Лекция 5

Блокови шифри

5.1 Общи понятия

Дефиниция 5.1. Нека $V_n = \{0,1\}^n$. Една функция $E : V_n \times V_k \to V_n$ наричаме *блоков шифър*, ако $E_K(P) := E(P,K)$ е обратимо изображение от V_n във V_n за всяко фиксирано $K \in V_k$.

Множеството V_n идентифицираме с множеството на всевъзможните блокове, а V_k – с множество ключовете. Числата n и k наричаме съответно дължина на блока и дължина на ключа (в битове) за въпросния блоков шифър. Изображението $E_K : V_n \to V_n$ наричаме *шифриращо изображение за ключ* K. Обратното изображение $E_K^{-1} : V_n \to V_n$ наричаме *дешифрираща функция* и бележим с D_K. Ще пишем $C = E_K(P)$, ако криптотекстът C се получава при шифриране на открития текст P с ключа K. Очевидно са изпълнени равенствата:

$$E_K^{-1}(E_K(P)) = P, \quad \text{и} \quad E_K(E_K^{-1}(C)) = C$$

за всеки $P, C \in V_n$. За упростяване на разглежданията по нататък ще приемем, че всички ключове са равновероятни.

Не съществува точно очертана граница между класовете на блоковите и поточните шифри. Приема се, че при блоковите шифри n е сравнително голямо ($n = 64, 128, 192, \ldots$), докато поточните шифри оперират върху по-малки единици, най-често отделни битове.

Шифри, реализиращи всяка пермутация на V_n имат дължина на ключа $k \geq log(2^n!) \approx n2^n$, което е огромно дори за умерени стойности на n. Вместо това от добрите блокови шифри се очаква следното свойство: шифриращата фиункция, съответстваща на случайно избран ключ K изглежда като "случайно избрана" обратима функция. По-нататък ще изясним какво ще разбираме под "случайно избрана функция".

Блокови шифри с малка дължина на блока са уязвими по отношение на атаки, основаващи се на статистически анализ. От друга страна увеличаването на n води до нарастване на сложността на имплементацията.

Типичния начин за използване на един блоков шифър е следния. Партиите $A, B, \ldots$, които ще обменят данни, избират случаен ключ K, който е неизвестен за

опонента. Ключът определя шифрираща функция E_K, с помощта на която се трансформират съобщенията, обменяни между $A, B, \ldots$. Опонентът може да наблюдава известен брой входно-изходни двойки (P, C), където $C = E_K(P)$, но не и ключа или части от него. Така цялата сигурност на един блоков шифър е в използвания ключ K и зависи от това, доколко той е защитен. Добрата защита на K необходимо, но не и достатъчно условие за сигурността на един блоков шифър. Дизайнът на един блоков шифър трябва да осигури неосъществимост (infeasibility) на задачата за пресмятане на K при зададени двойки открит текст-криптотекст.

5.2 Принципи за изграждане на блокови шифри

Съществуват две общи идеологии за конструирането на блокови шифри – мрежи на Feistel и мрежи от субституции и пермутации, накратко SP-мрежи. [1]

5.2.1 Шифри от тип Feistel

Шифрите от тип Feistel са итерирани блокови шифри[2] , изобразяващи $2n$-битови блокове открит текст $2n$-битови блокове криптотекст чрез h-стъпков процес, който ще опишем по-долу. Всеки блок открит текст се разделя на две равни части $\boldsymbol{m} = (\boldsymbol{m}_0, \boldsymbol{m}_1)$. Ключът K определя множество от подключове: $K_1, \ldots, K_h$, където h е някакво цяло число. За всяко k_i е зададена трансформация f_{K_i}, изобразяваща блокове с дължина n в блокове със същата дължина. Всяко съобщение се шифрира в h рунда съгласно правилото:

$$\begin{aligned}
\mu_0 = (\boldsymbol{m_0}, \boldsymbol{m_1}) &\longrightarrow \mu_1 = (\boldsymbol{m_1}, \boldsymbol{m_2}) \\
\ldots \quad \ldots &\quad \ldots \\
\mu_{i-1} = (\boldsymbol{m_{i-1}}, \boldsymbol{m_i}) &\longrightarrow \mu_i = (\boldsymbol{m_i}, \boldsymbol{m_{i+1}}) \\
\ldots \quad \ldots &\quad \ldots \\
\mu_{h-1} = (\boldsymbol{m_{h-1}}, \boldsymbol{m_h}) &\longrightarrow \mu_h = (\boldsymbol{m_h}, \boldsymbol{m_{h+1}}),
\end{aligned}$$

където $\boldsymbol{m_i} = \boldsymbol{m_{i-1}} + f_{K_i}(\boldsymbol{m_i})$. Ще отбележим , че $\boldsymbol{m_{i-1}} = \boldsymbol{m_{i+1}} + f_{K_i}(\boldsymbol{m_i})$. Това гарантира, че шифриращата функция е обратима. Отук следва, че можем да дешифрираме като използваме същата процедура, взимайки ключовете в обратен ред и сменяйки местата на двете половини в получения резултат.

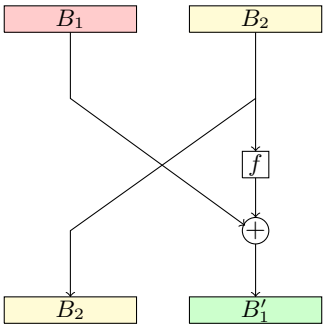
$$\begin{aligned}
\mu_h = (\boldsymbol{m_{h+1}}, \boldsymbol{m_h}) &\longrightarrow \mu_{h-1} = (\boldsymbol{m_h}, \boldsymbol{m_{h-1}}) \\
\ldots \quad \ldots &\quad \ldots \\
\mu_i = (\boldsymbol{m_{i+1}}, \boldsymbol{m_i}) &\longrightarrow \mu_{i-1} = (\boldsymbol{m_i}, \boldsymbol{m_{i-1}}) \\
\ldots \quad \ldots &\quad \ldots \\
\mu_1 = (\boldsymbol{m_2}, \boldsymbol{m_1}) &\longrightarrow \mu_0 = (\boldsymbol{m_1}, \boldsymbol{m_0}),
\end{aligned}$$

[1] Substitution-Permutation networks

[2] Под итериран блоков шифър обикновено се разбира блоков шифър, включващ последователното повтаряне на някаква вътрешна функция. Параметрите му включват броя на рундовете h, дължината на блока n, дължината k на ключа K, от който се получават подключовете K_i за отделните рундове. За всяко K_i вътрешната функция е биективна върху множеството на всевъзможните входове.

Важно свойство на тази идеология е, че тя не поставя никакви ограничения за функцията f. Дори тя да е необратима, схемите за шифриране и дешифриране ще работят правилно.

Така описаната схема наричме класическа схема на Feistel. Един рунд е представен на фигурата по-долу.



Освен класическия Feistel известни са и т.нар. *небалансирана схема на* Feistel, *алтернираща* Feistel, *схеми на* Feistel *от тип* I, II, и III. Един рунд за всяка от тези разновидности са представени на фигурите по-долу.

Известни шифри, които са мрежи на Feistel са:

- DES (класически Feistel);
- Skipjack (небалансиран Feistel);
- BEAR/LION (алтернираща Feistel);
- CAST-256 (мрежа на Feistel от тип I);
- RC6 (мрежа на Feistel от тип II);
- MARS (мрежа на Feistel от тип III).

5.2.2 Substitution-Permutation (SP) мрежи

За разлика от шифрите от тип Feistel върху всеки от байтовете (или битовете) на SP-мрежата се оперира по подобен начин чрез последователното прилагане на различни трансформации (*слоеве*). Примери за SP-мрежи са Rijndael, Serpent и SAFER.

Шифърът, който въвеждаме в този раздел е прост пример за SP-мрежа. Върху него ние илюстрираме две фундаментални атаки срещу блокови шифри в Глава **??**.

SPN приема за вход 16-битов входен блок и го обработва, повтаряйки базовите операции четири пъти. Всяко повторение (цикъл) се състои от:

(1) Субституция. Входният блок се разбива на четири 4-битови подблокове. Всеки подблок е вход на 4×4 S-box, който в този случай се имплементира като функция получаваща за вход четири и даваща на изхода четири бита. Линейният и диференциалният криптанализ работят еднакво добре, независимо дали S-box-овете в един цикъл са еднакви или различни. Тук за простота ще считаме, че четирите S-box-а са еднакви. Функцията, избрана за нашия S-box е дадена по-долу.

вход	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
изход	E	4	D	1	2	F	B	8	3	A	6	C	5	9	0	7

(2) Пермутация. Това е разместване на битовете на входната дума, зададено с пермутацията

$$\begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14 & 15 & 16 \\ 1 & 5 & 9 & 13 & 2 & 6 & 10 & 14 & 3 & 7 & 11 & 15 & 4 & 8 & 12 & 16 \end{pmatrix}.$$

да отбележим, че след последния цикъл не се извършва пермутация на елементите.

(3) Смесване с подключа. Това се извършва чрез побитов XOR между битовете на подключа, асоцциран с този цикъл и входния блок за същия подцикъл. Обикновено подключът за всеки цикъл се получава от избрания ключ с помощта на специален алгоритъм (key schedule). В SPN ще считаме че всички подключове са различни и независимо генерирани един от друг.

Дешифрирането при SPN е идентично с шифрирането. Разликата се състои в това, че изображенията в S-box-овете са обратни на изображенията, използвани при шифрирането. Освен това подключовете се подават в ред обратен на този при шифрирането.

5.3 DES

Шифърът DES е развит от IBM като модификация на по-ранен шифъ, известен като LUCIFER. Той е типичен пример за класическа мрежа на Feistel, приемаща за вход 64-битови блокове открит текст и извеждаща на изхода 64-битови блокове криптотекст. КЛючът е 64-битов и с еизползва както за шифриране, така и за дешифриране. Ефективният размер на ключа е 56 бита, тъй като всеки осми бит е проверка за четност. Широко споделяно е мнението, че проверочните битове са въведени с цел 256-кратното намаляване на изчерпващото търсене в множеството на всички ключове.

Шифрирането и дешифрирането с DES се извършва в 16 рунда. Специален алгоритъм, на който ще се спрем по-долу, генерира от зададения ключ шестнадесет 48-битови подключа K_i – по един за всеки рунд. Рундовете са функционално еквивалентни. Във всеки от тях 64-битовият вход се разбива на два 32-битови блока L_{i-1} и R_{i-1}, от които се получават два 32-битови блока L_i и R_i, $i = 1, 2, \ldots, 16$, по правилото

$$L_i = R_{i-1}; \tag{5.1}$$
$$R_i = L_{i-1} \oplus f(R_{i-1}, K_i). \tag{5.2}$$

Тук f е функция от $\{0,1\}^{32} \times \{0,1\}^{48}$ в $\{0,1\}^{32}$. Дешифрирането повтаря шифрирането като подключовете се подават в обратен ред.

Шифриращият алгоритъм на DES е представен схематично на Фигура 5.Ш. Битовете на един блок открит текст $m_1 m_2 \ldots m_{64}$ се подлагат на фиксирана начална пермутация IP

$$L_i = R_{i-1}; \tag{5.3}$$
$$R_i = L_{i-1} \oplus f(R_{i-1}, K_i). \tag{5.4}$$

(initial permutaition):

IP

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

Този запис означава, че в първа позиция отива 58-ми бит от входния блок, във втора позиция отива 50-ти бит, в трета – 42-ри бит и т.н. Пермутираният входен блок се разбива на два 32-битови блока, които се подлага на 16 итерации, зададени чрез (5.3) и (5.4). Функцията f е представена на Фигура 5.Ы. Входният блок се разширява от 32 до 48 бита чрез следната резширяваща трансформация E (bit selection table):

E-Bit Selection Table

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

$E(R_{i-1})$ се състои от битовете на R_{i-1}, взети в указания ред, като 16 от битовете се появяват по два пъти. По-нататък $E(R_{i-1}$ и пдоклыча K_i се събират побитово и резултатът се записва като конкатенация на 8 6-битови низа $B = E(R_{i-1}) \oplus K_i = B_1B_2 \dots B_8$. Следващата стъпка представлява субституция, използваща осем субституционни таблици (S-boxes) $S_1, S_2, \dots, S_8$. Всеки един от S-box-овете е таблица с 4 реда и 16 стълба и с елементи цели числа от 0 до 15. Низът B_1 се пребразува чрез използване на таблицата S_1, B_2 – чрез S_2 и т.н. Ако преобразуването, дефинирано чрез S_i, е означено също с S_i, и $B_i = b_1b_2b_3b_4b_5b_6$, то $S_i(B_i)$ се пресмята както следва:

- битовете b_1b_6 разгледани като число в двоична бройна система задават номер на ред от S_i, да речем u;

- по същия начин битовете $b_2b_3b_4b_5$ задават номер на стълб от S_i, да речем v;

- стойността на $S_i(B_i)$ е четирибитовия низ, който е двоичното представяне на елемента, намиращ се в позиция (u, v) на таблицата S_i.

По този начин пресмятаме $C_i = S_i(B_i)$, $i = 1, 2, \dots, 8$. Накрая 32-битовия низ $C = C_1C_2 \dots C_8$ се подлага на пермутацията P, зададена чрез

P

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

След изпълнение на 16-те рунда блокът $R_{16}L_{16}$ се подлага на пермутацията IP^{-1} – обратна на описаната по-горе пермутация IP:

IP^{-1}

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

Да отбележим, че редът на блоковете L_{16} и R_{16} е сменен преди прилагане на IP^{-1}. Целта е унификация на шифриращия и дешифриращия алгоритми.

Дешифрирането се извършва като приложим шифриращия алгоритъм към шифрирания блок. Единствената разлика е, че подучовете за поредните итерации трябва да се подават в обратен ред. След прилагане на IP получаваме $R_{16}L_{16}$ От равенствата (5.3) и (5.4), приложени за вход със сменени леви и десни подблокове, получаваме

$$\begin{aligned} R_{i-1} &= L_i; \\ L_{i-1} &= R_i \oplus f(L_i, K_i). \end{aligned}$$

Непосредствено се проверява, че на изхода се получава блока L_0R_0.

Остава да опишем алгоритъма за генериране подключовете от ключа K. Както отбелязахме, дължината на K е 64 бита, от които 56 бита се задават, а битове в позиции 8, 16, ... , 64 са дефинирани така, че всеки байт да съдържа нечетен брой единици. Следователно структурата на ключа позволява откриването на една грешка в коя да е от групите. Проверочните битове се игнорират при генриране на подключовете.

Алогиръмът за генериране на подключовете е представен схематично на Фигура 5.3. При зададен ключ K проверочните битове се игнорират, а останалите се разместват в съответствие с фиксираната пермутация PC-1. Записваме резултата след прилагането на PC-1 във вида C_0D_0, където C_0 са първите 28 бита, а D_0 – последните 28 бита на резултата. за всяко $i = 1, 2, \ldots, 16$ пресмятаме

$$\begin{aligned} C_i &= LS_i(C_{i-1}), \\ D_i &= LS_i(D_{i-1}), \end{aligned}$$

където LS_i е цикличен шифт наляво на една или две позиции взависимост от стойността на i: шифтът е на една позиция за $i = 1, 2, 9$ или 16 и на две позиции – в противен случай. По-нататък трансформираме C_iD_i чрез трансформацията PC-2. Резултатът е подключа $K_i =$.

PC-1

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

PC-2

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

Ще отбележим, че след генериране на K_{16} оригиналните стойности на регистрите C и D се възстановяват, тъй като всеки от тях е бил ротиран на 28 бита. Сега подключове $K_{16}, K_{15}, \ldots, K_1$ могат да бъдат генерирани (в указания ред) по ако заменим в гореописания алгоритъм левите шифтове с десни и сменим стойността на първия шифт на $v_1 = 0$.

По-долу ще коментираме някои свойства на DES

(1) *Комплементарност на DES.* Ако DES, използван с ключ K, задава преобразуването E_K, то за всеки 64-битов блок x от $y = E_K(x)$ следва $\overline{y} = E_{\overline{K}}(\overline{x})$. Това свойство намалява сложността на атака по известен открит текст, използваща пълно изчерпване на всички ключове. Нека опонентът разполага с двойките открит текст-криптотекст (P_1, C_1), $(\overline{P_1}, C_2)$ получена при шифриране с неизвестен за него ключ K. От $C_2 = E_K(\overline{P_1})$ получаваме $\overline{C_2} = E_{\overline{K}}(P_1)$. Достатъчно е да проверим дали шифрирането на P_1 с ключ K дава C_1 или $\overline{C_2}$; така с едно шифрираме имаме възможност да отхвърлим два ключа и да намалим броя на шифрираниятa достатъчни за упешен криптанализ от 2^{56} на 2^{55}. [3]

(2) *Слаби и полуслаби ключове.* Ако подключовете K_1 до K_{16} са равни, то оригиналната редица на подключовете и обърнатат и́ версия съвпадат; $K_1 = K_{16}$, $K_2 = K_{15}$, и т. н. Следователно шифрирането и дешифрирането ще съвпадат. Такива ключове се наричат *слаби ключове* или *палиндромни ключпве.*

(3) *DES не е група.* За всеки фиксиран ключ K DES дефинира пермутация на $\{0,1\}^{64}$. Множеството от от всички ключове на DES дефинира 2^{56} потенциално различни пермутации. Ако това множество е затворено относно композиция, т.е. про два произволни ключа K_1 и K_2, съществува трети ключ K_3, за който $E_{K_3}(x) = E_{K_2}(E_{K_1}(x))$ за всички x., то многократното шифриране е квивалентно на единствено шифриране. Беше доказано, че това не е изпълнено за DES.

Множеството от всички 2^{56} пермутации, дефинирани от всички ключове на DES не е затворено относно композиция. Нещо повече, групата породена от всевъзможните композиции е от ред поне 10^{2499}. Този факт е важен при прилагане на DES при многократно шифриране. Ако тази група беше от малък ред, многократно шифриране би било по-несигурно, отколкото се очаква.

[3] Разбира се, очакваният брой ключове е половината на това число – 2^{54}.

5.4 Rijndael

На 2. октомври 2000 г. NIST обяви, че за нов стандарт за блоков шифър или Advanced Encryption Standard (AES) е избран шифъра Rijndael. Името Rijndael е абревиатура от имената на авторите – Joan Daemen от (Proton World International) и Vincent Rijmen от (Katholieke Universiteit Leuven). Rijndael е относително прост шифър с богата математическа структура. Той поддържа дължини на блока и ключа от 128, 192 и 256 бита, като тези дължини могат да се използват в произволна комбинация. Rijndael е изграден като SP-мрежа. Шифрирането се извършва чрез многократно повторение на три слоя от трансформации:

- *нелинеен разбъркващ слой* (`ByteSub`): паралелно се прилага трансформация (S-Box) с оптимални нелинейни характеристики;
- *линеен разбъркващ слой* (`ShiftRow`, `MixColumn`): разбъркване на части на шифрирания блок, гарантиращо висока степен на дифузия след известен брой рундове;
- *добавяне на подключ* (`AddRoundKey`): изпълнение на прост XOR на подключ с междинното състояние.

С цел унифициране на алгоритмите за шифриране и дешифриране линейният разбъркващ слой в последния рунд е различен от този в другите рундове. Това не влияе на сигурността на шифъра. Сходна черта намираме и при DES, където в последния рунд липсва размяна на двете половини на преобразувания блок.

5.4.1 Основи на **Rijndael**

Част от операциите в Rijndael са дефинирани на байтово ниво, като байтовете се интерпретират като елементи на полето $GF(2^8)$. Останалата част от операциите са върху 4-байтови думи. По-долу въведем основните математически обекти, необходими за описание на шифъра.

Елементите на крайно поле могат да бъдат представени по няколко начина. Тук ние се спираме на класическото представяне през полиноми. Един байт $\boldsymbol{b}$, състоящ се от битовете $b_7, b_6, b_5, b_4, b_3, b_2, b_1, b_0$ разглеждаме като полином с коефициенти от $GF(2)$:

$$b_7 \cdot x^7 + b_6 \cdot x^6 + b_5 \cdot x^5 + b_4 \cdot x^4 + b_3 \cdot x^3 + b_2 \cdot x^2 + b_1 \cdot x + b_0$$

Така байтът с шестнадесетично представяне `'79'` (двоично `'01111001'`) съответства на полинома $x^6 + x^5 + x^4 + x^3 + 1$.

Сумирането и умножението на байтове се дефинират чрез сумирането и умножението на елементи в $GF(2^8)$. Например, `'79' + '35' = '46'`, което в полиномно представяне се записва като

$$(x^6 + x^5 + x^4 + x^3 + 1) + (x^5 + x^4 + x^2 + 1) = x^6 + x^3 + x^2$$

а в двоично представяне като `'01111001' + '00110101' = '01001100'`. Очевидно събирането представлява прост побитов XOR, който на ниво байт ще означаваме с $\oplus$.

Умножението на елементи от произволно крайно поле се извършва като умножение на полиноми по модул неразложим полином. Този неразложим полином в Rijndael се означава с $m(x)$ и е фиксиран като:

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

или `'11B'` в шестнадесетично представяне. Това е първият полином в списъка с неразложими полиноми от книгата на Lidl и Niedereiter [?]. Сега `'57' • '83' = 'C1'`, тъй като

$$\begin{aligned}(x^6 + x^4 + x^2 + x + 1)(x^7 + x + 1) &= x^{13} + x^{11} + x^9 + x^8 + x^7 \\ &\quad + x^7 + x^5 + x^3 + x^2 + x \\ &\quad + x^6 + x^4 + x^2 + x + 1 \\ &= x^{13} + x^{11} + x^9 + x^8 + x^6 + x^5 + x^4 + x^3 + 1\end{aligned}$$

$$x^{13} + x^{11} + x^9 + x^8 + x^6 + x^5 + x^4 + x^3 + 1 \pmod{x^8 + x^4 + x^3 + x + 1} = x^7 + x^6 + 1$$

За разлика от събирането, за умножението няма проста операция на байтово ниво.

Някои операции в Rijndael са дефинирани върху 4-байтови думи, които могат да бъдат разглеждани като полиноми над $GF(2^8)$. Събирането на думи се свежда до събиране на полиноми, т.е. до събиране на коефициентите пред членовете с равни степени. Тъй като събирането в $GF(2^8)$ е побитов XOR, то и събирането на два полинома от степен по ниска от 4 също е побитов XOR на думи.

Умножението на думи се дефинира чрез умножението на съответните им полиноми във факторпръстена $\mathbb{F}_{256}[x]/(M(x))$, където $M(x) = x^4 + 1$. Нека са дадени думите $\boldsymbol{a}$ и $\boldsymbol{b}$, с които се свързват полиномите:

$$a(x) = a_3x^3 + a_2x^2 + a_1x + a_0, \;\; b(x) = b_3x^3 + b_2x^2 + b_1x + b_0 \;\; a_i, b_j \in GF(2^8).$$

За произведението им $c(x) = a(x)b(x)$ имаме:

$$c(x) = c_6x^6 + c_5x^5 + c_4x^4 + c_3x^3 + c_2x^2 + c_1x + c_0,$$

където

$$\begin{aligned} c_0 &= a_0 \bullet b_0, & c_4 &= a_3 \bullet b_1 \oplus a_2 \bullet b_2 \oplus a_1 \bullet b_3, \\ c_1 &= a_1 \bullet b_0 \oplus a_0 \bullet b_1, & c_5 &= a_3 \bullet b_2 \oplus a_2 \bullet b_3, \\ c_2 &= a_2 \bullet b_0 \oplus a_1 \bullet b_1 \oplus a_0 \bullet b_2, & c_6 &= a_3 \bullet b_3, \\ c_3 &= a_3 \bullet b_0 \oplus a_2 \bullet b_1 \oplus a_1 \bullet b_2 \oplus a_0 \bullet b_3. \end{aligned}$$

За да представим $c(x)$ като 4-байтов вектор, ние го редуцираме $c(x)$ по модул полином от степен 4. При Rijndael това става по модул полинома $M(x) = x^4 + 1$. Тъй като

$$x^i \pmod{x^4 + 1} = x^{i \pmod 4},$$

произведението на $a(x)$ и $b(x)$ във $\mathbb{F}_{256}[x]/(M(x))$, което ще означаваме с $d(x) = a(x) \otimes b(x)$, се задава чрез:

$$d(x) = d_3x^3 + d_2x^2 + d_1x + d_0,$$

където

$$\begin{aligned}
d_0 &= a_0 \bullet b_0 \oplus a_3 \bullet b_1 \oplus a_2 \bullet b_2 \oplus a_1 \bullet b_3 \\
d_1 &= a_1 \bullet b_0 \oplus a_0 \bullet b_1 \oplus a_3 \bullet b_2 \oplus a_2 \bullet b_3 \\
d_0 &= a_2 \bullet b_0 \oplus a_1 \bullet b_1 \oplus a_0 \bullet b_2 \oplus a_3 \bullet b_3 \\
d_0 &= a_3 \bullet b_0 \oplus a_2 \bullet b_1 \oplus a_1 \bullet b_2 \oplus a_0 \bullet b_3
\end{aligned}$$

Ако полиномът $a(x)$ е фиксиран, тази операция може да се запише като умножение на циркулантна матрица с вектор:

$$\begin{pmatrix} d_0 \\ d_1 \\ d_2 \\ d_3 \end{pmatrix} = \begin{pmatrix} a_0 & a_3 & a_2 & a_1 \\ a_1 & a_0 & a_3 & a_2 \\ a_2 & a_1 & a_0 & a_3 \\ a_3 & a_2 & a_1 & a_0 \end{pmatrix} \begin{pmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \end{pmatrix}.$$

Тъй като $x^4 + 1 = (x + 1)^4$ е разложим над $GF(2^8)$ не всеки полином е обратим в $\mathbb{F}_{256}[x]/(M(x))$. Ще отбележим, че в трансформацията `MixColumnn` Rijndael използва фиксиран полином $c(x)$, за който съществува обратен в $\mathbb{F}_{256}[x]/(M(x))$(виж 5.4.2.3).

5.4.2 Спецификациии на **Rijndael**

Различните трансформации се прилагат върху блок с фиксирана дължина, наричан *вектор на състояние*.

Можем да разглеждаме междинното състояние като масив от байтове. Този масив е разположен в 4 реда. Броят на колоните се означава с `Nb` и е равен на дължината на блока разделена на 32.

По подобен начин и ключът се разглежда като правоъгълен масив с 4 реда. Броят на колоните се означава с `Nk` и е равен на дължината на ключа върху 32. Това представяне е илюстрирано на Фигура 2.1.

$a_{0,0}$	$a_{0,1}$	$a_{0,2}$	$a_{0,3}$	$a_{0,4}$	$a_{0,5}$
$a_{1,0}$	$a_{1,1}$	$a_{1,2}$	$a_{1,3}$	$a_{1,4}$	$a_{1,5}$
$a_{2,0}$	$a_{2,1}$	$a_{2,2}$	$a_{2,3}$	$a_{2,4}$	$a_{2,5}$
$a_{3,0}$	$a_{3,1}$	$a_{3,2}$	$a_{3,3}$	$a_{3,4}$	$a_{3,5}$

$k_{0,0}$	$k_{0,1}$	$k_{0,2}$	$k_{0,3}$
$k_{1,0}$	$k_{1,1}$	$k_{1,2}$	$k_{1,3}$
$k_{2,0}$	$k_{2,1}$	$k_{2,2}$	$k_{2,3}$
$k_{3,0}$	$k_{3,1}$	$k_{3,2}$	$k_{3,3}$

Фигура 2.1: Междинното състояние и ключът.

В някои случаи блоковете на междинното състояние и ключа се разглеждат като едномерни масиви от 4-байтови вектори. Всеки вектор представлява стълб в съответния масив. Следователно, тези масиви имат дължини 4, 6 или 8.

Входът и изходът, използвани от Rijndael във външния му интерфейс се разглеждт като едномерни масиви от 8-битови байтове, номерирани от 0 до 4`Nb` - 1. Тези блокове имат дължини съответно 16, 24 или 32 байта с индекси 0..15, 0..23 или

0..31. Ключът се разглежда като едномерен масив от байтове, номерирани от 0 до 4Nk - 1. Тези блокове също имат дължини 16, 24 или 32 байта с индекси 0..15, 0..23 или 0..31.

Входните байтове на шифъра (откритият текст) се изобразяват в байтовете на междинното състояние в реда $a_{0,0}$, $a_{1,0}$, $a_{2,0}$, $a_{3,0}$, $a_{0,1}$, $a_{1,1}$, $a_{2,1}$, $a_{3,1}$, $a_{0,2}$ ..., а байтовете на ключа - в масив с последователността $k_{0,0}$, $k_{1,0}$, $k_{2,0}$, $k_{3,0}$, $k_{0,1}$, $k_{1,1}$, $k_{2,1}$, $k_{3,1}$, $k_{0,2}$ ... В края на операцията, изходът се извлича от междинното състояние в същия ред. Ако едномерният индекс на даден байт е n, а двумерният - (i, j), то

$$i = n \pmod 4; \qquad j = \lfloor n/4 \rfloor; \qquad n = i + 4j$$

Освен това, индексът i е и номерът на байта в 4-байтовия вектор (дума), а j - индекса на вектора (думата) в дадения блок. Броят на рундовете се означава с Nr и зависи от Nb и Nk. Точният им брой е даден в Таблица 1.

Nr	Nb = 4	Nb = 6	Nb = 8
Nk = 4	10	12	14
Nk = 6	12	12	14
Nk = 8	14	14	14

Таблица 1: Брой на рундовете в зависимост от Nb и Nk.

Всеки рунд се състои от четири различни трансформации. Представени като псевдокод те са:

```
Round (State, RoundKey)
{
   ByteSub(State);
   ShiftRow(State);
   MixColumn(State);
   AddRoundKey(State, RoundKey);
}
```

Последната стъпка е различна с това, че при нея се пропуска MixColumn. Тя се представя чрез:

```
FinalRound (State, RoundKey)
{
   ByteSub(State);
   ShiftRow(State);
   AddRoundKey(State, RoundKey);
}
```

Функциите (Round, ByteSub, ShiftRow, ...) се прилагат върху масиви, за които са предоставени указатели (State, RoundKey).

По-долу ще опишем отделните трансформации.

5.4.2.1 Трансформацията ByteSub

Трансформацията ByteSub е нелинейна байтова субституция, която се прилага върху всички байтове на междинното сътояние поотделно. Субституционната таблица (S-box) е обратима и се състои от две трансформации:

1. Всеки байт се замества с обратния на него в $GF(2^8)$, както е описано в секция **??**. Байтът '00' се замества със себе си.

2. Върху всеки байт се прилага следната афинна (върху $GF(2)$) трансформация:

$$\begin{pmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \\ y_6 \\ y_7 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 \end{pmatrix} \begin{pmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \\ x_7 \end{pmatrix} + \begin{pmatrix} 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \\ 0 \end{pmatrix}$$

Прилагането на описания S-box върху всички байтове се означава с:

`ByteSub(State)`.

Обратната трансформация на ByteSub се получава като приложим обратната матрица и след това намерим обратния елемент в $GF(2^8)$.

5.4.2.2 Трансформацията ShiftRow

При трансформацията ShiftRow всеки от редовете на междинното състояние се премества циклично наляво на различен брой байтове. Ред 0 не се измества. Ред 1 се измества на C1 байта, Ред 2 – на C2, а Ред 3 – на C3 байта. C1, C2 и C3 зависят от дължината на блока Nb. Различните стойности са описани в Таблица 2.

Nb	C1	C2	C3
4	1	2	3
6	1	2	3
8	1	3	4

Таблица 2: Отмествания на редовете при трансформацията ShiftRow.

Операцията на преместването на всички байтове в междинното състояние се означава с:

`ShiftRow(State)`.

Обратната трансформация на ShiftRow е преместването на последните три реда съответно на Nb-C1, Nb-C2 и Nb-C3 байта, така че байтът на позиция j в ред i се премества на позиция $j + \mathtt{Nb} - \mathtt{C}i \pmod{\mathtt{Nb}}$.

5.4.2.3 Трансформацията `MixColumn`

При трансформацията `MixColumn` стълбовете на междинното състояние се разглеждат като полиноми над $GF(2^8)$ и се умножават по модул x^4+1 с фиксиран полином $c(x)$, зададен чрез:

$$c(x) = \texttt{'03'}x^3 + \texttt{'01'}x^2 + \texttt{'01'}x + \texttt{'02'}.$$

Този полином е взаимнопрост с x^4+1 и следователно е обратим. Както е описано в раздел **??**, това умножение може да се пердстави като умножение на матрица с вектор. Нека $b(x) = c(x) \otimes a(x)$. Тогава:

$$\begin{pmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \end{pmatrix} = \begin{pmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{pmatrix} \begin{pmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \end{pmatrix}.$$

Прилагането на тази операция върху всички колони на междинното състояние се означава с:

`MixColumn(State)`.

Обратната трансформация на `MixColumn` е подобна. Всыеки стълб се умножава с фиксиран полином $d(x)$, дефиниран чрез:

$$(\texttt{'03'}x^3 + \texttt{'01'}x^2 + \texttt{'01'}x + \texttt{'02'}) \otimes d(x) = 1.$$

Полиномът $d(x)$ е равен на:

$$d(x) = \texttt{'0B'}x^3 + \texttt{'0D'}x^2 + \texttt{'09'}x + \texttt{'0E'}.$$

5.4.2.4 Добавяне на подключ

При тази трансформация към междинното състояние се добавя подключ с прост побитов XOR. Алгоритъмът за получаване на подключове от зададен ключ е описан в раздел 5.4.2.5. Дължината на подключа е равна на дължината на блока `Nb`.

Трансформацията е илюстрирана на Фигура 2.2 и се означава с:

`AddRoundKey(State, RoundKey)`

`AddRoundKey` и обратната ѝ трансформация съвпадат.

5.4.2.5 Генериране на подключове

Подключовете за отделните стъпки се получават от зададения ключ чрез специална трансформация, която се състои от две компоненти – разширяване на ключа и избиране на подключове. Основният принцип е следният:

- Общият брой на битовете в даден подключ е равен на дължината на блока по броя на стъпките плюс 1 (напр. за дължина на блока 192 бита и 12 стъпки са нужни 2496 бита за подключа).

$$\begin{array}{|c|c|c|c|c|c|}\hline a_{0,0} & a_{0,1} & a_{0,2} & a_{0,3} & a_{0,4} & a_{0,5} \\ \hline a_{1,0} & a_{1,1} & a_{1,2} & a_{1,3} & a_{1,4} & a_{1,5} \\ \hline a_{2,0} & a_{2,1} & a_{2,2} & a_{2,3} & a_{2,4} & a_{2,5} \\ \hline a_{3,0} & a_{3,1} & a_{3,2} & a_{3,3} & a_{3,4} & a_{3,5} \\ \hline \end{array} \oplus \begin{array}{|c|c|c|c|c|c|}\hline k_{0,0} & k_{0,1} & k_{0,2} & k_{0,3} & k_{0,4} & k_{0,5} \\ \hline k_{1,0} & k_{1,1} & k_{1,2} & k_{1,3} & k_{1,4} & k_{1,5} \\ \hline k_{2,0} & k_{2,1} & k_{2,2} & k_{2,3} & k_{2,4} & k_{2,5} \\ \hline k_{3,0} & k_{3,1} & k_{3,2} & k_{3,3} & k_{3,4} & k_{3,5} \\ \hline \end{array} =$$

$$= \begin{array}{|c|c|c|c|c|c|}\hline b_{0,0} & b_{0,1} & b_{0,2} & b_{0,3} & b_{0,4} & b_{0,5} \\ \hline b_{1,0} & b_{1,1} & b_{1,2} & b_{1,3} & b_{1,4} & b_{1,5} \\ \hline b_{2,0} & b_{2,1} & b_{2,2} & b_{2,3} & b_{2,4} & b_{2,5} \\ \hline b_{3,0} & b_{3,1} & b_{3,2} & b_{3,3} & b_{3,4} & b_{3,5} \\ \hline \end{array}$$

Фигура 2.2: Трансформацията `AddRoundKey`

- Ключът се разширява до т.нар. *разширен ключ.*
- Подключовете се избират от разширения ключ по следния начин: първите `Nb` бита формират първия подключ, вторите `Nb` бита – втория подключ и т.н.

Разширяване на ключа. Разширеният ключ представлява масив от 4-байтови думи (вектори) и се означава с `W[Nb * (Nr + 1)]`. Първите `Nk` на брой думи съдържат оригиналния ключ. Всички други се дефинират рекурсивно чрез думи с по-малки индекси. Функцията за разширение на ключа зависи от `Nk`: за ная съществуват различни версии при $\texttt{Nk} \leq 6$ и при $\texttt{Nk} > 6$.

За $\texttt{Nk} \leq 6$ имаме:

```
KeyExpansion (byte Key[4*Nk], word W[Nb*(Nr+1)])
{
     for (i = 0; i < Nk; i++)
           W[i] = (Key[4*i], Key[4*i+1], Key[4*i+2], Key(4*i+3]);

     for (i = Nk; i < Nb * (Nr + 1); i++)
     {
           temp = W[i-1];
   if (i % Nk == 0)
        temp = SubByte(RotByte(temp)) ^ Rcon[i / Nk];
   W[i] = W[i - Nk] ^ temp;
     }
}
```

`SubByte(W)` връща 4-байтова дума (вектор), в която всеки байт е резултат от прилагането на описания в раздел 5.4.2.1 S-box върху съответната позиция на входната дума. Функцията `RotByte(W)` връща дума, в която байтовете са циклично преместени с една позиция от байтовете на входа т.е. ако входът е (a, b, c, d), то на изхода ще имаме (b, c, d, a).

Първите `Nk` думи са запълнени със сесийния ключ. Всяка следваща дума `W[i]` се получава като XOR на предишната дума `W[i-1]` и думата с `Nk` позиции по-назад `W[i-Nk]`. За думи с индекси кратни на `Nk` се прилага и допълнителна трансформация

преди XOR. Освен това с XOR се добавя и стъпковата константа. Трансформацията се състои в циклично преместване на байтовете в думата (т.е. `RotByte`), последвано от прилагането на таблицата върху всички байтове в думата (т.е. `SubByte`).

За `Nk` > 6 имаме:

```
KeyExpansion (byte Key[4*Nk], word W[Nb*(Nr+1)])
{
     for (i = 0; i < Nk; i++)
           W[i] = (Key[4*i], Key[4*i+1], Key[4*i+2], Key(4*i+3]);

     for (i = Nk; i < Nb * (Nr + 1); i++)
     {
           temp = W[i-1];
   if (i % Nk == 0)
        temp = SubByte(RotByte(temp)) ^ Rcon[i / Nk];
   else if (i % Nk == 4)
temp = SubByte(temp);
   W[i] = W[i - Nk] ^ temp;
     }
}
```

Разликата с функцията за `Nk` ≤ 6 е, че ако $i-4$ е кратно на `Nk`, `SubByte` се прилага върху `W[i-1]` преди XOR.

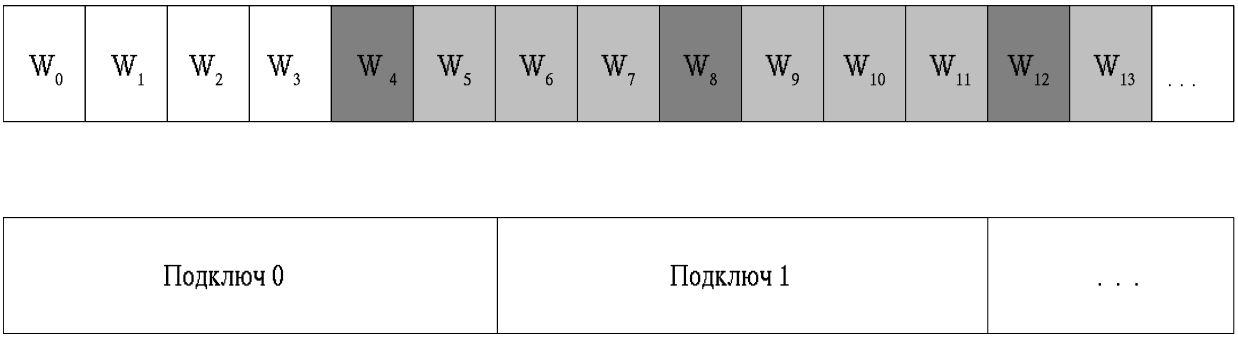
Стъпковите константи са независими от `Nk` и се дефинират чрез:

$$\texttt{Rcon[i]} = (\texttt{RC[i]}, \texttt{'00'}, \texttt{'00'}, \texttt{'00'}),$$

където `RC[i]` представлява елемент от $GF(2^8)$ със стойност x^{i-1}, така че:

$$\begin{aligned} \texttt{RC[1]} &= 1(\text{т.е.}\texttt{'01'}) \\ \texttt{RC[i]} &= x(\text{т.е.}\texttt{'02'}) \bullet (\texttt{RC[i} - 1\texttt{]}) = x^{i-1} \end{aligned}$$

Избиране на подключове. Подключ `i` се задава от думите на разширения ключ `W[Nb*i]` до `W[Nb*(i+1)]`. Това е показано на Фигура 2.3.



Фигура 2.3: Генериране на подключовете.

5.4.3 Шифърът **Rijndael**

Шифърът Rijndael се състои от:

- Първоначално добавяне на подключ;
- `Nr` $-$ 1 рунда;
- последен рунд.

В псевдокод имаме:

```
Rijndael(State, CipherKey)
{
   KeyExpansion(CipherKey, ExpandedKey);
   AddRoundKey(State, ExpandedKey);
   for (i = 1; i < Nr i++) Round(State, ExpandedKey + Nb*i);
   FinalRound(State, ExpandedKey + Nb*Nr);
}
```

Разшряването на ключа може да стане и преди това и функцията Rijndael да се дефинира чрез разширения ключ:

```
Rijndael(State, ExpandedKey)
{
   AddRoundKey(State, ExpandedKey);
   for (i = 1; i < Nr i++) Round(State, ExpandedKey + Nb*i);
   FinalRound(State, ExpandedKey + Nb*Nr);
}
```

5.5 Режими на работа на блокови шифри

Всеки блоков шифър преобразува открити текстове с фиксиран размер от n бита. Съобщения, чиято дължина надхвърля n, се разбиват на n-битови блокове, всеки от които се шифрира отделно. Съществуват различни режими на използване на блокови шифри върху съобщения по-дълги от n бита. По-долу ще опишем четирите най-често изпозвани режима: ECB, CBC, CFB и OFB.

По-долу E_K означава шифрираща функция, а E_K^{-1} – нейната обратна. Ще предполагаме, че в съобщението $x = x_1x_2\ldots x_s$ блоковете x_i са n-битови за режимите ECB и CBC и r-битови, $r \leq n$, за режимите CFB и OFB.

5.5.1 Режим **ECB**

В режима ECB (electronic codebook) всеки блок се шифрира независимо от останалите и се описва чрез следния алгоритъм:

$E_K(x_1x_2\ldots x_s)$	$E_K^{-1}(c_1c_2\ldots c_s)$
`for` $i = 1$ `to` s `do`	`for` $i = 1$ `to` s `do`
$\quad c_i \longleftarrow E_K(x_i)$	$\quad x_i \longleftarrow E_K(c_i)$
`return` $c_1c_2\ldots c_s$	`return` $x_1x_2\ldots x_s$

Лесно се забелязва, че при този режим идентични блокове отиват в идентични. Шифрирането на блоковете е независимо и пренареждането им води до пренареждане на съответните блокове криптотекст. Поради това режимът ECB не се препоръчва за съобщения по-дълги от един блок или за съобщения, при които един ключ се използва за шифрирането на повече блокове. Това може да бъде поправено донякъде чрез запълване на всеки блок със случайни битове.

От друга страна грешки при предаването на даден блок водят до грешки при дешифрирането единствено на този блок, т.е. нямаме разпространяване на грешките (error propagation). Тъй като различните шифрирани блокове са независими, злонамереното заместване на някои от тях не засяга дешифрирането на съседните блокове.

5.5.2 Режим **CBC**

Режимът CBC, описан по-долу, използва n-битов инициализиращ вектор v.

$E_K(vx_1x_2\dots x_s)$	$E_K^{-1}(c_1c_2\dots c_s)$
$c_0 \longleftarrow v$	`for` $i = 1$ `to` s `do`
`for` $i = 1$ `to` s `do`	$\quad x_i \longleftarrow E_K^{-1}(c_i \oplus c_{i-1})$
$\quad c_i \longleftarrow E_K(c_{i-1} \oplus x_i)$	`return` $x_1x_2\dots x_s$
`return` $c_1c_2\dots c_s$	

При този режим идентични блокове криптотекст се получават, когато един и същ открит текст се шифрира с един и същ ключ и един и същ инициализиращ вектор v. Промяна в получения блок криптотекст може да настъпи само ако променим поне един от низовете v, K или x_1. Блокът c_j зависи не само от x_j, но и всички предшестващи го блокове открит текст Пренареждането на блоковете криптотекст води до неправилно дешифриране.

Да приемем, че при предаването на криптотекста е сгрешен един бит в блока c_j. Това засяга дешифрирането на блоковете c_j и c_{j+1}. Блокът x'_j, получен от c_j е случаен, докато дешифрираният блок x'_{j+1} има грешки точно в сгрешените позиции на c_j. Така опонентът би могъл да предизвиква предвидими промени на битове в x_{j+1} изменяйки c_j. Режимът CBC е самосинхронизиращ се в смисъл, че при поява на грашка в блок c_j, но не и в c_{j+1} и c_{j+2} дешифрирането до x_{j+2} е правилно. Да отбележим обаче, че възстановяването на изтривания в шифрираните данни. Макар режимът CBC да се възстановява след грешки в шифрираните блокове, модификация в един блок открит текст води до промяна във всички следващи шифрирани блокове. Това ограничава приложимостта му при възможност за четене и писане в шифрираните данни.

Инициализиращият вектор v може да не бъде пазен в тайна, но целостта му трябва да бъде защитена. Модификацията му позволява на опонента да предизвика предсказуеми промени някои битове в първия блок открит текст. Запазването на секретността на v е един начин за защита от такава атака.

5.5.3 Режим **CFB**

В режим CBC n-битов шифър обработва на всяка стъпка n-битови блокове открит текст. В някои приложения може да се изисква обработването на r-битови блокове

за някакво фиксирано $r < n$. В такъв случай се използва режим CFB (cipher feedback mode). Използването му е описано по-долу.

ВХОД: k битов ключ K, n-битов инициализиращ вектор $\boldsymbol{v}$; r-битов блок открит текст.

ШИФРИРАНЕ:

(a) $I_1 \longleftarrow \boldsymbol{v}$ (I_j е начална стойност на шифт-регистър, $1 \leq j \leq u$.)

(b) `for j=1 to u`

$O_j \longleftarrow E_K(I_j)$

$t_j \longleftarrow$ най-левите r бита на на O_j

$c_j = x_j + t_j$ (r-битовият блок c_j се изпраща)

$I_{j+1} \longleftarrow 2^r \cdot I_j + c_j \pmod{2^n}$ (Блокът c_j се измества в десния край на шифт-регистъра.)

ДЕШИФРИРАНЕ:

(a) $I_1 \longleftarrow \boldsymbol{v}$

(b) `for j=1 to u` (след получаване на c_j)

$x_j \longleftarrow c_j \oplus t_j$

Както и при CBC използването на различни инициализиращи вектори върху фиксиран открит текст води до различни криптотекстове. Векторът $\boldsymbol{v}$ може да не е таен, но при някои приложени може да се изисква непредсказуем $\boldsymbol{v}$. Пренареждане на блоковете криптотекст води до грешно дешифриране. За правилно дешифриране е необходимо предхождащите $\lceil n/r \rceil$ блока криптотекст да са правилно получени. Един сгрешен бит в блока c_j влияе не дешифрирането на следващите $\lceil n/r \rceil$ блока (т.е. до обработването на n бита криптотекст, след което c_j не влияе на шифт-регистъра). Възстановеният открит текст x_j се различава от x_j точно в сгрешените битове на c_j; останалите неправилно дешифрирани блокове са случайни (очакват се 50% от битовете да са сгрешени. Така опонентът може да предизвиква предсказуеми грешки в x_j, променяйки съответните битове в c_j.

Подобно на CBC и режимът CFB е самосинхронизиращ се, но изисква за възстановяване $\lceil n/r \rceil$ правилини блока криптотекст. Скоростта на шифриране се намалява в сравнение със CBC: едно изпълнение на шифриращата трансформация дава дава само r бита криптотекст.

Вариант на ISO за CFB.

5.5.4 Режим **OFB**

Режимът OFB (output feedback mode) може да бъде използван в приложения, при които разпространението на грешки трябва да бъде избягвано. Той е подобен на CFB и позволява шифриране в различни блокови размери. Разпространени са две версии на OFB: версията на ISO – ISO 10116, изискваща n-битов feedback, и по-ранната версия FIPS 81 с feedback от $r < n$ бита.

Алгоритъм OFB с пълен feedback. (ISO 10116)

ВХОД: k битов ключ K, n-битов инициализиращ вектор $\boldsymbol{v}$; r-битови блокове открит текст $x_1, x_2, \ldots, x_u$.

ШИФРИРАНЕ:

(a) $I_1 \longleftarrow \boldsymbol{v}$
(b) `for j=1 to u`
$O_j \longleftarrow E_K(I_j)$
$t_j \longleftarrow$ най-левите r бита на O_j
$c_j \longleftarrow x_j \oplus t_j$
$I_{j+1} \longleftarrow O_j$

ДЕШИФРИРАНЕ:
(a) $I_1 \longleftarrow \boldsymbol{v}$
(b) `for j=1 to u`
$x_j \longleftarrow c_j \oplus t_j$

АЛгоритъмът за шифриране на OFB с r-битов feedback, $r < n$, (FIPS 81) е същият както по-горе с тази разлика, че редът $I_{j+1} \longleftarrow O_j$ се заменя с $I_{j+1} \longleftarrow 2^r \cdot I_j + t_j \pmod{2^n}$.

Както и при режимите CBC и CFB, промяната на $\boldsymbol{v}$ води до промяна на криптотекста при зададен открит текст. Отново не е задължително $\boldsymbol{v}$ да се пази в тайна; секретността на $\boldsymbol{v}$ е съществена, ако един и същ ключ K се използва многократно. Една или повече грешки в някое c_j засяга дешифрирането само на този блок и то точно в позициите, където c_j е сгрешен. Режимът OFB се възстановява от грешки в отделни битове, но не може да се самосинхронизира при загуба на битове от криптотекста. Скоростта на шифриране се намалява както при режим CFB. Тъй като ключовият поток е независим от откритият текст или криптотекста, той може да бъде предварително пресметнат (при известни ключ и инициализращ вектор $\boldsymbol{v}$.

Забележка 5.2. Едно възможно упростяване на OFB се състои в използването на входния блок като брояч: $I_{j+1} = I_j + 1$. Това решава проблема с възстановяването от грешки при пресмятане на E и избягва появата на къси цикли. Освен това можем да дешифрираме блок $i + 1$ без предварително да сме дешифрирали блок i (random access property).

Забележка 5.3. При OFB с пълен n-битов feedback ключовият поток се задава посредством $O_j = E_K(O_{j-1})$. Тъй като E_K е пермутация и K е случайно, то E_K може да се разглежда като случайна пермутация върху множеството от $(2^n)!$ пермутации на n елемента. Може да се покаже [4], че при фиксирани (случайни) ключ и инициализиращ вектор очакваната дължина на цикъл преди повтаряне на стойност на O_j е 2^{n-1}. От друга страна, при използване на r бита feedback, $r < n$, клучовият поток се задава чрез $O_j = f(O_{j-1})$ за някоя функция f, която не е пермутация. Ако допуснем, че тя има поведение на случайна функция, то очакваната дължина на цикъл е $2^{n/2}$. По тази причина се препоръчва режим OFB да се използва с пълен (n-битов) feedback.

Забележка 5.4. И двата алгоритъма, задаващи режим OFB, както и CFB могат да бъдат разглеждани като приложение на блоков шифър за генератор на ключова редица на поточен шифър. Дори режима CBC позволява разглеждане като поточен шифър, при който n битовите блокове са елементи на много голяма азбука. Това позволява въвеждането на поточните шифри чрез режимите на работа на блокови шифри.

[4] Да се формулира задачазадача!!