnă că dacă o cheie folosită pentru a cripta mai multe fluxuri de date, protocolul trebuie să specifice cum aceasta trebuie combinată cu vectorul de inițializare pentru a genera cheia fluidă pentru RC4. O abordare pentru aceasta este de a genera o cheie RC4 nouă prin a aplica o funcție de hash asupra cheii, ce se va dori a se folosi pentru mai multe fluxuri, și vectorul de inițializare. Deși, multe aplicații ce folosesc RC4 doar concatenează cheia cu vectorul de inițializare. Algoritmul slab de dispersare al cheii al cifrului RC4 dă naștere atacurilor asupra cheii, cum ar fi atacul Fluhrer, Mantin și Shamir, ce a spart standardul WEP.

Deoarece RC4 este un cifru flux, este mai vulnerabil decât unele cifruri bloc de tip bloc. Dacă nu este folosit împrepună cu un cod de autentificare a mesajului, atunci criptarea este vulnerabilă la atacuri de schimbare a biților. Este de menționat faptul că, deși RC4 este un cifru flux, pentru o perioadă îndelungată a fost singurul cifru simplu ce era imun la atacul BEAST din 2011 asupra protocolului TLS 1.0. Folosirea algoritmului RC4 în protocoale TLS este interzisă prin documentul RFC 7465 publicat în februarie 2015. Cea mai mare slăbiciune a cifrului RC4 provine din mărimea prea mică a cheii dispersate; primii octeți expun informații legat de cheie. Aceasta poate fi corectată prin simpla eliminare a porțiunii inițiale din șirul de ieșire. Această tehnică este cunoscută ca RC4-drop*N*, unde *N* este deobicei un multiplu de 256.

Un număr de încercări s-a făcut pentru a îmbunătăți algoritmul RC4, de menționat sunt algoritmii Spritz, RC4A, VMPC și RC4+.

Capitolul 3

Prezentarea aplicației

Aplicația este dezvoltată integral în limbajul C++, și folosește librăria SFML pentru desenarea și administrarea ferestrelor. Aplicația constă atât în implementarea algoritmilor de criptare DES, 3DES, AES și RC4 cât și în implementarea modurilor ECB și CBC pentru cifrurile de tip bloc. Scopul aplicației este de a primi un fișier ca dată de intrare, și de al cripta cu algoritmul, respectiv modul de operare ales.

Prezentarea interfeței aplicației

Aplicația se prezintă cu o intefață grafică (Figura 3.1), unde avem mai multe butoane, și o casetă de text. Pe dreapta, avem un set de patru butoane ce corespund celor 4 cifruri implementate (DES, 3DES, AES, RC4). Sub acestea avem două butoane corespunzătoare modurilor de operare ECB și respectiv CBC, ce apar numai în cazul selectării unui algoritm de tip bloc, deoarece doar asupra algoritmilor de tip bloc are sens aplicarea acestor moduri de operare. În partea de jos a interfeței observăm butoanele de aplicare funcției de criptare, respectiv decriptare ce aplică funcția corespunzătoare butonului și algoritmului selectat. De altfel, sub butonul de criptare, există un buton adițional numit “Pixeli” care atunci când este selectat, funcția de criptare înloc să primească ca date de intrare șirul de octeți, primește valorile șirului de pixeli dintr-o imagine. Evident aplicarea acestui buton are sens doar când se dorește a se cripta imagini, întrucât doar acestea pot fi reprezentate sub formă de pixeli. Motivul introducerii acestui buton este de a evidenția diferența dintre aplicarea funcției de criptare cu modul de operare ECB, respectiv CBC asupra aceleiași poze. Sub butonul de criptare, există un buton numit “.ext”, prescurtare de la extensie, iar acesta are scopul de a adăuga, la finalul numelui unui fișier decriptat, extensia sa originală. Deasupra butoanelor menționate există o caseta de text ce are ca scop introducerea căii către fișierul ce se dorește a fi criptat. Deasupra acestei casete de text va apărea, în caz de criptare pe pixeli a unei imagini, imaginea originală, criptarea sa cu modul de operare ECB, și respectiv modul CBC.

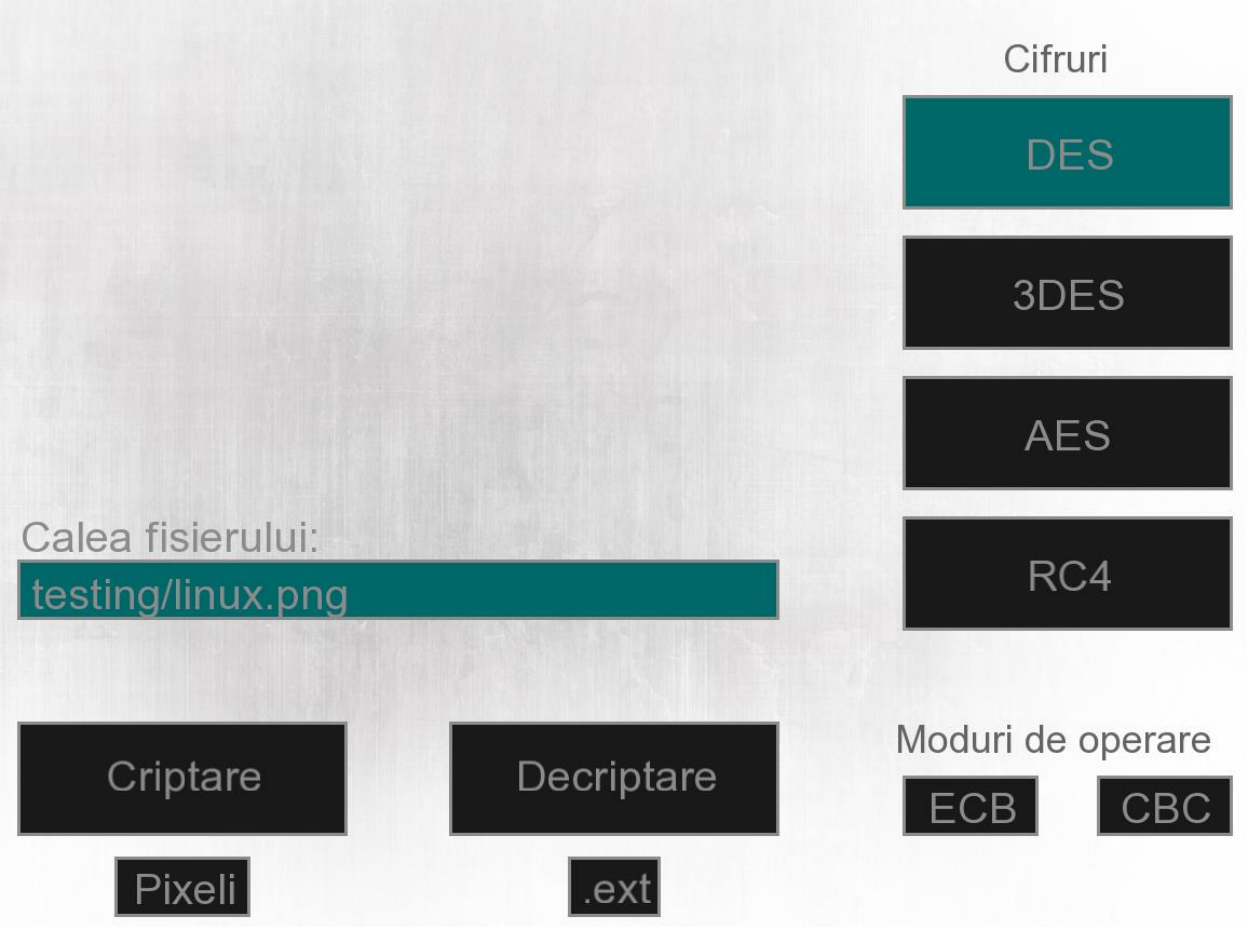

Fig. 3.1

Prezentarea internelor aplicației

Toate cifrurile au trei date de intrare: numele fișierului ce se dorește a fi criptat, numele fișierului criptat, și numele fișierului de unde se procură cheia. Fiecare cifru are fișierul său special de procurat cheia sub forma "<numele_cifrului>Key.txt". De exemplu, algoritmul AES își procură cheia din fișierul "AESKey.txt", RC4 din fișierul "RC4Key.txt" și așa mai departe. Pentru modificarea cheii unui algoritm se va deschide fișierul text corespunzător al algoritmului și se va scrie cheia în acesta, sau se va redenumi un fișier în modul precizat anterior. Formatul unui fișier nu contează atâta timp cât acesta respectă notația menționată mai sus, întrucât indiferent de formatul fișierului, aplicația citește primii octeți necesari cheii corepunzătoare algoritmului, iar acei octeți vor fi cheia efectivă cu care se va face criptarea, respectiv decriptarea.

Fiecare algoritm implementat este situat în propria clasă, unde după caz, sunt definite tabelele de substituție, tabelele de permutare, mărimea cheii, respectiv mărimea blocului și alte caracteristici specifice fiecărui algoritm în parte. Implementarea operațiilor algoritmilor este modulară în sensul că fiecare operație este implementată separat într-o funcție, ca atunci când "asamblăm" algoritmul să se poată vedea foarte clar cum sunt apelate funcțiile corespunzător descrierilor lor din capitolul anterior. De exemplu, în Figura 3.2 se observă structura algoritmului DES; mai întâi se aplică algoritmul de dispersare al cheii (KeySchedule()), după care se face Permutarea inițială (InitialPermutation()), se împarte mesajul (SplitMessage()), pentru fiecare rundă vom aplica internele funcției Feistel, adică Expansiunea (ExpansionInF()), operația de XOR din interiorul funcției Feistel, evaluarea cutiilor S (S-boxes) cu funcția SBoxComputation(), aplicarea permutării (PermutationInF()), operația de XOR din exteriorul funcției Feistel, încrucișarea părții stângi cu partea dreaptă pentru runda următoare (CrissCross()), iar după cele 16 runde, se va face o interschimbare finală între cele două părți, se va aplica Permutarea Inversă (InversePermutation()) și se returnează blocul criptat (MsgAfterInversePermutation)

```
long long int DES::Encipher()
{
    KeySchedule();
    InitialPermutation();
    SplitMessage();
    for (int round = 0; round < 16; ++round)
    {
        ExpansionInF();
        XorInFeistelNetwork = RightAfterExpansion ^ SubKeys[round];
        SBoxComputation();
        PermutationInF();
        XorOutOfFeistelNetwork = Left ^ RightAfterPermutation;
        CrissCross();
    }
    SwapLeftRight();
    InversePermutation();
    return MsgAfterInversePermutation;
}
```

Fig. 3.2

Capitolul 4

Rezultate și concluzii

Rezultate

Rezultatele aplicației constau în obținea fișierului criptat. De exemplu, în Figura 4.1, observăm un fișier text cu un mesaj înauntru, mai jos cum arată acesta în varianta criptată, iar apoi varianta decriptată a acestuia. Criptarea în acest caz, s-a făcut cu algoritmul AES, modul de operare ECB.

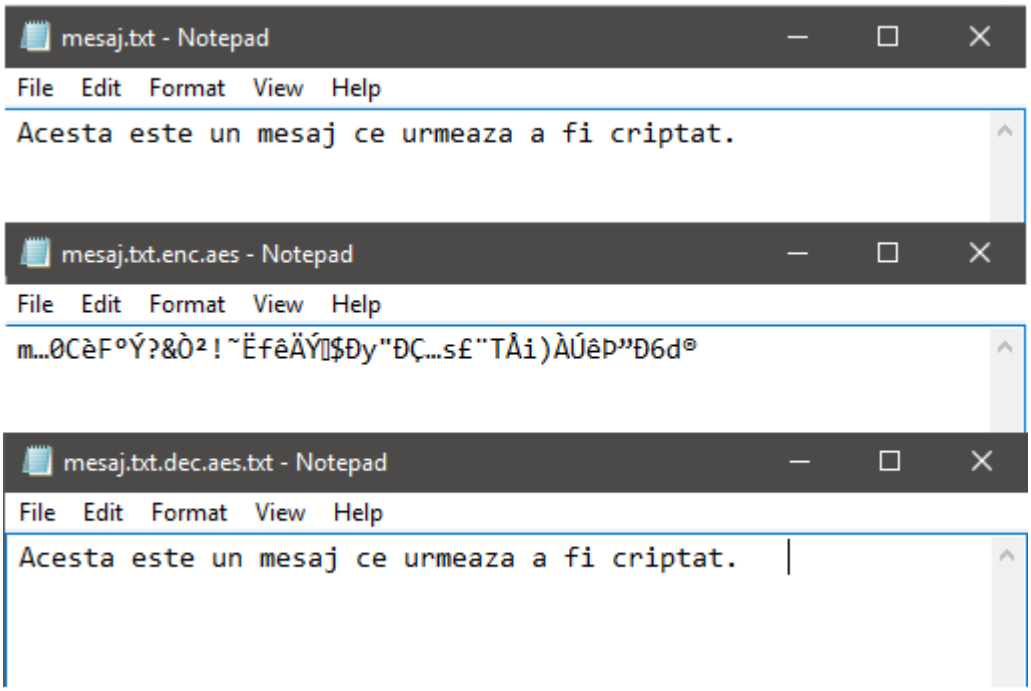

Fig. 4.1

După cum se observă, mesajul inițial nu are un număr de octeți divizibil cu numărul de octeți necesar pentru AES (128 biți / 8 = 16 octeți). Numărul de octeți din primul fișier este de 45, iar cum AES funcționează numai pe blocuri de câte 16 octeți, înseamnă că mesajul trebuie adus la un multiplu de 16 înainte de criptare, astfel ultimului bloc i se mai adaugă trei caractere nule, care de altfel se regăsesc și în varianta decriptată, marcatorul fiind cu trei spații poziționat după mesaj.

Un alt rezultat al aplicație constă în criptarea valorilor pixelilor unei imagini cu modul ECB, respectiv CBC pentru punerea în evidență a nivelului de securitate ale celor două moduri. În Figura 4.2 sunt prezentate, în ordine, de la stângă la dreapta: imaginea originală, imaginea criptată cu algoritmul DES cu modul de operare ECB, imaginea criptată cu algoritmul DES cu modul de operare CBC. După cum bine se observă, în acest caz, modul de operare CBC oferă un nivel de securitate mult mai bun decât modul ECB. Întrucât ECB criptează fiecare bloc independent, când e vorba de aceeași culoare pe o suprafață mare în imaginea originală, atunci tiparul răspândirii culorilor se va observa și în imaginea criptată cu modul ECB, iar cum modul CBC ține cont de blocul anterior pentru criptarea blocului curent, acesta oferă o imagine criptată cu aspect pseudo-aleator.

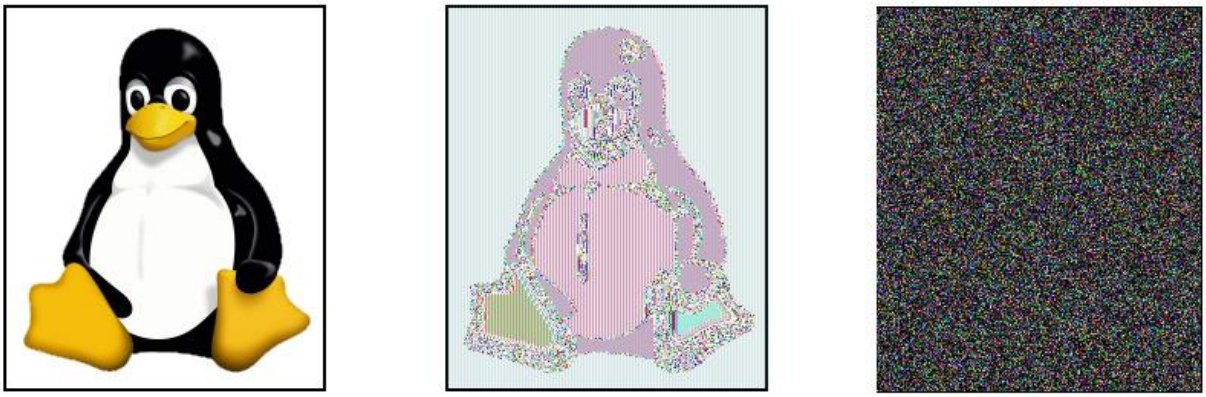

Fig. 4.2

Concluzii

Este evident că toți algoritmii criptografici practici au și o durată de viață, aceasta însemnând perioada de funcționare în care aceștia sunt siguri din punct de vedere criptografic, adică nu pot fi sparți cu tehnologia disponibilă curentă. De asemenea, creșterea volumului de informații fiind într-o continuă dezvoltare, conduce la implementarea algoritmilor în funcție de volumul datelor. Pentru proiectarea oricărui algoritm criptografic, tot mereu va trebui avută în vedere și perioada de funcționare a acestora, ca acesta să asigure protecția datelor fără a pune în pericol datele. Domeniul securității informatice va necesita întotdeauna implementarea algoritmilor ce oferă un grad înalt de securitate.

Capitolul 5

Bibliografie

[1] **C. Răcuciu –** Criptografie și securitatea informației, Editura Renaissance, București, 2010, ISBN 978-606-8321-89-9

[2] **C. Răcuciu, D. L. Grecu –** Metode și sisteme criptografice secvențiale, Editura ERICOM, București, 2008, ISBN 978-973-88290-9-1

[3] **A. J. Menezes, P. C. van Oorschot, S. A. Vanstone –** Handbook of Applied Cryptography, CRC Press, 1996, ISBN 0849385237

[4] **N. Ferguson, B. Schneier, T. Kohno –** Cryptography Engineering: Design Principles and Practical Applications, Ed. Wiley, 2010, ISBN 0470474246

[5] **Bruce Schneier –** Applied Cryptography, Second Edition: Protocols, Algorithms, and Source Code in C, Ed. Wiley, 1996, ISBN 0471128457

[6] **C. Paar, J. Pelzl –** Understanding Cryptography, Ed. Springer, 2010, ISBN 3642041000

[7] **NIST –** Advanced Encryption Standard (AES), FIPS-PUB 197, 2001, http://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.197.pdf 29.05.2017 ora 22:00.

[8] **NIST –** Recommendation for Block Cipher Modes of Operation, SP 800-38A, 2001, http://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-38a.pdf 29.05.2017 ora 23:00.