СУ “Св. Климент Охридски”
Факултет по Математика и Информатика
София

Курсова работа

по

Въведение в криптография и кодове за защита

Тема: Шифърът Salsa20

Изготвил:
Щеряна Шопова М23681
Маг.Програма: ЗИКСМ
СУ “Св. Климент Охридски”, ФМИ

Юни, 2012
София

Съдържание

Въведение

Salsa20 е потоков шифър, представен през 2005 година като част от проекта eSTREAM(2) от Даниел Бърнстейн. Той е построен на основата на псевдо-произволна функция, която включва 32-битово събиране, побитово изключващо ИЛИ (XOR) и ротации, който преобразуват 256-битов ключ, 64-битова случайна стойност (nonce, или IV – инициализиращ вектор), и 64-битова позиция от потока в 512-битов изход. Съществува и версия на алгоритъма с по-малък - 128-битов ключ. Това дава на Salsa20 необичайното свойство, че потребителят може ефективно да намери стойността на произволна позиция в изходящия поток. Скоростта на изпълнение на алгоритъма е от порядъка на 4-14 процесорни цикъла за байт в софтуерната реализация на алгоритъма на съвременни x86 процесори с подходящия хардуер. Алгоритъмът не е патентован и авторът му е предоставил няколко отворени имплементации, оптимизирани за най-често срещаните съвременни архитектури.

Структура

Вътрешно шифърът използва побитово събиране (изключващо ИЛИ), 32-битово събиране по модул 2^{32} , и ротации на фиксиран брой позиции спрямо вътрешно състояние от 16 32-битови думи. Изборът на операции е направен така, че да се избегне възможността за реализирането на времеви атаки (или по-общо атаки чрез скрити канали) в софтуерните реализации на *Salsa20*, които например могат да се приложат успешно (3,4) за възстановяването на частен *AES (5)* ключ при някои имплементации на алгоритъма *AES*.

Първоначалното състояние на *Salsa20* алгоритъма се конструира от 8те думи на ключа, 2 думи, представляващи позициите в потока, 2 думи от случайни стойности (инициализиращ вектор, по същество представляват допълнителени битове с позиции в потока), и 4 фиксирани думи. 20 кръга от една итерация от изпълнението на алгоритъма генерират 16 думи от изходния шифрован текст. Всяка итерация се състои от 4 операции, които се извършват върху редовете или колоните на 16-битовата дума, която представя вътрешното състояние на алгоритъма чрез квадратна матрица с размерност 4. На всеки две итерации, шаблонът се повтаря. На следващата фигура е позакан псевдокода за първите две итерации на алгоритъма.

Фигура 1. *Псевдокод за първите две итерации на Salsa20 алгоритъма*

```
Означение на операциите

⊞ - събиране по модул 2^32
<<< - ротация на битове наляво
^ - изключващо или
x ^= y е съкращение на израза x = x ^ y


x[ 4] ^= (x[ 0] ⊞ x[12])<<<7;    x[ 9] ^= (x[ 5] ⊞ x[ 1])<<<7;
x[14] ^= (x[10] ⊞ x[ 6])<<<7;    x[ 3] ^= (x[15] ⊞ x[11])<<<7;
x[ 8] ^= (x[ 4] ⊞ x[ 0])<<<9;    x[13] ^= (x[ 9] ⊞ x[ 5])<<<9;
x[ 2] ^= (x[14] ⊞ x[10])<<<9;    x[ 7] ^= (x[ 3] ⊞ x[15])<<<9;
x[12] ^= (x[ 8] ⊞ x[ 4])<<<13;   x[ 1] ^= (x[13] ⊞ x[ 9])<<<13;
x[ 6] ^= (x[ 2] ⊞ x[14])<<<13;   x[11] ^= (x[ 7] ⊞ x[ 3])<<<13;
x[ 0] ^= (x[12] ⊞ x[ 8])<<<18;   x[ 5] ^= (x[ 1] ⊞ x[13])<<<18;
x[10] ^= (x[ 6] ⊞ x[ 2])<<<18;   x[15] ^= (x[11] ⊞ x[ 7])<<<18;

x[ 1] ^= (x[ 0] ⊞ x[ 3])<<<7;    x[ 6] ^= (x[ 5] ⊞ x[ 4])<<<7;
x[11] ^= (x[10] ⊞ x[ 9])<<<7;    x[12] ^= (x[15] ⊞ x[14])<<<7;
x[ 2] ^= (x[ 1] ⊞ x[ 0])<<<9;    x[ 7] ^= (x[ 6] ⊞ x[ 5])<<<9;
x[ 8] ^= (x[11] ⊞ x[10])<<<9;    x[13] ^= (x[12] ⊞ x[15])<<<9;
x[ 3] ^= (x[ 2] ⊞ x[ 1])<<<13;   x[ 4] ^= (x[ 7] ⊞ x[ 6])<<<13;
x[ 9] ^= (x[ 8] ⊞ x[11])<<<13;   x[14] ^= (x[13] ⊞ x[12])<<<13;
x[ 0] ^= (x[ 3] ⊞ x[ 2])<<<18;   x[ 5] ^= (x[ 4] ⊞ x[ 7])<<<18;
x[10] ^= (x[ 9] ⊞ x[ 8])<<<18;   x[15] ^= (x[14] ⊞ x[13])<<<18;
```

Пълният *Salsa20* алгоритъм използва 20 итерации за получаване на изходния шифър, но към допълнение към оригиналния алгоритъм са дефинирани и вариантите *Salsa20/8* и *Salsa20/12* с намален брои итерации – 8 и съответно 12. Целта е тези варинати да допълнят, а не да заменят оригиналния алгоритъм, като могат да се ползват за приложения, при които се допуска компромис със сигурността на алгоритъма за сметка на по-бързата обработка на информацията.

Избор за участие в проекта eSTREAM

eSTREAM е прокт на европейската инициатива *ECRYPT* за рабработка и изследване свойствата на нови симетрични алгоритми/поточни шифри, които са подходящи за широка употреба.

Алгоритмите предложени за анализ в рамките на проекта попадат в един от два профила

- *профил 1* – алгоритми за софтуерна реализация за приложения, които обменят големи количества криптирани данни
- *профил 2* - алгоритми за хардуерна реализация на платформи с ограничени ресурсни, вкл. налична памет, консумация на ток и др.

И двата профила дефинират и “А” подпрофили за алгоритми, които освен криптиране на данните, предоставят и методи за аутентикация. Въпреки това, нито един от алгоритмите с аутентикация не достига до третата фаза на проекта, тъй като тази допълнителна възможност влошава бързодействието на самите алгоритми.

Проектът *eSTREAM* протича на три етапа (фази) -

- фаза 1 – обстоен анализ на всички алгоритми, предложени за участие в проекта; всеки предложен алгоритъм бива проучен и евентуално одобрен за по-нататъшно участие в проекта на базата на няколко определящи фактора – сигурност, производителност (на основа на сравнение с производителността на няколко стандартни алгоритми, вкл. и *AES-128),* простота и гъвкавост на алгоритъма, обстоен анализ и доказателства на качествата на алгоритъма, както и яснота и пълнота на съпровождащата го документация
- фаза 2 – в рамките на тази фаза, проектът определя алгоритмите, които заслужават специален интерес и подлага свойствата им на допълнителен анализ, включително оценка на производителността и обстоен криптоанализ; до края на фаза 2 отпадат всички предложени алгоритми, срещи които има открити успешни криптографски атаки, т.е. постановка за отгатване на частния ключ по-бързо от прилагане на brute force атака
- фаза 3 – в края на фаза 3 остават само няколко алгоритъма, одобрени от проекта
    - профил 1 – алгоритмите *HC-128, Rabbit, Salsa20/12 и SOSEMANUC*
    - профил 2 - алгоритмите *F-FCSR-H v2, Grain v1, Mickey v2 и Trivium*

Интересно е да се отбележи, че *Salsa20* получава най-високата сборна оценка от алгоритмите одобрени в края на 3тия етап в профил 1 (софтуерна имплементация), поради което именно този алгоритъм е описан в настоящата разработка.

Криптанализ на Salsa20

Към момента не съществуват публикувани атаки на *Salsa20/12* или пълната версия на алгоритъма *Salsa20/20* . Най-добрата позната атака (6) успява да компрометира 8те кръга на *Salsa20/8.* Досега публикуваните работи, свързани с криптанализа на *Salsa20* са следните

- 2005 - Paul Crowley осъществава атака на алгоритъма *Salsa20/5* със сложност откъм време 2^{165} , тази и всички последвали атаки на алгоритъма се основават на диференциалния криптоанализ (7). През 2006 г. Fischer, Meier, Berbain, Biasse, и Robshaw демонстрират атака на *Salsa20/6* с теоретична сложността от 2^{177} .
- През 2007 г. Tsunoo и др. обяви криптоанализ на Salsa20 който разбива 8 от 20те кръга, така че успява да възстанови 256 частен битов ключ в 2^{255} операции, използвайки $2^{11.37}$ двойки ключ/шифриран текст

Детайлно описание на Salsa20

Основата на *Salsa20* алгоритъма е хеш функция с 64-байта входни и 64-байта изходни данни. Хеш функцията се използва като потоков шифър - *Salsa20* работи с отделните 64-байтови блокове от изходния открит текст, като за всеки такъв блок от изходния текст, намира 64-байтов хеш, на базата на ключа, инициализиращия вектор и номера на блока, като така получената стойност бива XOR-ната с първоначалните 64-байта от изходния открит текст, а получената стойност представлява крайния шифрован текст за този блок.

В тази глава ще опишем в детайли работата на алгоритъма, като започнем с три прости операции върху 4-байтови думи, преминем към хеш функцията на алгоритъма, и завършим с криптиращата функция на алгоритъма. Оттук нататък, считаме че байт е елемент от

множеството {1,2, .., 255 }. Други дефиниции на понятето и представянето на байт като поредица от електрически сигнали нямат отношение по изложението на алгоритъма.

Дефиниции

Дума – елемент от множеството $\{1, 2, \ldots, \quad 2^{32}-1 \quad\}$. Думите ще записваме и като шестнасетични числа, предхождани от символа ***0x,*** така например *0xc0a8787e* = $12*2^{28}+0*2^{24}+10*2^{20}+8*2^{16}+7*2^{12}+8*2^{8}+7*2^{4}+14*2^{0} \quad = 3232266366$.

Под ***сума*** на две думи U и V, имаме предвид $U + V \ \ mod \quad 2^{32}$. Където не съществува опасност от двусмислие, ще означаваме тази сума накратко с $U + V$. Така например *0xc0a8787e + 0x9fd1161d = 0x60798e9b.*

Изключващо ИЛИ на две думи U и V, означаваме като $U \oplus V$ и дефинираме като сумата на две думи без да отчитаме пренос, т.е. ако $U = \sum 2^i * U_i$, $V = \sum 2^i * V_i$, то $U \oplus V = \sum 2^i (U_i + V_i - 2 * U_i * V_i)$. Така например $0\text{xc0a8787e} \oplus 0\text{x9fd1161d} = 0x60798e9b$.

За всяко $c \in \{0, 1, 2, 3, \ldots\}$ ***Ротация на c-бита наляво*** на думата U, означаваме с $U <<c$, и дефинираме като уникална ненулева дума съответна на $2^c * u \ \ mod \quad 2^{32}-1$ **,** като свитаме, че $0 << c = 0$. С други думи, ако $U = \sum 2^i * U_i$, то $U << c = \sum 2^{(i+c \bmod 32)} * U_i$. Например, *0xc0a8787e << 5 = 0x150f0fd8.*

Функция quaterround(x)

Вход: Ако y е поредица от 4 думи, то резултатът от *quaterround(y),* също е поредица от 4 думи.

Дефиниция: Ако $y = (y0, y1, y2, y3)$, то *quaterround(y) = (z0, z1, z2, z3),* където

$$z1 = \quad y1 \oplus ((y0 + y3) \lll 7) \quad ,$$

$$z2 = \quad y2 \oplus ((z2 + y0) \lll 9) \quad ,$$

$$z3 = \quad y3 \oplus ((z2 + z1) \lll 13) \quad ,$$

$$z0 = \quad y0 \oplus ((z3 + z1) \lll 18)$$

Примери:

quaterround(0x00000000, 0x00000000, 0x00000000, 0x00000000) =
(0x00000000, 0x00000000, 0x00000000, 0x00000000)
quaterround(0x00000001, 0x00000000, 0x00000000, 0x00000000) =
(0x08008145, 0x00000080, 0x00010200, 0x20500000)
quaterround(0x00000000, 0x00000001, 0x00000000, 0x00000000) =
(0x88008145, 0x00000080, 0x00010200, 0x20500000)
quarterround(0x00000000, 0x00000000, 0x00000001, 0x00000000) =
(0x80040000, 0x00000000, 0x00000001, 0x00002000)
quarterround(0x00000000, 0x00000000, 0x00000000, 0x00000001) =
(0x00048044, 0x00000080, 0x00010000, 0x20100001).

quarterround(0xe7e8c006, 0xc4f9417d, 0x6479b4b2, 0x68c67137) =

(0xe876d72b, 0x9361dfd5, 0xf1460244, 0x948541a3).

quarterround(0xd3917c5b, 0x55f1c407, 0x52a58a7a, 0x8f887a3b) =

(0x3e2f308c, 0xd90a8f36, 0x6ab2a923, 0x2883524c).

Коментар: *quaterround(y)* функцията може да бъде представена като заместване на *y* на място, първо *y1* се заменя от *z1*, *y2* от *z2*, *y3* от *z3*, *y0* от *z0*. Всяка промяна е необратима, и така цялата функция е необратима.

Функция rowround(x)

Вход: Ако *y* е поредица от 16 думи, то резултатът от *rowround(y),* също е поредица от 16 думи.

Дефиниция: Ако *y = (y0, y1, y2, y3, ..., y15),* то *rowround(y) = (z0, z1, z2, z3, ..., z15),* където

(*z0, z1, z2, z3) = quarterround(y0, y1, y2, y3),*

(*z5, z6, z7, z4) = quarterround(y5, y6, y7, y4),*

(*z10, z11, z8, z9) = quarterround(y10, y11, y8, y9),*

(*z15, z12, z13, z14) = quarterround(y15, y12, y13, y14)*

Примери:

rowround(0x00000001, 0x00000000, 0x00000000, 0x00000000,

0x00000001, 0x00000000, 0x00000000, 0x00000000,

0x00000001, 0x00000000, 0x00000000, 0x00000000,

0x00000001, 0x00000000, 0x00000000, 0x00000000)

= (0x08008145, 0x00000080, 0x00010200, 0x20500000,

0x20100001, 0x00048044, 0x00000080, 0x00010000,

0x00000001, 0x00002000, 0x80040000, 0x00000000,

0x00000001, 0x00000200, 0x00402000, 0x88000100).

rowround(0x08521bd6, 0x1fe88837, 0xbb2aa576, 0x3aa26365,

0xc54c6a5b, 0x2fc74c2f, 0x6dd39cc3, 0xda0a64f6,

0x90a2f23d, 0x067f95a6, 0x06b35f61, 0x41e4732e,

0xe859c100, 0xea4d84b7, 0x0f619bff, 0xbc6e965a)

= (0xa890d39d, 0x65d71596, 0xe9487daa, 0xc8ca6a86,

0x949d2192, 0x764b7754, 0xe408d9b9, 0x7a41b4d1,

0x3402e183, 0x3c3af432, 0x50669f96, 0xd89ef0a8,

0x0040ede5, 0xb545fbce, 0xd257ed4f, 0x1818882d).

Коментар: Входът на функцията *rowround(y)* може да бъде представен като квадратна матрица:

y0	*y1*	*y2*	*y3*
y4	*y5*	*y6*	*y7*
y9	*y10*	*y11*	*y12*
y12	*y13*	*y14*	*y15*

Функцията *rowround(y)* променя редовете на матрицата чрез функцията *quaterround(y),* като едновременно с това прави пермутация в реда.

Функция columnround(x)

Вход: Ако *x* е поредица от 16 думи, то резултатът от *columnround(x),* също е поредица от 16 думи.

Дефиниция: Ако *x = (x0, x1, x2, x3, ..., x15),* то *columnround(x) = (y0, y1, y2, y3, ..., y15),* където

(*y0, y4, y8, y12) = quarterround(x0, x4, x8, x12),*

(y*5, y9, y13, y1) = quarterround(x5, x9, x13, x1),*

(y10, y14, y2, y6) = quarterround(x10, x14, x2, x6)

(y15, y3, y7, y11) = quarterround(x15, x3, x7, x11)

Примери:

columnround(0x00000001, 0x00000000, 0x00000000, 0x00000000,
0x00000001, 0x00000000, 0x00000000, 0x00000000,
0x00000001, 0x00000000, 0x00000000, 0x00000000,
0x00000001, 0x00000000, 0x00000000, 0x00000000)
= (0x10090288, 0x00000000, 0x00000000, 0x00000000,
0x00000101, 0x00000000, 0x00000000, 0x00000000,
0x00020401, 0x00000000, 0x00000000, 0x00000000,
0x40a04001, 0x00000000, 0x00000000, 0x00000000).

columnround(0x08521bd6, 0x1fe88837, 0xbb2aa576, 0x3aa26365,
0xc54c6a5b, 0x2fc74c2f, 0x6dd39cc3, 0xda0a64f6,
0x90a2f23d, 0x067f95a6, 0x06b35f61, 0x41e4732e,
0xe859c100, 0xea4d84b7, 0x0f619bff, 0xbc6e965a)
= (0x8c9d190a, 0xce8e4c90, 0x1ef8e9d3, 0x1326a71a,
0x90a20123, 0xead3c4f3, 0x63a091a0, 0xf0708d69,
0x789b010c, 0xd195a681, 0xeb7d5504, 0xa774135c,
0x481c2027, 0x53a8e4b5, 0x4c1f89c5, 0x3f78c9c8).

Коментар: Входът на функцията *columnround(x)* може да бъде представен като квадратна матрица:

x0	*x1*	*x2*	*x3*
x4	*x5*	*x6*	*x7*
x9	*x10*	*x11*	*x12*
x12	*x13*	*x14*	*x15*

най-просто казано, функцията *columnround(x)* е транспозиция на функцията *rowround(y)* - променя колоните на матрицата чрез функцията *quaterround(y),* като едновременно с това прави пермутация в колоната.

Функция doubleround(x)

Вход: Ако *x* е поредица от 16 думи, то резултатът от *doubleround(x)* също е поредица от 16 думи.

Дефиниция: Ако *x = (x0, x1, x2, x3, ..., x15),* то *doubleround(x) = (y0, y1, y2, y3, ..., y15),* където

(y0, y1, y2, y3, ..., y15) = rowround (columnround(x0, x1, x2, x3, ..., x15))

Примери:

doubleround(0x00000001, 0x00000000, 0x00000000, 0x00000000,
0x00000000, 0x00000000, 0x00000000, 0x00000000,
0x00000000, 0x00000000, 0x00000000, 0x00000000,
0x00000000, 0x00000000, 0x00000000, 0x00000000)
= (0x8186a22d, 0x0040a284, 0x82479210, 0x06929051,
0x08000090, 0x02402200, 0x00004000, 0x00800000,
0x00010200, 0x20400000, 0x08008104, 0x00000000,
0x20500000, 0xa0000040, 0x0008180a, 0x612a8020).

doubleround(0xde501066, 0x6f9eb8f7, 0xe4fbbd9b, 0x454e3f57,
0xb75540d3, 0x43e93a4c, 0x3a6f2aa0, 0x726d6b36,
0x9243f484, 0x9145d1e8, 0x4fa9d247, 0xdc8dee11,
0x054bf545, 0x254dd653, 0xd9421b6d, 0x67b276c1)
= (0xccaaf672, 0x23d960f7, 0x9153e63a, 0xcd9a60d0,
0x50440492, 0xf07cad19, 0xae344aa0, 0xdf4cfdfc,
0xca531c29, 0x8e7943db, 0xac1680cd, 0xd503ca00,
0xa74b2ad6, 0xbc331c5c, 0x1dda24c7, 0xee928277).

Коментар: Функцията *doubleround(x)* модифицира едновременно редовете и колоните – всяка дума от входа се модифицира по два пъти.

Функция littleendian(x)

Вход: Ако *b* е поредица от 4 байта, то резултатът от *littleendian(x)* е дума.

Дефиниция: Ако *b = (b0, b1, b2, b3),* то *littleendian(b) =* $b0 + 2^8 * b1 + 2^{16} * b2 + 2^{24} * b3$

Примери:

littleendian(0, 0, 0, 0) = 0x00000000.

littleendian(86, 75, 30, 9) = 0x091e4b56.

littleendian(255, 255, 255, 250) = 0xfaffffff.

Коментар: Функцията *littleendian(x)* също е необратима.

Хешираща функция на Salsa20

Вход: Ако *x* е поредица от 64 байта, то резултатът от *Salsa20(x)* също е поредица от 64 байта

Дефиниция:

На кратко:

$Salsa20(x) = x + (doubleround(x))^{10}$, където всяка поредица от 4-байта се разглежда като дума в *little endian* представяне.

По-точно, нека *x = (x[0], x[1], x[2], ..., x[63]),* дефинираме

x0 = littleendian(x[0], x[1], x[2], x[3]),

x1 = littleendian(x[4], x[5], x[6], x[7]),

x2 = littleendian(x[8], x[9], x[10], x[11]),

...

x15 = littleendian(x[60], x[61], x[62], x[63]).

Дефинираме $(z0, z1, \ldots, z15) = (doubleround(x0, x1, \ldots, x15))^{10}$.

Тогава *Salsa20(x)* е конкатенацията на

$littleendian^{-1}(z0 + x0)$,

$littleendian^{-1}(z1 + x1)$,

$littleendian^{-1}(z2 + x2)$,

...

$littleendian^{-1}(z15 + x15)$,

Примери:

Salsa20(0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0) =
(0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0)

Salsa20(211,159, 13,115, 76, 55, 82,183, 3,117,222, 37,191,187,234,136,
49,237,179, 48, 1,106,178,219,175,199,166, 48, 86, 16,179,207,
31,240, 32, 63, 15, 83, 93,161,116,147, 48,113,238, 55,204, 36,

79,201,235, 79, 3, 81,156, 47,203, 26,244,243, 88,118,104, 54)

= (109, 42,178,168,156,240,248,238,168,196,190,203, 26,110,170,154,

29, 29,150, 26,150, 30,235,249,190,163,251, 48, 69,144, 51, 57,

118, 40,152,157,180, 57, 27, 94,107, 42,236, 35, 27,111,114,114,

219,236,232,135,111,155,110, 18, 24,232, 95,158,179, 19, 48,202).

Salsa20(88,118,104, 54, 79,201,235, 79, 3, 81,156, 47,203, 26,244,243,

191,187,234,136,211,159, 13,115, 76, 55, 82,183, 3,117,222, 37,

86, 16,179,207, 49,237,179, 48, 1,106,178,219,175,199,166, 48,

238, 55,204, 36, 31,240, 32, 63, 15, 83, 93,161,116,147, 48,113)

= (179, 19, 48,202,219,236,232,135,111,155,110, 18, 24,232, 95,158,

26,110,170,154,109, 42,178,168,156,240,248,238,168,196,190,203,

69,144, 51, 57, 29, 29,150, 26,150, 30,235,249,190,163,251, 48,

27,111,114,114,118, 40,152,157,180, 57, 27, 94,107, 42,236, 35).

Извеждаща функция на Salsa20

Вход: Ако *к* е поредица от 16 или 32 байта и *n* е поредица от 16 байта, то $Salsa20_k(n)$ е поредица от 64 байта.

Дефиниция:

Дефинираме $\sigma_0 = (101, 120, 112, 97)$,

$\sigma_1 = (110, 100, 32, 51)$,

$\sigma_2 = (50, 45, 98, 121)$,

$\sigma_3 = (116, 101, 32, 107)$

Ако *k0, k1* и *n* са поредици с по 16 байта, то

$Salsa20_{(k0,к1)}(n) = Salsa20(\sigma_0, k_0, \sigma_1, n, \sigma_2, k_1, \sigma_3)$

Дефинираме $\tau_0 = (101, 120, 112, 97)$,

$\tau_1 = (110, 100, 32, 49)$,

$\tau_2 = (54, 45, 98, 121)$,

$\tau_3 = (116, 101, 32, 107)$

Ако *k* и *n* са поредици с по 16 байта, то

$Salsa20_k(n) = Salsa20(\tau_0, k, \tau_1, n, \tau_2, k, \tau_3)$

Примери:

Нека k0 = (1, 2, 3, 4, 5, . . . , 16), k1 = (201, 202, 203, 204, 205, . . . , 216), и n =

(101, 102, 103, 104, 105, . . . , 116). Тогава

$Salsa20_{(k0,к1)}(n)$ = *Salsa20(101,120,112, 97, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16,110,100, 32, 51,101,102,103,104,105,106,107,108, 109,110,111,112,113,114,115,116, 50, 45, 98,121,201,202,203,204, 205,206,207,208,209,210,211,212,213,214,215,216,116,101, 32,107)*

= *(69, 37, 68, 39, 41, 15,107,193,255,139,122, 6,170,233,217, 98, 89,144,182,106, 21, 51,200, 65,239, 49,222, 34,215,114, 40,126, 104,197, 7,225,197,153, 31, 2,102, 78, 76,176, 84,245,246,184, 177,160,133,130, 6, 72,149,119,192,195,132,236,234,103,246, 74)*

и

$Salsa20_k(n)$ = *Salsa20(101,120,112, 97, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 4, 13, 14, 15, 16,110,100, 32, 49,101,102,103,104,105,106,107,108, 109,110,111,112,113,114,115,116, 54, 45, 98,121, 1, 2, 3, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16,116,101, 32,107)*

= *(39,173, 46,248, 30,200, 82, 17, 48, 67,254,239, 37, 18, 13,247, 241,200, 61,144, 10, 55, 50,185, 6, 47,246,253,143, 86,187,225, 134, 85,110,246,161,163, 43,235,231, 94,171, 51,145,214,112, 29, 14,232, 5, 16,151,140,183,141,171, 9,122,181,104,182,177,193).*

Криптиращата функция на Salsa20

Вход: Нека *к* е поредица от 16 или 32 байта. Нека *v* е поредица от 8 байта. Нека *m* е поредица от *l* байта като $l \in \{0, 1, ..., 2^{70}\}$ и *n* е поредица от 16 байта, то $Salsa20_k(n)$ е поредица от 64 байта. Изходът на криптиращата функция *Salsa20* за *m* с инициализиращ вектор *v* и ключ *k* е поредица от *l* байта. Функцията означаваме с $Salsa20_k(v) \oplus m$.

Обикновено, *к* е таен ключ (за предпочитане с дължина 32 байта), *v* е произволна стойност, която се използва за инициализация, *m* е открития изходен текст, а изходът на функцията - $Salsa20_k(v) \oplus m$ - шифрования текст. Или обратното, ако *m* е шифрования текст, то изходът на $Salsa20_k(v) \oplus m$ е съответстващия му открит текст.

Дефиниция:

Нека $Salsa20_k(v)$ е следната поредица от 2^{70} байта -

$Salsa20_k(v,0), Salsa20_k(v,1), Salsa20_k(v,3), ..., Salsa20_k(v,2^{64}-1)$.

С *i* означаваме уникалната поредица от 8 байта ($i_0, i_1, ..., i_7$), такава че

$i = i_0 + 2^8 * i_1 + 2^{16} * i_2 + ... + 2^{56} * i_7$

Формулата $Salsa20_k(v) \oplus m$ неявно отрязва поредицата $Salsa20_k(v)$, така че да е със същата дължина като входната поредица, т.е m. С други думи,

$Salsa20_k(v) \oplus (m[0], m[1], \ldots, m[l-1]) = (c[0], c[1], \ldots, c[l-1])$, където

$c[i] = m[i] \oplus Salsa20_k(v, \lfloor i/64 \rfloor)[i \bmod 64]$

В ***Приложение 1*** е дадена примерна имплементация на алгоритъма, реализирана на *C*.

Приложение 1: Примерна имплементация на Salsa20

```
#include <sys/types.h>
#ifdef  LINUX
#include <limits.h>
#else
#include <sys/limits.h>
#endif /* LINUX */
#include <sys/socket.h>

#include <arpa/inet.h>
#include <netinet/in.h>

#include <err.h>
#include <errno.h>
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <signal.h>
#include <unistd.h>

#define SALSA20_MAXKEYSIZE    256
#define SALSA20_KEYSIZE(i)    (128 + (i)*128)

#define SALSA20_MAXIVSIZE     64
#define SALSA20_IVSIZE(i)     (64 + (i)*64)

#define  ROTL32(v, n)         (((v) << (n)) | ((v) >> (32 - (n))))
#define ROTATE(v, c)          (ROTL32((v), (c)))
#define XOR(v, w)             ((v) ^ (w))
#define PLUS(v,w)             ((v) + (w))
#define  PLUSONE(v)           (PLUS((v), 1))
#define  CLRBITS(v)
#define LITTLEEND32(v)               \
        ((uint32_t)((v)[0])) |       \
        ((uint32_t)((v)[0]) << 8) |  \
        ((uint32_t)((v)[0]) << 16) | \
        ((uint32_t)((v)[0]) << 24)
```

```
#define LITTLEEND32_REV(p, v) do {                          \
        ((uint32_t *)p)[0] = (v) & 0xFF;                    \
        ((uint32_t *)p)[1] = ((v) & 0xFF00) >>  8;          \
        ((uint32_t *)p)[2] = ((v) & 0xFF0000) >>  16;               \
        ((uint32_t *)p)[3] = ((v) & 0xFF000000) >>  24;             \
} while (0)


struct crypt_ctx {
        uint32_t       input[16];
};

/* Forward declarations */
static void salsa20_init(struct crypt_ctx *);
static void salsa20_keysetup(struct crypt_ctx *, const uint8_t *, uint32_t);
static void salsa20_ivsetup(struct crypt_ctx *, const uint8_t *);
static void salsa20_encrypt(struct crypt_ctx *, const uint8_t *,  uint8_t *, uint32_t);
static void salsa20_decrypt(struct crypt_ctx *, const uint8_t *,  uint8_t *, uint32_t);

static const char sigma[] = "expand 32-byte k";
static const char tau[] = "expand 16-byte k";

static void
salsa20_init(struct crypt_ctx *ctx)
{
        memset(ctx, 0, sizeof(struct crypt_ctx));
}

static void
salsa20_keysetup(struct crypt_ctx *ctx, const uint8_t * key, uint32_t keysize)
{
        const char *consts;

        ctx->input[1] = LITTLEEND32(key + 0);
        ctx->input[2] = LITTLEEND32(key + 4);
        ctx->input[3] = LITTLEEND32(key + 8);
        ctx->input[4] = LITTLEEND32(key + 12);

        if (keysize == 256) {
                key += 16;
                consts = sigma;
        } else
                consts = tau;
```

```
        ctx->input[11] = LITTLEEND32(key + 0);
        ctx->input[12] = LITTLEEND32(key + 4);
        ctx->input[13] = LITTLEEND32(key + 8);
        ctx->input[14] = LITTLEEND32(key + 12);
        ctx->input[0] = LITTLEEND32(consts + 0);
        ctx->input[5] = LITTLEEND32(consts + 4);
        ctx->input[10] = LITTLEEND32(consts + 8);
        ctx->input[15] = LITTLEEND32(consts + 12);
}

static void
salsa20_ivsetup(struct crypt_ctx *ctx, const uint8_t * iv)
{
        ctx->input[6] = LITTLEEND32(iv + 0);
        ctx->input[7] = LITTLEEND32(iv + 4);
        ctx->input[8] = 0;
        ctx->input[9] = 0;


}

static void
salsa20_encrypt(struct crypt_ctx *ctx, const uint8_t *plaintext,  uint8_t *ciphertext, uint32_t msglen)
{
        uint32_t x[16], j[16];
        uint8_t *ctarget;
        uint8_t tmp[64];
        int32_t i;

        if (msglen == 0)
                return;

        for (i = 0; i < 16; ++i)
                j[i] = ctx->input[i];

        for ( ; ; ) {
                if (msglen < 64) {
                        for (i = 0; i < msglen; ++i)
                                tmp[i] = plaintext[i];
                        plaintext = tmp;
                        ctarget = ciphertext;
                        ciphertext = tmp;
```

```
}

for (i = 0; i < 16; ++i)
       x[i] = j[i];

for (i = 20; i > 0; i -= 2) {
       x[4] = XOR(x[4], ROTATE (PLUS(x[0],x[12]), 7));
       x[8] = XOR(x[8], ROTATE(PLUS(x[4], x[0]), 9));
       x[12] = XOR(x[12], ROTATE(PLUS(x[8], x[4]), 13));
       x[0] = XOR(x[0], ROTATE(PLUS(x[12], x[8]), 18));
       x[9] = XOR(x[9], ROTATE(PLUS(x[5], x[1]), 7));
       x[13] = XOR(x[13], ROTATE(PLUS(x[9], x[5]), 9));
       x[1] = XOR(x[1], ROTATE(PLUS(x[13], x[9]), 13));
       x[5] = XOR(x[5], ROTATE(PLUS(x[1], x[13]), 18));
       x[14] = XOR(x[14], ROTATE(PLUS(x[10], x[6]), 7));
       x[2] = XOR(x[2], ROTATE(PLUS(x[14],x[10]), 9));
       x[6] = XOR(x[6], ROTATE(PLUS(x[2], x[14]), 13));
       x[10] = XOR(x[10],ROTATE(PLUS(x[6], x[2]), 18));
       x[3] = XOR(x[3], ROTATE(PLUS(x[15], x[11]), 7));
       x[7] = XOR(x[7], ROTATE(PLUS(x[3], x[15]), 9));
       x[11] = XOR(x[11], ROTATE(PLUS( x[7], x[3]), 13));
       x[15] = XOR(x[15], ROTATE(PLUS(x[11], x[7]), 18));
       x[1] = XOR(x[1], ROTATE(PLUS(x[0], x[3]), 7));
       x[2] = XOR(x[2], ROTATE(PLUS(x[1], x[0]), 9));
       x[3] = XOR(x[3], ROTATE(PLUS(x[2], x[1]), 13));
       x[0] = XOR(x[0], ROTATE(PLUS(x[3], x[2]), 18));
       x[6] = XOR(x[6], ROTATE(PLUS(x[5], x[4]), 7));
       x[7] = XOR(x[7], ROTATE(PLUS(x[6], x[5]), 9));
       x[4] = XOR(x[4], ROTATE(PLUS(x[7], x[6]), 13));
       x[5] = XOR(x[5], ROTATE(PLUS(x[4], x[7]), 18));
       x[11] = XOR(x[11], ROTATE(PLUS(x[10], x[9]), 7));
       x[8] = XOR(x[8], ROTATE(PLUS(x[11], x[10]), 9));
       x[9] = XOR(x[9], ROTATE(PLUS(x[8], x[11]), 13));
       x[10] = XOR(x[10], ROTATE(PLUS(x[9], x[8]), 18));
       x[12] = XOR(x[12], ROTATE(PLUS(x[15], x[14]), 7));
       x[13] = XOR(x[13], ROTATE(PLUS(x[12], x[15]), 9));
       x[14] = XOR(x[14], ROTATE(PLUS(x[13], x[12]), 13));
       x[15] = XOR(x[15], ROTATE(PLUS(x[14], x[13]), 18));
}

for (i = 0; i < 16; ++i)
       x[i] = PLUS(x[i], j[i]);
```

```
                for (i = 0; i < 16; ++i)
                        x[i] = XOR(x[i], LITTLEEND32(plaintext + i * 4));

                j[8] = PLUSONE(j[8]);
                if (j[8] != 0)
                        j[9] = PLUSONE(j[9]); /* stopping at 2^70 bytes per nonce is user's
responsibility */

                for (i = 0; i < 16; ++i)
                        LITTLEEND32_REV(ciphertext + i * 4, x[i]);

                if (msglen <= 64) {
                        if (msglen < 64) {
                                for (i = 0; i < msglen ;++i)
                                        ctarget[i] = ciphertext[i];
                        }
                        ctx->input[8] = j[8];
                        ctx->input[9] = j[9];
                        return;
                }
                msglen -= 64;
                ciphertext += 64;
                plaintext += 64;
        }
}

static void
salsa20_decrypt(struct crypt_ctx *ctx, const uint8_t *ciphertext,  uint8_t *plaintext, uint32_t msglen)
{
        salsa20_encrypt(ctx, ciphertext, plaintext, msglen);
}

int
main(int argc, char **argv, char **envp)
{
        int32_t status, nbytes, i, readb;
        int fd = -1, kfd = -1, ofd = -1;
        uint8_t key[256], *message = NULL, *cipher, op = 0;
        int32_t iv = random();
        struct crypt_ctx ctx;

        if (argc != 5)
                errx(1, "Usage: ./salsa20 <op> <hex-key> <file-to-encrypt> <output-file>\n");
```

```
if (argv[1][0] == 'd')
        op = 1;

status = -1;
if ((fd = open(argv[2], O_RDONLY)) == -1) {
        fprintf(stderr, "Failed open source file %s - %s", argv[2], strerror(errno));
        goto cleanup;
}

if ((kfd = open(argv[3], O_RDONLY)) == -1) {
        fprintf(stderr, "Failed open key file %s - %s", argv[3], strerror(errno));
        goto cleanup;
}

if ((ofd = open(argv[4], O_WRONLY | O_APPEND)) == -1) {
        fprintf(stderr, "Failed open output file %s - %s", argv[4], strerror(errno));
        goto cleanup;
}

for (nbytes = 0, i = 0;  ; i++) {
        if ((readb = read(fd, key, 256)) <= 0) {
                /* Reopen file to position at beggining */
                if ((fd = open(argv[2], O_RDONLY)) == -1) {
                        fprintf(stderr, "Failed reopen source %s", strerror(errno));
                        goto cleanup;
                }
                break;
        }
        nbytes += readb;
}

if ((message = (uint8_t *)malloc((nbytes + 64) * 2)) == NULL) {
        fprintf(stderr, "Failed to allcate memory %s", strerror(errno));
        goto cleanup;
}
memset(message, 0, (nbytes + 64) * 2);
memset(key, 0, 256);

(void) read(kfd, key, 256);
cipher = message + nbytes + 64;
(void) read(fd, message, nbytes);

fprintf(stderr, "IV is set to 0x%x\n", *((uint32_t *) &iv));
```

```
        salsa20_init(&ctx);
        salsa20_keysetup(&ctx, key, 256);
        salsa20_ivsetup(&ctx, (uint8_t *) &iv);

        if (op)
                salsa20_decrypt(&ctx, message, cipher, nbytes);
        else
                salsa20_encrypt(&ctx, message, cipher, nbytes);

        if ((readb = write(ofd, cipher, nbytes)) <= 0) {
                fprintf(stderr, "Failed to write output cipher - %s", strerror(errno));
                goto cleanup;
        }

cleanup:
        if (status != 0)
                status = errno;
        if (fd != -1)
                close(fd);
        if (kfd != -1)
                close(kfd);
        if (ofd != -1)
                close(ofd);
        if (message != NULL)
                free(message);
        exit(status);

}
```

Използвани материали:

1. *Официална страница на шифъра Salsa20 - http://cr.yp.to/snuffle.html*
2. *Проектът eSTREAM - http://www.ecrypt.eu.org/stream/project.html*
3. *Colin Percival: Cache Missing for Fun and Profit, May 13, 2005 - http://www.daemonology.net/papers/htt.pdf*
4. *Daniel J. Bernstein: Cache timing attacks on AES – http://cr.yp.to/antiforgery/**cachetiming**-20050414.pdf*
5. *FIPS PUB 197: the official AES standard - http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf*
6. *Jean-Philippe Aumasson, Simon Fischer, Shahram Khazaei, Willi Meier, and Christian Rechberger: New Features of Latin Dances - http://eprint.iacr.org/2007/472.pdf*
7. *Crowley, Paul (2006). "Truncated differential cryptanalysis of five rounds of Salsa20" - http://www.ciphergoth.org/crypto/salsa20/*
8. *Daniel J. Bernstein, Salsa20 Specification - http://cr.yp.to/snuffle/spec.pdf*
9. *Daniel J. Bernstein, Salsa20 design - http://cr.yp.to/snuffle/812.pdf*