Алгоритм Rijndael

Разработка Advanced Encryption Standard (AES)

- **MARS**
- **RC6**
- **Rijndael**
- **Serpent**
- **Twofish**

Алгоритм Rijndael

Критерий оценки

1. Безопасность
2. Стоимость
3. Характеристики алгоритма и его реализаций

Алгоритм Rijndael

Выполнение шифрования и расшифрования на различных платформах

	32 бита (C)	32 бита (Java)	64 бита (C и ассемблер)	8 бит (C и ассемблер)	32 бита смарт-карты (ARM)	Цифровые сигнальные процессоры
MARS	II	II	II	II	II	II
RC6	I	I	II	II	I	II
Rijndael	II	II	I	I	I	I
Serpent	III	III	III	III	III	III
Twofish	II	III	I	II	III	I

Алгоритм Rijndael

Выполнение управления ключом в зависимости от платформы

	32 бита (C)	32 бита (Java)	64 бита (C и ассемблер)	8 бит (C и ассемблер)	Цифровые сигнальные процессоры
MARS	II	II	III	II	II
RC6	II	II	II	III	II
Rijndael	I	I	I	I	I
Serpent	III	II	II	III	I
Twofish	III	III	III	II	III

Алгоритм Rijndael

Результирующая оценка выполнения

	Шифрование/дешифрование	Установление ключа
MARS	II	II
RC6	I	II
Rijndael	I	I
Serpent	III	II
Twofish	II	III

Алгоритм Rijndael

- Длина блока и длина ключа могут быть независимо установлены в 128, 192 или 256 бит
- Входные байты алгоритма отображаются в байты состояния в следующем порядке:

$A_{0,0}$, $A_{1,0}$, $A_{2,0}$, $A_{3,0}$, $A_{0,1}$, $A_{1,1}$, $A_{2,1}$, $A_{3,1}$, $A_{4,1}$, ….

- Байты ключа шифрования отображаются в массив в следующем порядке:

$K_{0,0}$, $K_{1,0}$, $K_{2,0}$, $K_{3,0}$, $K_{0,1}$, $K_{1,1}$, $K_{2,1}$, $K_{3,1}$, ….

Алгоритм Rijndael

Пример состояния

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

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

Алгоритм Rijndael

Число раундов как функция от длины блока и длины ключа

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

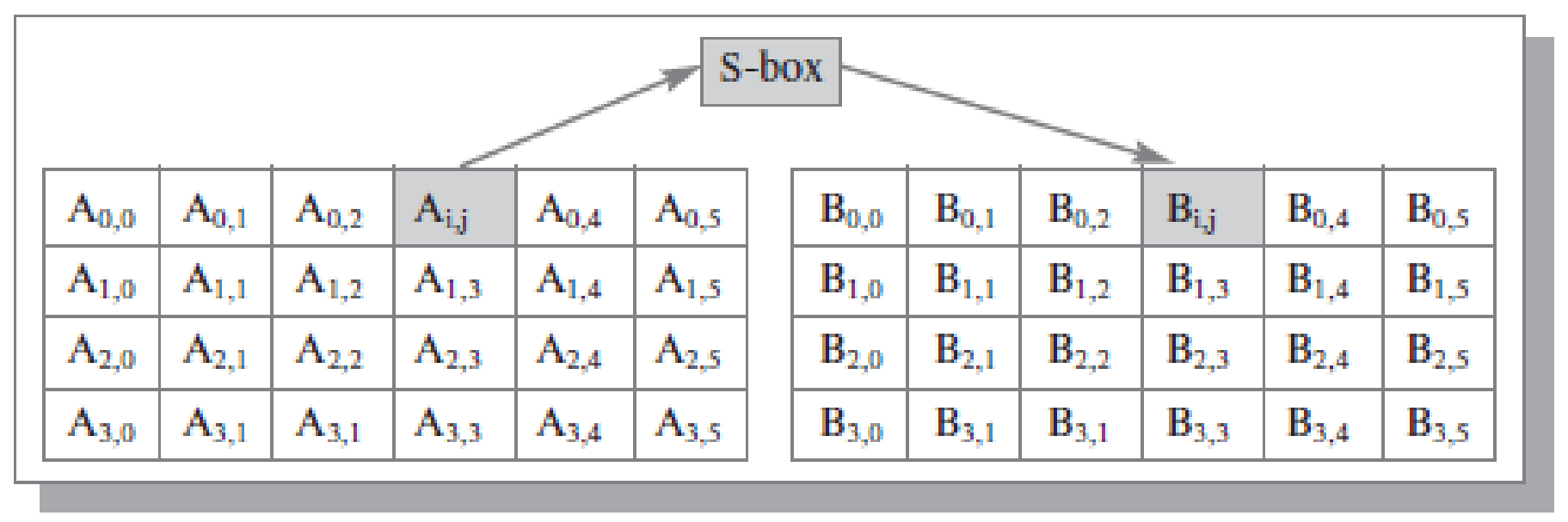
Алгоритм Rijndael

Преобразование раунда

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

Алгоритм Rijndael

Преобразование ByteSub



Алгоритм Rijndael

- *Предварительные математические понятия*

Практически все операции Rijndael определяются на уровне **байта**. Байты можно рассматривать как элементы конечного поля GF (2^8). Некоторые операции определены в терминах четырехбайтных слов. Введем основные математические понятия, необходимые для обсуждения алгоритма.

- **Поле GF (2^8)**

Элементы конечного поля могут быть представлены несколькими различными способами. Для любой степени простого числа существует единственное конечное поле, поэтому все представления GF (2^8) являются изоморфными. Несмотря на подобную эквивалентность, представление влияет на сложность реализации. Выберем классическое полиномиальное представление.

Байт **b**, состоящий из битов $b_7, b_6, b_5, b_4, b_3, b_2, b_1, b_0$, можно записать в виде полинома с коэффициентами из {0, 1}:

$$b_7 x^7 + b_6 x^6 + b_5 x^5 + b_4 x^4 + b_3 x^3 + b_2 x^2 + b_1 x^1 + b_0$$

Алгоритм Rijndael

Пример: байт с шестнадцатеричным значением **‘57’** (двоичное значение **01010111**) соответствует полиному

$$x^6 + x^4 + x^2 + x + 1$$

Определим операцию **сложения.**

Сумма двух элементов, представленных в виде полинома, является полиномом с коэффициентами, равными сумме по модулю 2 (т.е. 1 + 1 = 0) коэффициентов слагаемых.

Пример: **‘57’ + ‘83’ = ‘D4’**

В полиномиальной нотации:

$$(x^6 + x^4 + x^2 + x + 1) + (x^7 + x + 1) = x^7 + x^6 + x^4 + x^2$$

В бинарной нотации:

01010111 + 10000011 = 11010100

Введенное таким образом сложение есть **XOR** битов в байте.

Алгоритм Rijndael

Определим операцию **умножения.**

В полиномиальном представлении умножение в GF (2^8) соответствует умножению полиномов по модулю неприводимого двоичного полинома степени 8. Полином называется неприводимым, если он не имеет делителей, кроме 1 и самого себя. Для Rijndael такой полином является **m(x)** :

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

или **'11B'** в шестнадцатеричном представлении.

Пример: **'57' • '83' = 'C1'**

Сначала перемножим два полинома

$$(x^6 + x^4 + x^2 + x + 1) \bullet (x^7 + x + 1) = x^{13} + x^{11} + x^9 + x^8 + \cancel{x^7} +$$

$$\cancel{x^7} + x^5 + x^3 + \cancel{x^2} + \cancel{x} +$$

$$x^6 + x^4 + \cancel{x^2} + \cancel{x} + 1$$

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

Алгоритм Rijndael

Теперь найдем остаток от деления на полином **m(x)**

$$(x^8 + x^4 + x^3 + x + 1) \bullet (x^5 + x^3) + x^7 + x^6 + 1$$

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

Таким образом, получили

$$x^{13} + x^{11} + x^9 + x^8 + x^6 + x^5 + x^4 + x^8 + 1 \bmod (x^8 + x^4 + x^3 + x + 1)$$

$$= \quad x^7 + x^6 + 1$$

В этом случае результат будет полиномом не выше 7 степени. В отличие от сложения, простой операции умножения на уровне байтов не существует.

Введенная операция умножения является ассоциативной, существует единичный элемент **'01'**. Для любого двоичного полинома **b(x)** не выше 7-й степени можно использовать расширенный алгоритм Евклида для вычисления полиномов **a(x)** и **c(x)** таких, что

$$b(x) \bullet a(x) + m(x) \bullet c(x) = 1$$

Алгоритм Rijndael

Это можно записать как

$$a(x) \bullet b(x) \bmod m(x) = 1 \text{ или}$$

$$b^{-1}(x) = a(x) \bmod m(x)$$

Более того, можно показать, что

$$a(x) \bullet (b(x) + c(x)) = a(x) \bullet b(x) + a(x) \bullet c(x)$$

Из всего этого следует, что множество из 256 возможных значений байта образует конечное поле GF (2^8) с **XOR** в качестве сложения и умножением, определенным выше.

Выполнены все необходимые условия **Абелевой группы**: определена **операция сложения**, обладающая свойством, что каждой паре элементов сопоставляется третий элемент группы, называемый их суммой. Существует нулевой элемент, равный '00', обратный элемент относительно операции сложения. Операция обладает свойствами ассоциативности и коммутативности.

Алгоритм Rijndael

Полиномы с коэффициентами из {0,1}

$a(x) = a_7 x^7 + \ldots + a_2 x^2 + a_1 x + a_0$

$b(x) = b_7 x^7 + \ldots + b_2 x^2 + b_1 x + b_0$

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

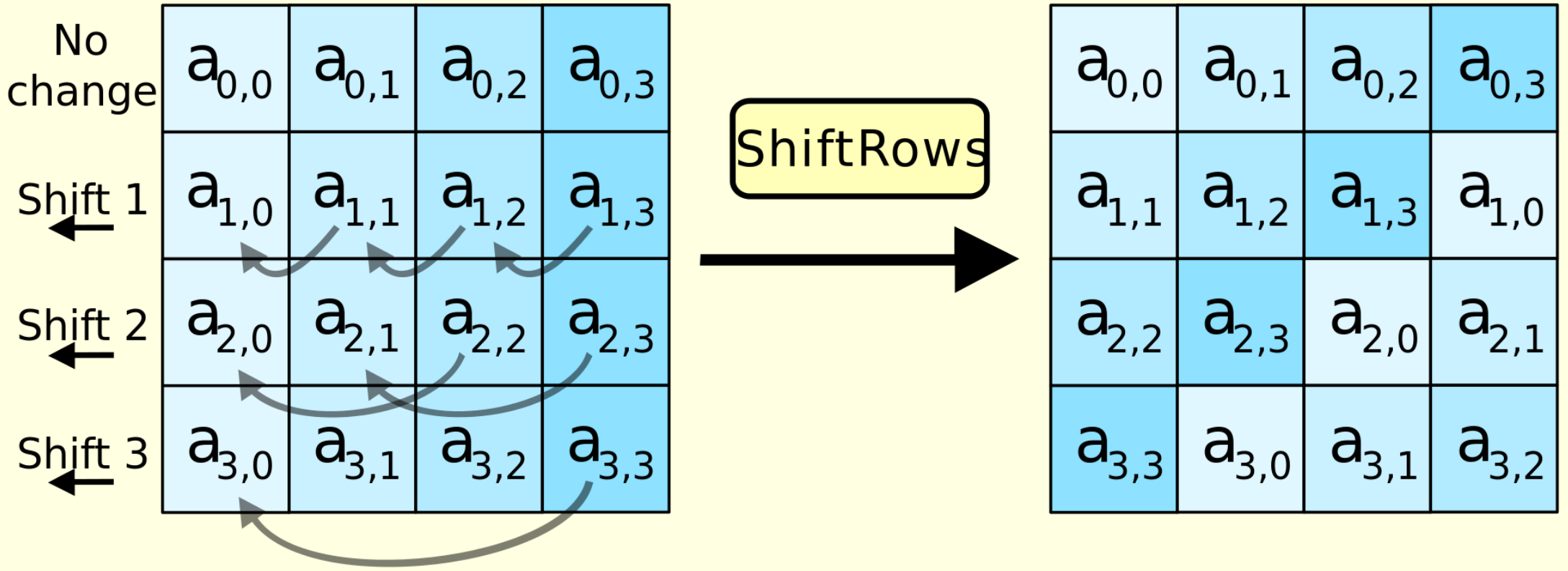
Алгоритм Rijndael

Преобразование ShiftRow

Nb	C_1	C_2	C_3
4	1	2	3
6	1	2	3
8	1	3	4

Алгоритм Rijndael

Преобразование ShiftRow



Алгоритм Rijndael

Преобразование MixColumn

$M(x) = x^4 + 1$

$c(x) = 0x03\ x^3 + 0x01\ x^2 + 0x01\ x + 0x02$

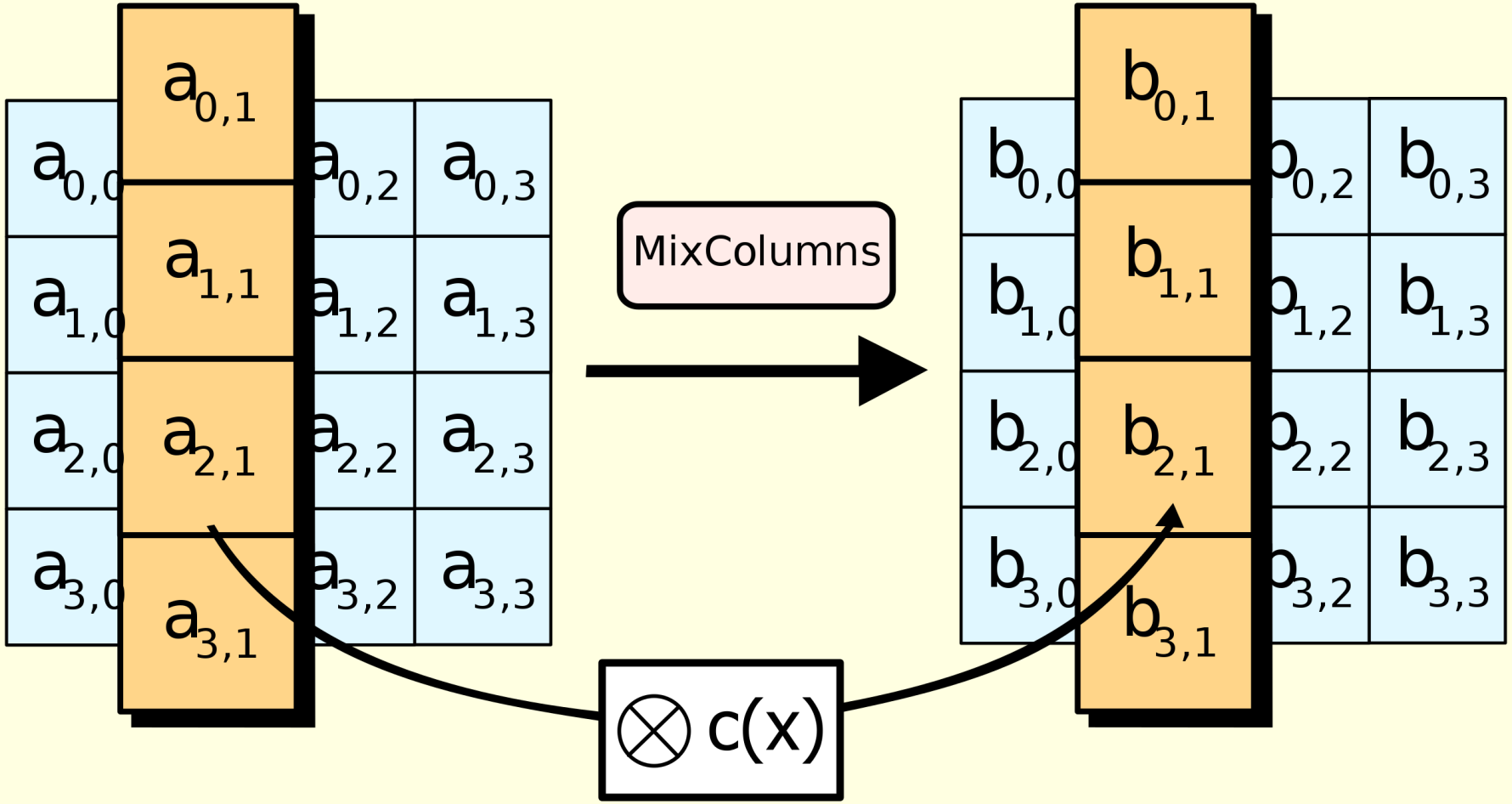
$b(x) = c(x) \otimes a(x)$

Полиномы с коэффициентами из GF (2^8)

Полиномы могут быть определены с коэффициентами из GF (2^8). В этом случае четырехбайтный вектор соответствует полиному степени 4.

Операцию сложения можно ввести как **`XOR`** соответствующих коэффициентов.

Алгоритм Rijndael



Алгоритм Rijndael

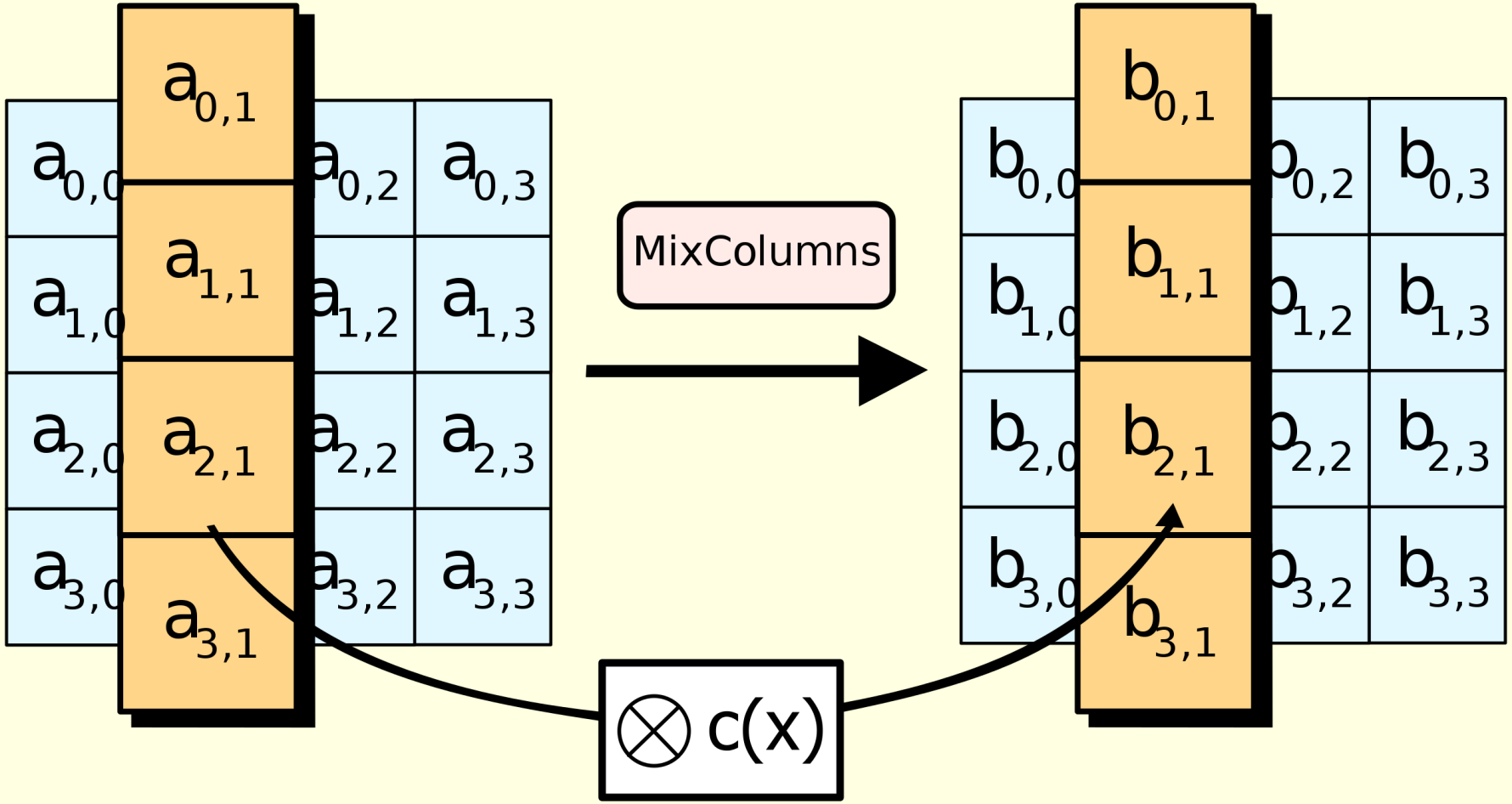
Инверсия **MixColumn** является преобразованием, аналогичным **MixColumn**. Каждый столбец преобразуется умножением его на полином **d(x)**, определяемый следующим образом:

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

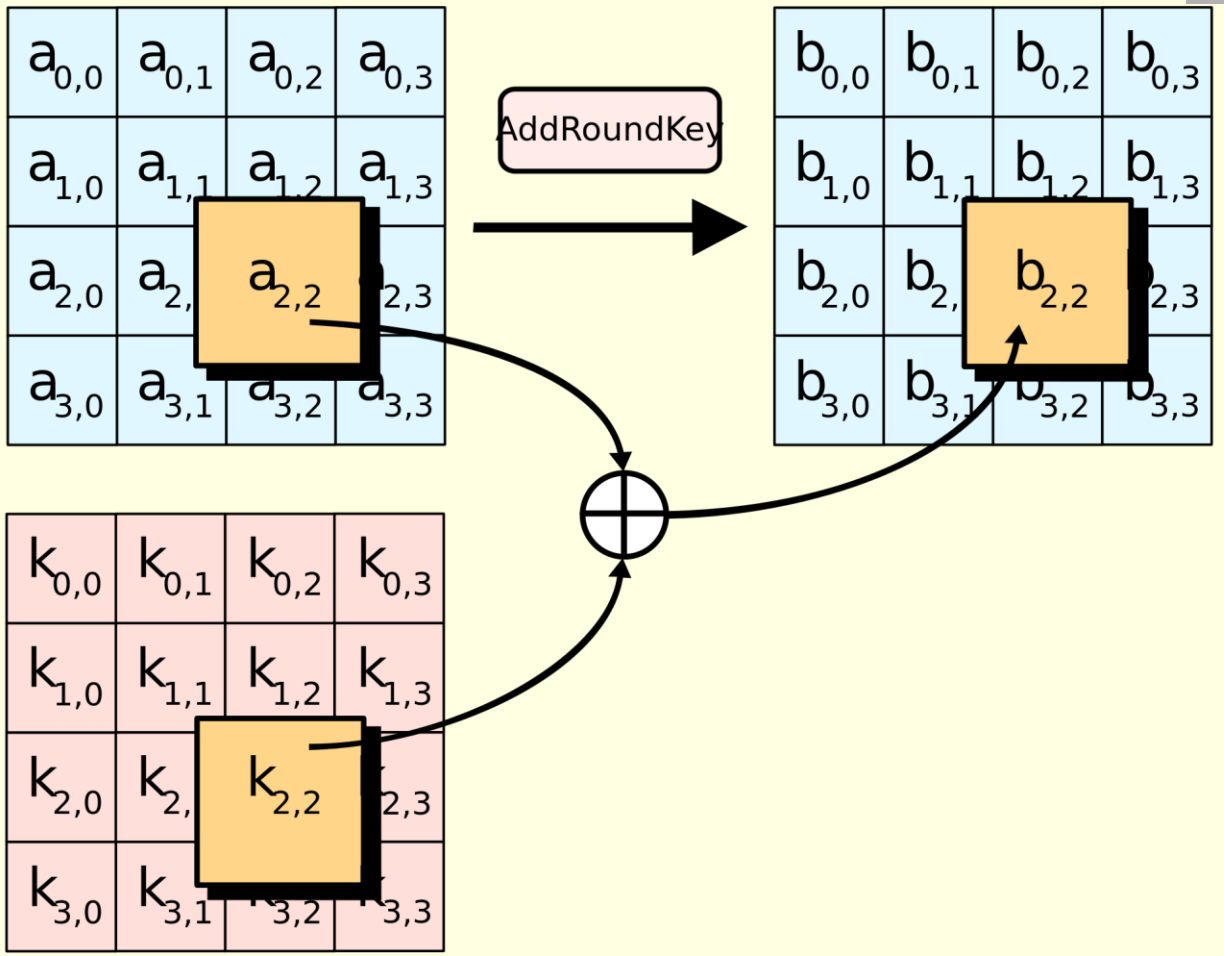
В результате получаем

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

Алгоритм Rijndael



Алгоритм Rijndael



Алгоритм ГОСТ 34.12-2015

В 2015 году был принят новый стандарт на алгоритм симметричного шифрования. Алгоритм называется ГОСТ 34.12-2015 или «Кузнечик».

Основные характеристики

- SP-сеть (как и в AES)
- Длина блока n: 128 бит
- Длина ключа K: 256 бит
- Длина ключей раунда: 128 бит
- Количество раундов: 10

Алгоритм ГОСТ 34.12-2015

Обозначения

- Открытый текст: a
- Шифртекст: C(a)
- Ключи раундов: K_1, …, K_{10}
- Преобразование раунда:

  Подмешивание ключа раунда: **X**

  Нелинейное преобразование: **S**

  Линейное преобразование: **L**
- **XSL**-схема

Алгоритм ГОСТ 34.12-2015

$\mathbb{F}$	Конечное поле $GF(2^8)[x]/p(x)$, где $\boldsymbol{p(x) = x^8 + x^7 + x^6 + x + 1} \in GF(2^8)[x]$; элементы поля $\mathbb{F}$ являются целыми числами, причем элементу из $\mathbb{F}$ соответствует число $z_0 + 2 \cdot z_1 + \ldots + 2^7 \cdot z_7$, где $z_i \in \{0, 1\}$, $i = 0, 1, \ldots, 7$, в классе вычетов по модулю $p(x)$;

Алгоритм ГОСТ 34.12-2015

- **Шифрование:**

  $\mathbf{E_{K1\ \ldots\ K10}\ (a) = X[K_{10}]\ L\ S\ X[K_9]\ \ldots\ L\ S\ X[K_1]\ (a) = C(a)}$

  - 9 раундов полные XSL-схемы, последний раунд состоит только из подмешивания ключа K_1.

- **Расшифрование:**

  $\mathbf{D_{K1\ldots K10}\ (C(a)) = X[K_1]\ S^{-1}\ L^{-1}\ \ldots\ S^{-1}\ L^{-1}\ X[K_{10}]\ (C(a)) = a}$

Алгоритм ГОСТ 34.12-2015

Схема развертывания ключа:

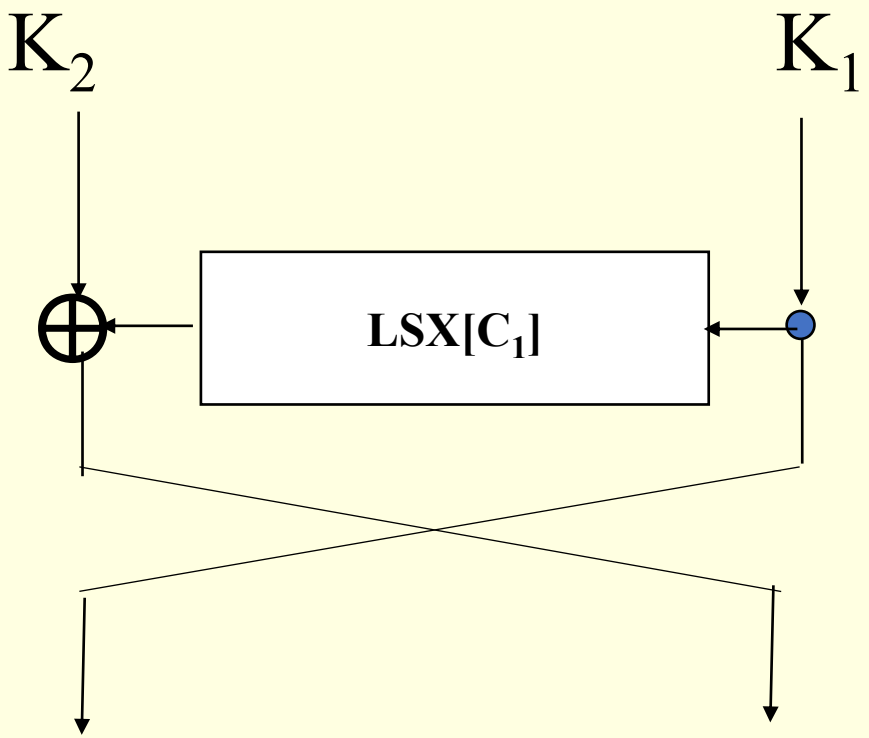
- Сеть Фейстеля, 32 итерации:

  Открытый текст: два исходных подключа

  Шифртекст после 8 раундов: два новых подключа

- Функция усложнения: XSL-схема (совпадает с самим шифром)
- «Ключи» в схеме Фейстеля – это итерационные константы C_i (номер i после преобразования L)

Алгоритм ГОСТ 34.12-2015



В результате ключи раунда имеют более сложные итерационные связи.

Алгоритм ГОСТ 34.12-2015

- **Нелинейное преобразование S**

  Используются 16 S-box **8 x 8**, S-box из ГОСТ-34.11-2012.

- **Преобразование L**

Строится на основе рекурсивного МДР-кода (MDS – maximum distance separable). Обеспечивает максимальный темп перемешивания вектора.

$L(a) = R^{16}(a)$

$R: V_{128} \rightarrow V_{128}$	$\boldsymbol{R(a) = R(a_{15}\|\|\ldots\|\|a_0) = \ell(a_{15}, \ldots, a_0)\|\|a_{15}\|\|\ldots\|\|a_1}$, где $a = a_{15}\|\|\ldots\|\|a_0 \in V_{128}$, $a_i \in V_8$, $i = 0, 1, \ldots, 15$;

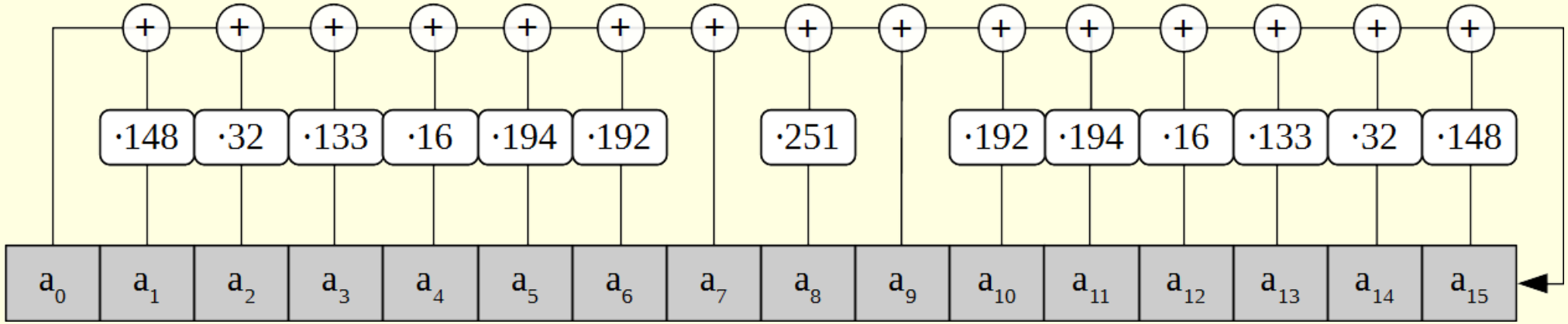
Алгоритм ГОСТ 34.12-2015

- Линейное преобразование задается отображением ℓ: $V_{128} \to V_8$, которое определяется следующим образом:

$$\ell(a_{15}, \ldots, a_0) = \mathbf{148} \cdot a_{15} + \mathbf{32} \cdot a_{14} + \mathbf{133} \cdot a_{13} + \mathbf{16} \cdot a_{12} + \mathbf{194} \cdot a_{11} + \mathbf{192} \cdot a_{10} + \mathbf{1} \cdot a_9 + \mathbf{251} \cdot a_8 + \mathbf{1} \cdot a_7 + \mathbf{192} \cdot a_6 + \mathbf{194} \cdot a_5 + \mathbf{16} \cdot a_4 + \mathbf{133} \cdot a_3 + \mathbf{32} \cdot a_2 + \mathbf{148} \cdot a_1 + \mathbf{1} \cdot a_0$$

- для любых $a_i \in V_8$, $i = 0, 1, \ldots, 15$, где операции *сложения* и *умножения* осуществляются *в поле* $\mathbb{F}$, а константы являются элементами поля.

Алгоритм ГОСТ 34.12-2015



Алгоритм ГОСТ 34.12-2015

- Пример:

$148 \cdot a_{15}$ в поле $\mathbb{F}$ =GF(2)/p(x)

$148 = x^7 + x^4 + x^2 \in \mathbb{F}$

$a_{15} = (c_7, \ldots, c_0)$ – 8 бит

$a_{15} = c_7x^7 + \ldots + c_0$

$148 \cdot a_{15} = (x^7 + x^4 + x^2) \cdot (c_7x^7 + \ldots + c_0) \bmod p(x)$

Алгоритм ГОСТ 34.12-2015

- Сравнение **блочных** алгоритмов симметричного шифрования

Алгоритм Параметры	ГОСТ 28147-89 (сеть Фейстеля)	AES (SP-сеть)	«Кузнечик» (SP-сеть)
Длина блока	64 бита	128 бит	128 бит
Длина ключа	256 бит	128/192/256 бит	256 бит
Число раундов	32	10/12/14	10
Полное перемешивание	За 3 итерации	За 2 итерации	За 1 итерацию

Поточные алгоритмы шифрования

- Поточные шифры – **шифрование каждого бита или байта с использованием гаммирования**. Гаммирование (или Шифр XOR) — метод симметричного шифрования, заключающийся в «наложении» последовательности, состоящей из случайных чисел, на открытый текст.
- Последовательность случайных чисел называется гамма-последовательностью и используется для зашифровывания и расшифровывания данных.
- Суммирование обычно выполняется в каком-либо конечном поле. Например, в поле Галуа суммирование принимает вид операции «исключающее ИЛИ (XOR)».

Алгоритм Salsa20

- AES (FIPS 197) стал золотым стандартом в криптографии. Его эффективность, распространенность и аппаратная поддержка позволяют добиваться высокой производительности во многих областях. На большинстве современных платформ AES от четырех до восьми раз быстрее, чем предыдущий широко используемый шифр – тройной DES.
- Тем не менее существует несколько проблем. Если будущие достижения криптоанализа обнаружат уязвимости в AES, пользователи окажутся в незавидном положении. Единственным другим широко поддерживаемым шифром окажется более медленный 3DES. Поэтому **необходим запасной алгоритм**. Другая проблема состоит в том, что, хотя AES является очень быстрым на специализированной аппаратуре, его **производительность на платформах, в которых отсутствует такая аппаратура, существенно ниже**. Еще одной проблемой является то, что **многие реализации AES уязвимы к тайминг-атакам**, связанным с коллизиями кэше.

Алгоритм Salsa20

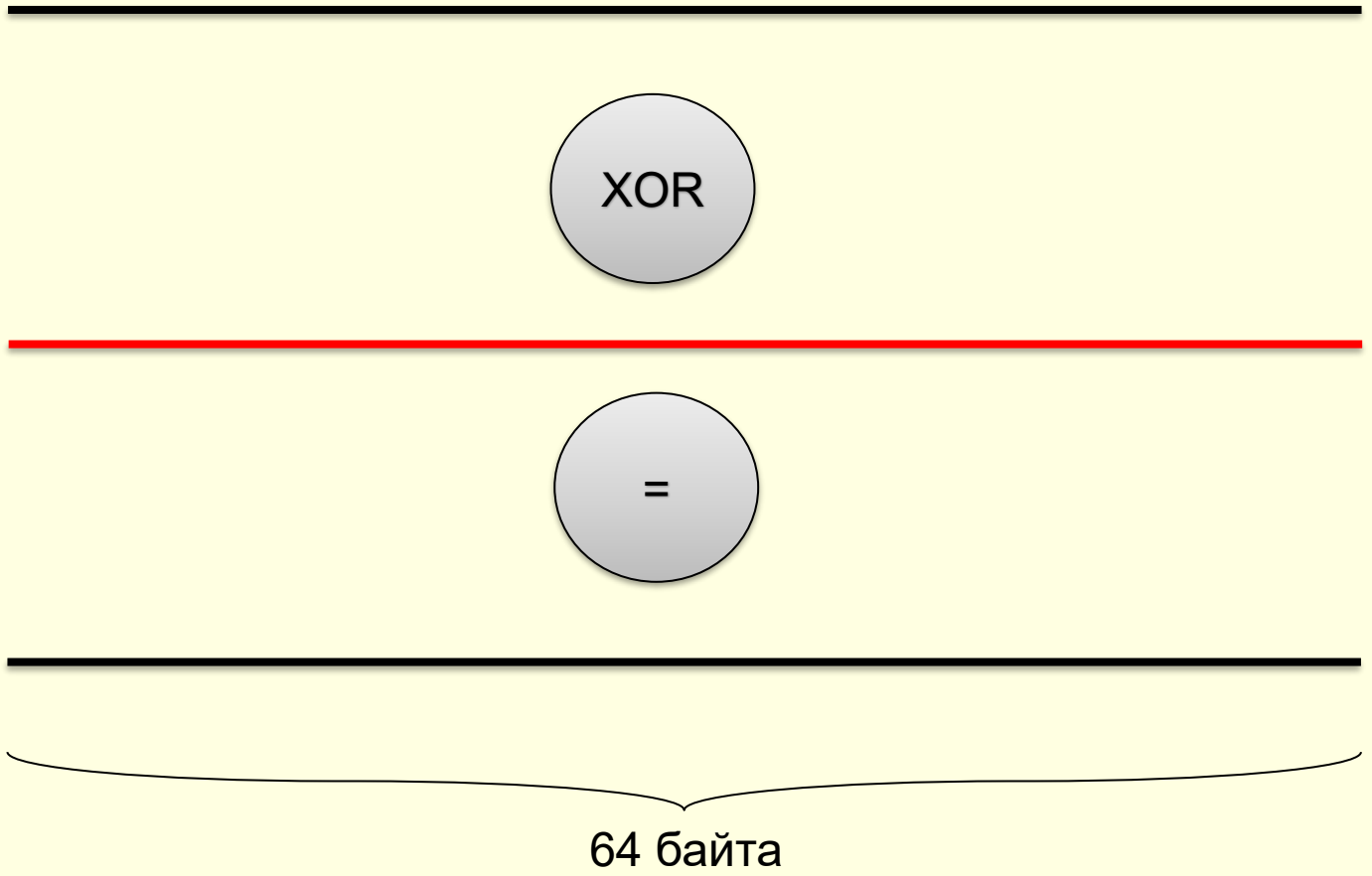
- Алгоритм поточного шифрования **Salsa20** разработан Даниэлем Бернштейном в 2005 г.
- **Вариантом Salsa20 является поточный алгоритм ChaCha** (2008 г.).
- Алгоритм представлен на конкурсе **eStream**, целью которого было создание **европейского стандарта для шифрования данных**. Победитель в первом профиле (поточные шифры для программного применения с большой пропускной способностью).
- Оба шифра основаны на PRF, в основе которых лежит XOR с ротацией и сложением по модулю 2^{32} (add-rotate-XOR – ARX):
  - + mod 2^{32} – **+**
  - XOR – $\oplus$
  - Сдвиги битов
- В основе Salsa20 лежит **хеш-функция с 64-байтным входом и 64-байтным выходом**. Хеш-функция используется в режиме счетчика как потоковый шифр**:**
- **Salsa20 шифрует 64-байтный блок незашифрованного текста, хешируя ключ, nonce и номер блока и выполняя xor результата с незашифрованным текстом**.

Алгоритм Salsa20

P

H (K, nonce, N блока)

C



Алгоритм Salsa20

Опишем Salsa20, начиная с самого низкого уровня, сначала опишем три простых **операции над 4-байтными словами**, далее определим **хеш-функцию Salsa20**, и наконец определим **функцию шифрования Salsa20**.

Структура

Внутреннее состояние – это 16 **32-битных слов**, представленных в виде матрицы 4x4.

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

Алгоритм Salsa20

Слова

- **Словом** называется элемент из {0, 1, …, $2^{32} – 1$}. Слова будем записывать в 16-ричном виде 0x…
- **Сумма** двух слов u, v есть **(u+v) mod 2^{32}**. Сумму будем обозначать просто **u+v**,
- **Исключающее или** двух слов u, v будем обозначать **u⊕v**,
- Для каждого c ∈ {0, 1, 2,…} **c-битной левой ротацией** слова, обозначаемой **u<<<c**, является ненулевое слово, соответствующее **2^cu** по модулю 2^{32} -1, за исключением того, что 0<<<c = 0.

Алгоритм Salsa20

Функция quarterround

y – последовательность из **4-х слов**, quarterround (y) также последовательность из **4-х слов**.

Если y = (y0, y1, y2, y3) , то quarterround (y) = (z0, z1, z2, z3), где

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

$z2 = y2 \oplus ((z1 + y0) <<< 9),$

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

$z0 = y0 \oplus ((z3 + z2) <<< 18),$

Каждое преобразование обратимо, поэтому вся функция обратима.

Функция quarterround изменяет y, оставляя его на том же месте: сначала y_1 изменяется на z_1, затем y_2 изменяется на z_2, затем y_3 изменяется на z_3, затем y_0 на z_0. Каждое изменение обратимо, поэтому вся функция обратима.

Алгоритм Salsa20

Функция **rowround**

y – последовательность из **16 слов**,

rowround (y) также последовательность из **16 слов**.

Пусть $y = (y_0, y_1, \ldots y_{15})$, тогда rowround $(y) = (z_0, z_1, \ldots z_{15})$, где

$(z_0, z_1, z_2, z_3) = \text{quarterround}(y_0, y_1, y_2, y_3)$,

$(z_5, z_6, z_7, z_4) = \text{quarterround}(y_5, y_6, y_7, y_4)$,

$(z_{10}, z_{11}, z_8, z_9) = \text{quarterround}(y_{10}, y_{11}, y_8, y_9)$,

$(z_{15}, z_{12}, z_{13}, z_{14}) = \text{quarterround}(y_{15}, y_{12}, y_{13}, y_{14})$,

Алгоритм Salsa20

Входные значения можно представить в виде матрицы.

y_0 y_1 y_2 y_3

y_4 y_5 y_6 y_7

y_8 y_9 y_{10} y_{11}

y_{12} y_{13} y_{14} y_{15}

Функция **rowround** изменяет строки матрицы одновременно, выполняя перестановку каждой строки с помощью функции **quarterround**.

Алгоритм Salsa20

- В первой строке функция rowround изменяет y_1, затем y_2, затем y_3, затем y_0;
- Во второй строке функция rowround изменяет y_6, затем y_7, затем y_4, затем y_5;
- В третьей строке функция rowround изменяет y_{11}, затем y_8, затем y_9, затем y_{10};
- В четвертой строке функция rowround изменяет y_{12}, затем y_{13}, затем y_{14}, затем y_{15}.

Алгоритм Salsa20

Функция columnround

x является последовательностью из **16 слов**,

columnround (x) также является последовательностью из **16 слов**.

Пусть x = (x_0, x_1, …., x_{15}), тогда columnround (x) = (y_0, y_1, …, y_{15}), где

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

(y5, 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).

Алгоритм Salsa20

Наглядно входные данные можно представить в виде матрицы.

x0	x1	x2	x3
x4	x5	x6	x7
x8	x9	x10	x11
x12	x13	x14	x15

Функция columnround просто является перестановкой функции quarterround: она изменяет столбцы матрицы, параллельно переставляя каждый столбец, пропуская его через функцию quarterround. В первом столбце функция columnround изменяет y_4, затем y_8, затем y_{12}, затем y_0; во втором столбце функция columnround изменяет y_9, затем y_{13}, затем y_1, затем y_5; в третьем столбце функция columnround изменяет y_{14}, затем y_2, затем y_7, затем y_{11}, затем y_{15}.

Алгоритм Salsa20

Функция doubleround

Пусть x является последовательностью из **16 слов**,

тогда doubleround (x) является последовательностью из **16 слов**.

doubleround (x) = rowround (columnround (x))

doubleround изменяет столбцы входного значения, а затем изменяет строки. Каждое слово изменяется дважды.

Алгоритм Salsa20

Функция littleendian

Пусть b является последовательностью из **4 байтов**,

тогда littleendian (b) есть **слово**.

Пусть $\mathbf{b = (b_0, b_1, b_2, b_3)}$, тогда

$$\mathbf{littleendian\ (b) = b_0 + 2^8 b_1 + 2^{16} b_2 + 2^{24} b_3}$$

Заметим, что функция littleendian является обратимой.

Хеш-функция Salsa20

Пусть x есть последовательность из **64 байтов**, тогда результат выполнения Salsa20 (x) также есть последовательность из **64 байтов**.

$$\mathbf{Salsa20\ (x) = x + doubleround^{10}\ (x)},$$

Где каждая 4-х байтовая последовательность рассматривается как слово в little-endian форме.

Алгоритм Salsa20

Начальное состояние Salsa20			
"expa"	Key	Key	Key
Key	"nd 3"	Nonce	Nonce
Pos	Pos	"2-by"	Key
Key	Key	Key	"te k"

Алгоритм Salsa20

Функция expansion Salsa20

Пусть **k** является 32-байтной или 16-байтной последовательностью,

n является 16-байтной последовательностью,

тогда **Salsa20 (n)** есть 64-байтная последовательность.

Определим

$\sigma_0 = (101, 120, 112, 97)$, $\sigma_1 = (110, 100, 32, 51)$, $\sigma_2 = (50, 45, 98, 121)$, и $\sigma_3 = (116, 101, 32, 107)$.

Алгоритм Salsa20

Пусть $\mathbf{k_0}$, $\mathbf{k_1}$ **и** $\mathbf{n}$ являются 16-байтными последовательностями.

$$\mathbf{Salsa20_{k0,\,k1}\,(n) = Salsa20\,(\sigma_0, k_0, \sigma_1, n, \sigma_2, k_1, \sigma_3).}$$

Определим

$\tau_0 = (101, 120, 112, 97)$, $\tau_1 = (110, 100, 32, 49)$, $\tau_2 = (54, 45, 98, 121)$,

и $\tau_3 = (116, 101, 32, 107)$. Пусть **k, n** являются 16-байтными последовательностями

$$\mathbf{Salsa20_k\,(n) = Salsa20\,(\tau_0, k, \tau_1, n, \tau_2, k, \tau_3).}$$

Алгоритм Salsa20

«Расширение» означает расширение (k, n), используя $Salsa20_k$ (n). Это также означает расширение k на большой поток выходных значений $Salsa20_k$ для различных n.

Константы σ_0 σ_1 σ_2 σ_3 и τ_0 τ_1 τ_2 τ_3 являются «расширением 32-байтного k» и «расширением 16-байтного k» в ASCII.

Алгоритм Salsa20

Функция шифрования Salsa20

Входные и выходные значения

Пусть ***k*** есть 32-байтная *или* 16-байтная последовательность. Пусть ***v*** есть 8-байтная последовательность. Пусть ***m*** есть *l*-байтная последовательность для некоторого $l \in \{0, 1, \ldots, 2^{70}\}$. **Шифрование Salsa20 *m* с nonce *v* ключом *k***, обозначаемое как $\mathbf{Salsa20}_k(v) \oplus m$, является *l*-байтной последовательностью.

k является **секретным ключом** (предпочтительно 32 байта);

v является **nonce**, т.е. уникальным числом сообщения;

m является исходным сообщением (**plaintext**);

$\mathbf{Salsa20}_k(v) \oplus m$ является зашифрованным сообщением (**ciphertext**).

Или ***m*** может быть зашифрованным сообщением (ciphertext), в этом случае

$\mathbf{Salsa20}_k(v) \oplus m$ является исходным незашифрованным сообщением.

Алгоритм ChaCha20

Рассмотрим поточный шифр **ChaCha20**, а также его использование в аутентификаторе **Poly1305**, причем рассмотрим их и как отдельные алгоритмы, и в «режиме комбинации», т.е. алгоритм аутентифицированного шифрования со связанными данными (authenticated encryption with associated data – AEAD).

Алгоритмы описаны в RFC 7539 и 8439.

Рассмотрим определение и рекомендации по реализации для следующих трех алгоритмов:

1. Шифр ChaCha20. Он существенно быстрее, чем AES при только программных реализациях, что делает его в три раза быстрее на платформах, в которых отсутствует аппаратура с поддержкой AES. Также ChaCha20 не чувствителен к тайминг-атакам.
2. Аутентификатор Poly1305. Это очень быстрый код аутентификации сообщений. Алгоритм имеет простую реализацию.
3. CHACHA20-POLY1305 – аутентифицированное шифрование со связанными данными (AEAD).

Алгоритм ChaCha20

Разработкой стандартов для данных алгоритмов занимается Crypto Forum Research

Group (CFRG) (https://datatracker.ietf.org/rg/cfrg/documents/).

Используемые соглашения

Описание алгоритма ChaCha в разное время называет состояние ChaCha то «вектор», то «матрица». Термин «матрица» более наглядный, по этой же причине некоторые раунды называются «раунды столбцов», а другие «диагональные раунды». Приведем диаграмму как матрицы соотносятся с векторами.

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

Элементы данного вектора или матрицы являются 32-битными unsigned целыми.

Алгоритм называется ChaCha. **ChaCha20** – это конкретный вариант алгоритма, который выполняется **20 раундов** (или 80 четверть-раундов). Определены другие варианты с 8 и 12 раундами, но здесь мы будем рассматривать только 20-раундовый ChaCha, поэтому названия «ChaCha» и «ChaCha20» будем считать синонимами.

Алгоритм ChaCha20

Рассмотрим алгоритмы и конструирование AEAD.

Четверть раунда ChaCha

Базовой операцией алгоритма ChaCha является четверть раунда. Алгоритм оперирует с четырьмя 32-битными unsigned-целыми, обозначаемыми как **a**, **b**, **c** и **d**. Операции следующие (в C-нотации):

a += b,	**d ^= a,**	**d <<<= 16,**
c += b,	**b ^= c,**	**b <<<=12,**
a += b,	**d ^= a,**	**d <<<=8,**
c +=d,	**b ^=c,**	**b <<<=7,**

Где «+» обозначает целочисленное **сложение по модулю 2^{32}**, «^» **обозначает побитовое исключающее или (XOR)**, «<<< n» обозначает **n-битный циклический сдвиг влево** (к старшим битам).

Например,

```
a = 0x11111111
b = 0x01020304
c = 0x77777777
d = 0x01234567
c = c + d = 0x77777777 + 0x01234567 = 0x789abcde
b = b ^ c = 0x01020304 ^ 0x789abcde = 0x7998bfda
b = b <<< 7 = 0x7998bfda <<< 7 = 0xcc5fed3c
```

Алгоритм ChaCha20

Четверть раунда для состояния ChaCha

Состояние ChaCha 16 целых чисел. Операция для четверти раунда выполняется только над 4 из них, отсюда и название. Каждая четверть раунда выполняется над четырьмя числами из состояния. Обозначим **QUARTERROUND (x, y, z, w)** операцию четверти раунда для чисел, обозначенных **x, y, z** и **w** для состояния ChaCha, если рассматривать состояние как вектор. Например, если мы применим QUARTERROUND (1, 5, 9, 13) к состоянию, это означает выполнение операции четверть раунда над элементами, помеченными «*», в то время как остальные останутся без изменения.

0	*a	2	3
4	*b	6	7
8	*c	10	11
12	*d	14	15

Заметим, что данное выполнение четверти раунда является частью того, что называется «раунд столбца».

Алгоритм ChaCha20

Функция блока ChaCha20

Функция блока ChaCha преобразует состояние ChaCha, выполняя несколько четверть раундов.

Входными данными в ChaCha являются:

256-битный ключ, который трактуется как конкатенация восьми 32-битных little-endian целых.
96-битный nonce, который трактуется как конкатенация трех 32-битных little-endian целых.
32-битный параметр счетчика блока, который трактуется как 32-битное little-endian целое.

Выходным значением является 64 байтное значение, которое выглядит как случайное.

Алгоритм ChaCha20

Описываемый алгоритм ChaCha использует **256-битный ключ**. В исходном алгоритме кроме того могут использоваться 128-битные ключи и 8- и 12-раундовые варианты, но это вне предметной области нашего рассмотрения. Рассмотрим функцию блока ChaCha.

Алгоритм ChaCha получает на вход **64-битный nonce** и **64-битный счетчик блока**. Рассмотрим модифицированный алгоритм, чтобы обеспечить согласованность с рекомендациями по использованию аутентифицированного шифрования (RFC 5116). При данных ограничениях используется единственная комбинация (ключ, nonce) для 2^{32} блоков, или 256 GB, что является достаточным для большинства использований. В тех случаях, когда единственный ключ используется несколькими отправителями, важно гарантировать, что они не используют одни и те же nonce. Это можно гарантировать разбиением пространства nonce таким образом, чтобы первые 32 бита были уникальными для каждого отправителя, в то время как другие 64 бита брались из счетчика.

Алгоритм ChaCha20

Состояние ChaCha инициализируется следующим образом:

Первые четыре слова (0-3) являются константами: 0x61707865, 0x3320646e, 0x79622d32, 0x6b206574.

Следующие восемь слов (4-11) берутся из 256-битного ключа, байты берутся в little-endian порядке, 4-байтными кусками (chunk).

12-е слово является счетчиком блока. Так как каждый блок состоит из 64 байтов, 32-битное слово достаточно для 256 гигабайтов данных.

Слова с 13 по 15 являются nonce, который НЕ ДОЛЖЕН повторяться для одного и того же ключа. 13-е слово является первыми 32 битами входного nonce, который рассматривается как little-endian целое, а 15-е слово является последними 32 битами.

```
cccccccc cccccccc cccccccc cccccccc

kkkkkkkk kkkkkkkk kkkkkkkk kkkkkkkk

kkkkkkkk kkkkkkkk kkkkkkkk kkkkkkkk

bbbbbbbb nnnnnnnn nnnnnnnn nnnnnnnn
```

c – константа, k – ключ, b – счетчик блока, n – nonce.

Алгоритм ChaCha20

ChaCha выполняется 20 раундов, переключаясь между «раундами столбца» и «диагональными раундами». Каждый раунд состоит из четырёх четверть раундов, которые выполняются следующим образом. Четверть раунда **1 – 4** являются частью «**раунда столбца**», а **5 – 8** являются частью **«диагонального раунда»**:

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

Алгоритм ChaCha20

Псевдокод функции блока ChaCha20

```
inner_block (state):
   Qround(state, 0, 4, 8, 12)
   Qround(state, 1, 5, 9, 13)
   Qround(state, 2, 6, 10, 14)
   Qround(state, 3, 7, 11, 15)
   Qround(state, 0, 5, 10, 15)
   Qround(state, 1, 6, 11, 12)
   Qround(state, 2, 7, 8, 13)
   Qround(state, 3, 4, 9, 14)
   end
```

Алгоритм ChaCha20

Входными значениями ChaCha20 являются:

- **256-битный ключ**
- **32-битный начальный счетчик**. Он может быть установлен в любое значение, но обычно это 0 или 1. Имеет смысл использовать единицу, если нулевой блок используется как то еще, например, для создания одноразового ключа аутентификатора как часть AEAD-алгоритма.
- **96-битный nonce**. В некоторых протоколах это называется инициализационным вектором.
- **Plaintext произвольной длины**.

Выходным значением является зашифрованное сообщение, или ciphertext, той же самой длины.

Расшифрование выполняется тем же самым способом. Функция блока ChaCha20 используется для расширения ключа в поток ключа, который XORed с ciphertext, получая plaintext.