Алгоритм AES

Усовершенствованный стандарт шифрования (ADVANCED ENCRYPTION STANDARD) — стандарт на блочный шифр с симметричными ключами, изданный Национальным Институтом Стандартов и Технологии (NIST) в декабре 2001 года.

История

В 1997 году NIST начал искать замену для DES, которая была названа *Усовершенствованным стандартом шифрования (ADVANCED ENCRYPTION STANDARD или AES).* В NIST-спецификациях были заложены требования размера блока из 128 битов и трех различных размеров ключей: 128, 192 и 256 битов. Спецификации также требовали, чтобы AES был открытым алгоритмом, публично доступным во всем мире.

Спецификации стандартов были объявлены во многих странах, чтобы запросить ответы у специалистов и заинтересованных лиц со всех континентов.

После *Первой Конференции по выбору Кандидатов AES* NIST объявила, что 15 из 21 полученных алгоритмов отвечают поставленным требованиям и выбраны как первые кандидаты (август 1998 г.). Алгоритмы были представлены от многих стран; разнообразие этих предложений демонстрировало открытость процесса и участие всего мира.

После Второй Конференции по выбору Кандидата AES, которая была проведена в Риме, NIST объявила 5 из 15 из кандидатов. Алгоритмы MARS, RC6, Rijndael, Serpent и Twofish были выбраны как финалисты (август 1999).

После *Третей Конференции по выбору Кандидата AES* NIST объявила, что выбран алгоритм *Усовершенствованного Стандарта Шифрования — Rijndael,* спроектированный бельгийскими исследователями Джоном Даеменом и Винсентом Риджменом (октябрь 2000 г.).

В феврале 2001 г. NIST объявил, что эскиз **Федерального Стандарта обработки Информации (FIPS)** доступен для общественного рассмотрения и комментариев.

Наконец, AES был издан как FIPS 197 в *Федеральном Регистре* в декабре 2001 г.

Критерии

Критерии, определенные NIST для выбора AES, относятся к трем областям: безопасность, стоимость и реализация. В конце концов *Rijndael* был оценен в совокупности как лучший, отвечающий этим критериям.

Безопасность

Особое внимание уделялось безопасности. Поскольку NIST ясно потребовал 128-битовый ключ, этот критерий определял то, что внимание обращалось на устойчивость шифра к другим атакам криптоанализа, нежели атака грубой силы.

Стоимость

Вторым критерием была стоимость, которая задает требуемую вычислительную эффективность и требования для различных реализаций, таких, как аппаратные средства, программное обеспечение или интеллектуальные карты доступа.

Реализация

Этот критерий включал требование, что алгоритм должен иметь гибкость (возможность быть реализованным на любой платформе) и простоту.

Раунды

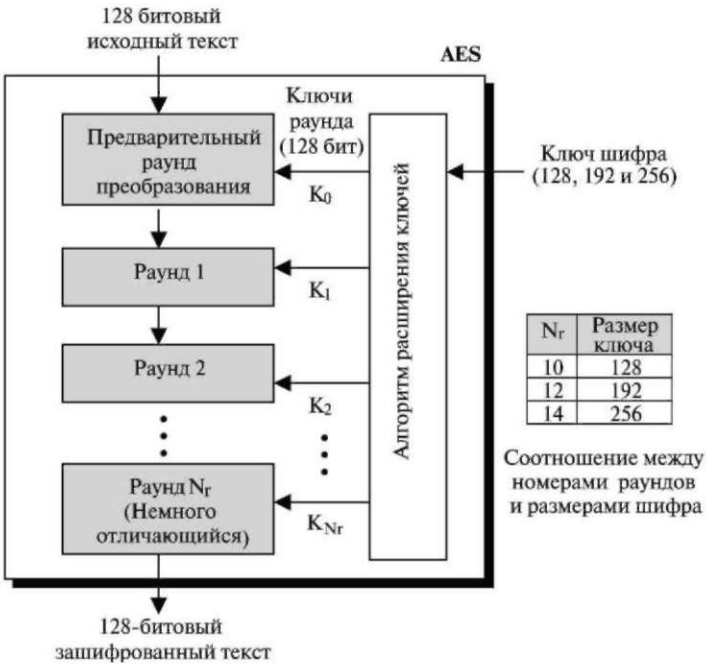
AES — шифр не-Файстеля, который зашифровывает и расшифровывает блок данных 128 битов, используя 10, 12 или 14 раундов. Размер ключа может быть 128, 192 или 256 битов и зависит от числа раундов. Рисунок 7.1 показывает общую схему: алгоритм шифрования (называемого шифром); алгоритм дешифрования (называемый обратным шифром), для которого применяются те же ключи, но в обратном порядке.

На рис. 7.1 N_r определяет число раундов. Рисунок также показывает отношение между числом раундов и размером ключа — это означает, что мы имеем три различных версии AES; они обозначаются как AES-128, AES-192 и AES-256. Однако ключи раунда, которые созданы алгоритмом расширения ключей всегда 128 бит, имеют тот же самый размер, что и блоки зашифрованного или исходного текста.

AES определил три версии, с 10, 12 и 14 раундами. Каждая версия использует различный размер ключа шифра (128, 192 или 256), но ключ раунда — всегда 128 бит.

Число ключей раунда, сгенерированных алгоритмом расширения ключей, всегда на один больше, чем число раундов. Другими словами, мы имеем

$$\text{число ключей раунда} = N_r + 1$$



Nr	Размер ключа
10	128
12	192
14	256

Соотношение между номерами раундов и размерами шифра

Рис. Общее построение шифрования AES-шифра

Мы обозначаем ключи раунда как K_0, K_1, K_2..., K_N.

Единицы данных

AES использует пять единиц для представления данных: биты, байты, слова, блоки и массивы состояний. Бит — наименьшая и элементарная единица; другие единицы могут быть выражены в терминах меньших единиц. Рисунок 7.2 показывает единицы неэлементарных данных: байт, слово, блок, массив состояний (state).

Бит

В AES **бит** — двоичная цифра со значением 0 или 1. Мы используем строчные буквы для обозначения бит.

Байт

Байт — группа из восьми битов, которая может быть обработана как единый объект: матрица из одной строки (1 x 8) восьми битов или столбец матрицы (8x1) из восьми битов. Когда информация байта обрабатываются как матрица строки, то биты вставляются в матрице слева направо. Когда байт обрабатывается как матрица столбца, биты вставляются в матрице сверху вниз. Мы будем использовать строчную «жирную» букву (Bold) для обозначения байта.

Слово

Слово — группа из 32 битов, которая может быть обработана как единый объект. Это матрица из строки в четыре байта или столбец матрицы из четырех байтов. Когда слово обрабатывается как матрица-строка, байты вставляются слева направо. Когда слово представляется матрицей-колонкой, байты вставляются сверху вниз. Мы будем использовать строчную «жирную» букву **W** для обозначения слова.

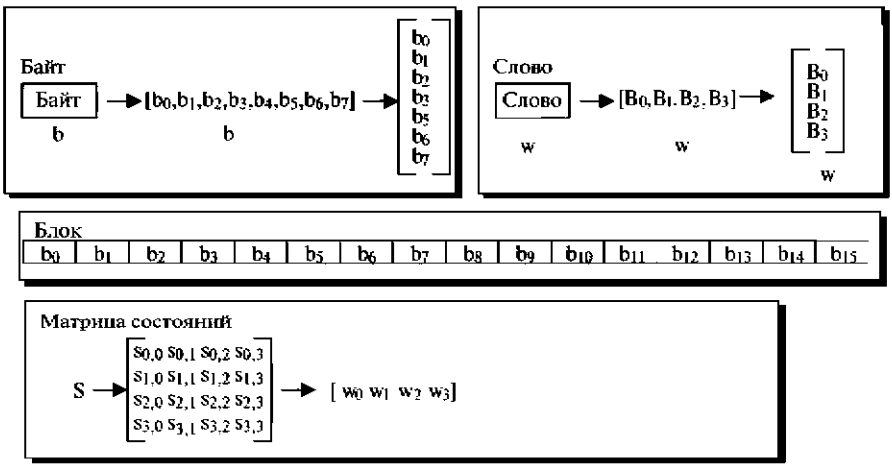


Рис. Единицы данных, используемых в AES

Блок

AES зашифровывает и расшифровывает блоки данных. **Блок** в AES — группа 128 битов. Однако блок может быть представлен как матрица-строка из 16-ти байтов.

Матрица состояний

AES использует несколько раундов, каждый раунд состоит из несколько каскадов. Блок данных преобразовывается от одного каскада к другому. В начале и в конце шифра AES применяется термин *блок данных;* до и после каждого каскада блок данных называется **матрицей состояний.** Мы используем «жирную» заглавную букву, чтобы обозначить эту матрицу. Хотя матрица состояний на различных каскадах обычно обозначается **S**, мы иногда применяем букву **T**, чтобы обозначить временную матрицу состояний. Матрицы состояний, подобно блокам, состоят из 16 байтов, но обычно обрабатываются как матрицы 4 × 4 байтов. В этом случае каждый элемент матрицы состояний обозначается как $S_{r,c}$, где r (от 0 до 3) определяет строку и c (от 0 до 3) определяет столбец. Иногда матрица состояний обрабатывается как матрица-строка слов (1 × 4). Это имеет смысл, если мы представляем слово как матрицу-столбец. В начале шифра байты в блоке данных вставляются в матрицу состояний столбец за столбцом, в каждом столбце — свер-

ху вниз. В конце шифра байты в матрице состояний извлекаются, как это показано на рис. 7.3.

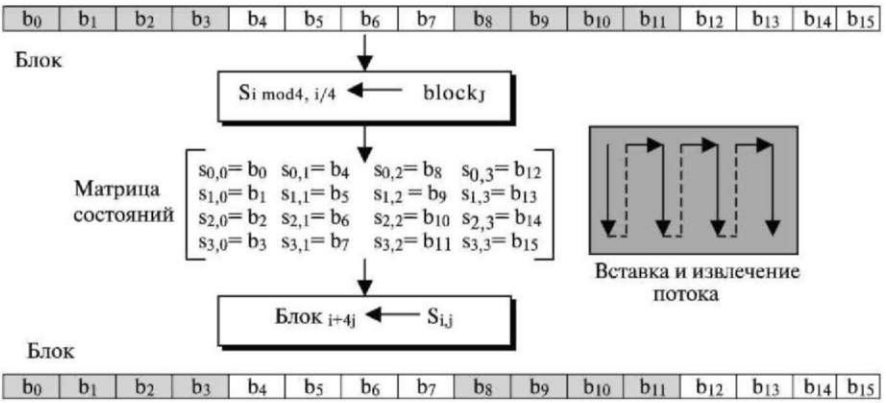


Рис. Преобразование блок — матрица состояний и матрица состояний — в блок

Пример

Рассмотрим, как можно изобразить блок с 16 символами в виде матрицы 4 × 4. Предположим, что текстовый блок — «AES uses a matrix». Добавим два фиктивных символа в конце и получим «AESUSESAMATRIXZZ». Теперь мы заменим каждый символ целым числом между 00 и 25. Представим каждый байт как целое число с двумя шестнадцатеричными цифрами. Например, символ «S» сначала поменяем на 18, а затем запишем в шестнадцатеричном изображении как 12. Матрица состояний тогда заполняется столбец за столбцом, как это показано на рис. 7.4.

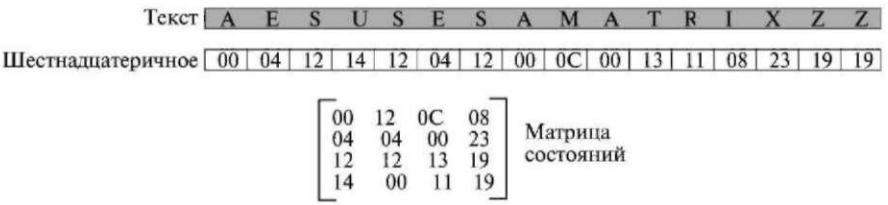


Рис. Переход шифрованного текста в матрицу состояний

Структура каждого раунда

Рисунок 7.5 показывает структуру каждого раунда на стороне шифрования. Каждый раунд, кроме последнего, использует четыре преобразования, которые являются обратимыми. Последний раунд имеет только три преобразования.

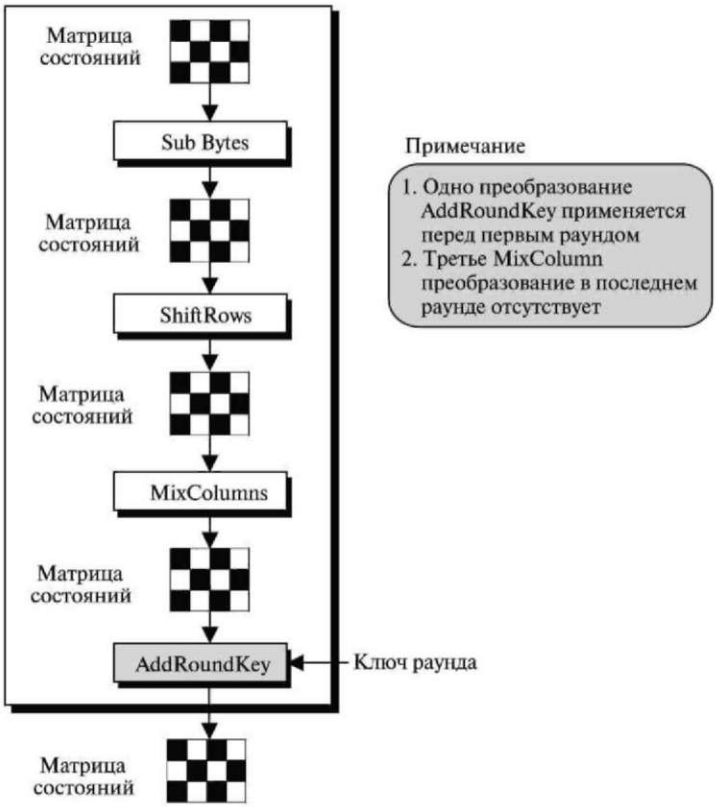


Рис. Структура каждого раунда на стороне шифрования

. Преобразования

Чтобы обеспечить безопасность, AES использует четыре типа преобразований: подстановка, перестановка, смешивание и добавление ключа. Ниже мы обсудим каждое.

Подстановка

AES подобно DES применяет подстановку. Однако этот механизм имеет различия. Первое: подстановка делается для каждого байта. Второе: для преобразова-

ния каждого байта используется только одна таблица — это означает, что если два байта одинаковы, то и результат преобразования одинаков. Третье: преобразование определяется или процессом поиска в таблице, или математическим вычислением в $GF(2^8)$ поле. AES работает с двумя обратимыми преобразованиями.

SubBytes

Первое преобразование, **SubBytes**, применяется на стороне шифрования. Чтобы применить подстановку к байту, мы интерпретируем байт как две шестнадцатеричные цифры. Левая цифра определяет строку, а правая — колонку в таблице перестановки. На пересечении строки и колонки, обозначенных этими шестнадцатеричными цифрами, находится новый байт. Рисунок 7.6 иллюстрирует идею.

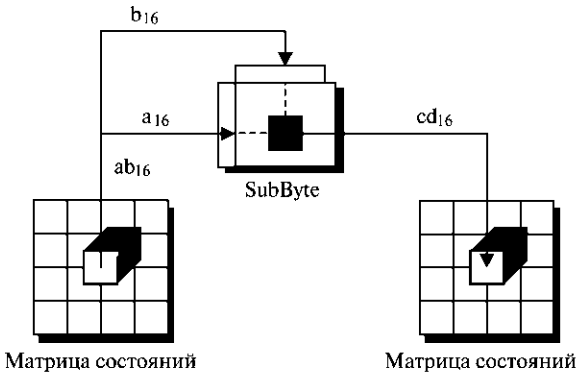


Рис. Преобразование SubByte

В преобразовании SubBytes состояние обрабатывается как матрица байтов 4 × 4. В один момент проводится преобразование одного байта. Содержание каждого байта изменяется, но расположение байтов в матрице остается тем же самым. В процессе преобразования каждый байт преобразуется независимо от других — это шестнадцать личных преобразований байт в байт.

Операция SubByte включает 16 независимых преобразований байта в байт.

Таблица 7.1 показывает таблицу подстановки (S-блок) для преобразования SubBytes. Преобразование обеспечивает эффект перемешивания. Например, два байта, $5A_{16}$ и $5B_{16}$, которые отличаются только одним битом (самый правый бит), преобразованы в BE_{16} и 39_{16}, которые отличаются четырьмя битами.

InvSubBytes

InvSubBytes — инверсия **SubBytes**.

Таблица Таблица преобразования SubBytes

	0	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*A*	*B*	*C*	*D*	*E*	*F*
0	63	7C	77	7B	F2	6B	6F	C5	30	01	67	2B	FE	D7	AB	76
1	CA	82	C9	7D	FA	59	47	FO	AD	D4	A2	AF	9C	A4	72	CO
2	B7	FD	93	26	36	3F	F7	CC	34	A5	E5	F1	71	D8	31	15
3	04	C7	23	C3	1 8	96	05	9A	07	12	80	E2	EB	27	B2	75
4	09	83	2C	1A	1B	6E	5A	AO	52	3B	D6	B3	29	E3	2F	84
5	53	D1	00	ED	20	FC	B1	5B	6A	CB	BE	39	4A	4C	58	CF
6	DO	EF	AA	FB	43	4D	33	85	45	F9	02	7F	50	3C	9F	A8
7	51	A3	40	8F	92	9D	38	F5	BC	B6	DA	21	10	FF	F3	D2
8	CD	0C	13	EC	5F	97	44	17	C4	A7	7E	3D	64	5D	19	73
9	60	81	4F	DC	22	2A	90	88	46	EE	B8	14	DE	5E	0B	DE
A	E0	32	3A	0A	49	06	24	5C	C2	D3	AC	62	91	95	E4	79
B	E7	CB	37	6D	8D	D5	4E	A9	6C	56	F4	EA	65	7A	AE	08
C	BA	78	25	2E	1C	A6	B4	C6	E8	DD	74	IF	4B	BD	8B	8A
D	70	3E	B5	66	48	03	F6	OE	61	35	57	B9	86	C1	1D	9E:
E	El	F8	98	11	69	D9	8E	94	9B	1E	87	E9	CE	55	28	DF
F	8C	Al	89	OD	BF	E6	42	68	41	99	2D	OF	BO	54	BB	16

Таблица Таблица преобразования InvSubBytes

	0	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*A*	*B*	*C*	*D*	*E*	*F*
0	52	099	6A	D5	30	36	A5	38	BF	40	A3	9E	81	F3	D7	FB
1	7C	E3	39	82	9B	2F	FF	87	34	8E	43	44	C4	DE	E9	CB
2	54	7B	94	32	A6	C2	23	3D	EE	4C	95	0B	42	FA	C3	4E
3	08	2E	A1	66	28	D9	24	B2	76	5B	A2	49	6D	8B	D1	25
4	72	F8	F6	64	86	68	98	16	D4	A4	5C	CC	5D	65	B6	92
5	6C	70	48	50	FD	ED	B9	DA	5E	15	46	57	A7	8D	9D	84
6	90	D8	AB	00C	8C	BC	D3	0A	F7	E4	58	05	B8	B3	45	06
7	D0	2C	1E	8F	CA	3F	0F	02	C1	AF	BD	03	01	13	8A	6B
8	3A	91	11	41	4F	67	DC	EA	97	F2	CF	CE	F0	B4	E6	73
9	96	AC	74	22	E7	AD	35	85	E2	F9	37	E8	1C	75	DE	6E
A	47	E1	1A	71	1D	29	C5	89	6F	B7	62	0E	AA	18	BE	1B
B	FC	56	3E	4B	C6	D2	79	20	9A	DB	C0	FE	78	CD	5A	F4
C	1F	DD	A8	33	88	07	C7	31	B1	12	10	59	27	80	EC	5F
D	60	51	7E	A9	19	B5	4A	0D	2D	E5	7A	9F	93	C9	9C	EF:
E	A0	E0	3B	4D	AE	2A	F5	B0	CB	EB	BB	3C	83	53	99	61
F	17	2B	04	7E	BA	77	D6	26	E1	69	14	63	55	21	0C	7D

Пример

Рисунок 7.7 показывает, как матрица состояний преобразуется с использованием SubBytes. Рисунок также показывает, что InvSubBytes однозначно воспроиз-

водит оригинал. Заметим, что если два байта имеют одинаковое значение, то они преобразуются одинаково. Например, два байта 04_{16} и 04_{16} в левой матрице состояний преобразуются в $F2_{16}$ и $F2_{16}$ в правой матрице состояний и наоборот. Причина в том, что каждый байт использует одну и ту же таблицу преобразований.

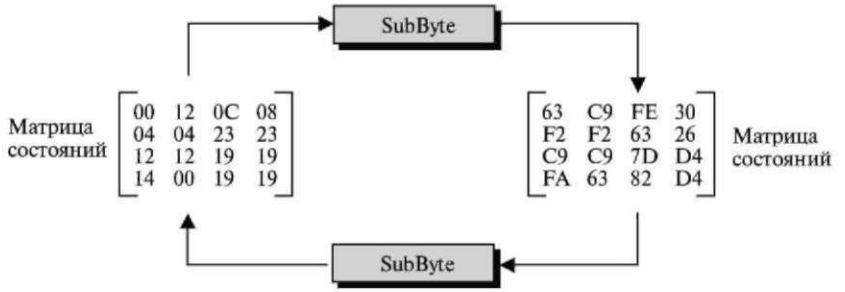


Рис. Преобразование SubByte

Преобразование с использованием поля GF(2^8)

Хотя мы можем использовать таблицы 7.1 и.7.2 для перестановки каждого байта, AES дает определение алгебраическим преобразованиям на основе поля GF(2^8) с помощью неприводимых полиномов ($x^8 + x^4 + x^3 + x + 1$), как это показано на рис. 7.8.

Преобразование SubByte повторяет процесс, называемый *subbyte*, шестнадцать раз. Inv SubByte повторяет процесс, называемый *invsubbyte*. Каждый шаг преобразования обрабатывает один байт.

В процедуре *subbyte* байт мультипликативная инверсия байта (двоичной строки на 8 битов) находится в GF(2^8) с помощью неприводимого полинома по модулю ($x^8 + x^4 + x^3 + x + 1$). Обратите внимание, что байт — 00_{16} сам является собственной инверсией. Инвертированный байт затем интерпретируется как матрица-столбец с самым младшим битом наверху и самым старшим битом внизу. Эта матрица-столбец умножается на постоянную квадратную матрицу, **X**, и результат, который является матрицей-столбцом, складывается с постоянной матрицей столбца **y**, что дает новый байт. Обратите внимание, что умножение и сложение битов происходит в GF(2). *invsubbyte* делает те же действия в обратном порядке.

После нахождения байта инверсного сомножителя процесс похож на аффинное шифрование, которое мы обсуждали в лекции 3. При шифровании умножение является первой операцией, сложение — второй. При дешифровании вычитание (сложение инверсией) является первым, а деление (умножение с инверсией) — вторым. Мы можем легко доказать, что эти два преобразования инверсны друг другу, потому что сложение или вычитание в GF(2) — фактически операция ИСКЛЮЧАЮЩЕЕ ИЛИ.

subbyte: $\quad d = \mathbf{X}\,(s_{r,c})^{-1} \oplus \mathbf{y}$

invsubbyte: $\quad [\mathbf{X}^{-1}\,(d \oplus y)^{-1} = [\mathbf{X}^{-1}\,(\mathbf{X}\,((s_{r,c})^{-1}\,\mathbf{y} \oplus \mathbf{y}]^{-1} = [(s_{r,c})^{-1}]^{-1} = s_{r,c}$

Преобразования SubBytes и InvSubBytes инверсны друг другу.

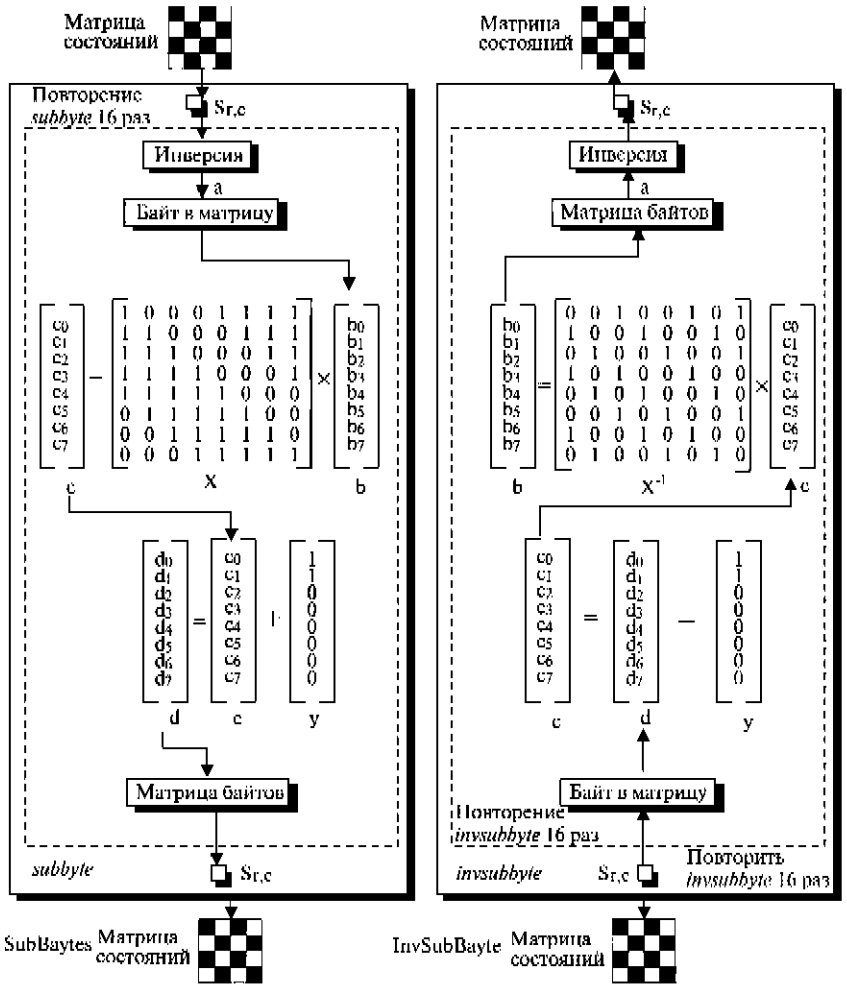


Рис. Процессы *SubBytes и InvBytes*

Пример

Покажем, как байт 0C преобразуется в FE с помощью процедуры *subbyte* и преобразуется обратно в 0C с помощью процесса *invsubbyte.*

1. *subbyte*
   a. Инверсный сомножитель в поле GF(2^8) есть B0 или в двоичном отображении **b** (1011 0000).
   b. Умножая на матрицу **X** эту матрицу, имеем в результате c = (10011101).
   c. Результат применения операции XOR (ИСКЛЮЧАЮЩЕЕ ИЛИ) – **d** = (11111110) FE в шестнадцатеричном представлении.

2. *invsubbyte*
   a. Результат применения операции XOR (ИСКЛЮЧАЮЩЕЕ ИЛИ) с (10011101)
   b. Результат умножения на матрицу X^{-1} (11010000) или B0.
   c. Инверсия по умножению B0 это 0C.

Алгоритм

Хотя на рисунке мы показали матрицы, чтобы подчеркнуть характер подстановки (аффинное преобразование), алгоритм не обязательно использует умножение и сложение матриц, потому что большинство элементов в постоянной квадратной матрице — только 0 или 1. Значение постоянной матрицы столбца — 0×63. Мы можем написать простой алгоритм, чтобы выполнить SubByte. Алгоритм 7.1 вызывает процедуру *subbyte* 16 раз один раз для каждого байта в матрице состояний.

Процедура ByteToMatrix преобразовывает байт к матрицу-столбец 8×1. Процедура MatrixToByte преобразовывает матрицу-столбец 8×1 к байту. Расширение этого алгоритма для InvSubBytes оставляем как упражнение.

Нелинейность

Хотя умножение и сложение матриц в процедуре *subbyte* преобразование аффинного типа и линейно, замена байта его мультипликативной инверсией в $GF(2^8)$ — нелинейная. Этот шаг делает все преобразование нелинейным.

Перестановка

Другое преобразование производит сдвиг в раунде. Этот сдвиг переставляет байты. В отличие от DES, в котором делается поразрядная перестановка, преобразование сдвига делается на уровне байта; порядок битов в байте не меняется.

Алгоритм Программа на псевдокоде для преобразования *SubBytes*

```
SubBytes (S)
{
   for (r = 0 to 3)
      (c = 0 to 3)
         Sr,c=subbyte(Sr,c)
}

subbyte (byte)
{
   a ← byte-1          //Multiplicative inverse in GF(28 ) with inverse
   ByteToMatrix (a,b)  //of 00 to be 00
   For (i= 0 to7)
   {
      ci ← bi ⊕ b(i+4)mod8 ⊕ b(i+5)mod8 ⊕ b(i+6)mod8 ⊕ b(i+7)mod8
      di ← ci ⊕ ByteToMatrix (0 × 63)
   }
   MatrixToByte (d,d)
   byte ← d
}
```

ShiftRows

При шифровании применяется преобразование, называемое **ShiftRows**, со смещением влево. Число сдвигов зависит от номера строки (0, 1, 2 или 3) матрицы состояний. Это означает, что строка 0 не сдвигается и последняя строка сдвигается на три байта. Рисунок 7.9 показывает преобразование смещения.

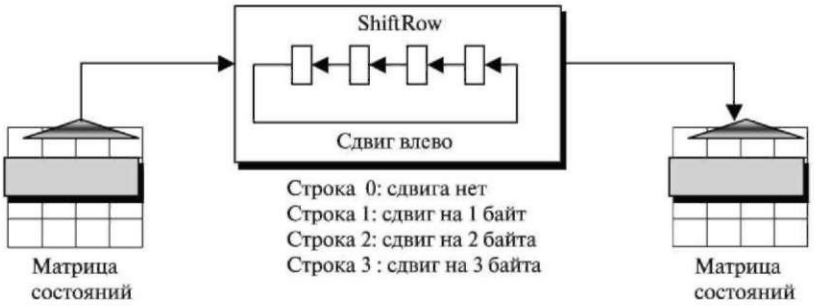


Рис. Преобразование *ShiftRows*

Обратите внимание, что преобразование ShiftRows работает одновременно только с одной строкой.

InvShiftRows

При дешифровании применяется преобразование, называемое **InvShiftRows**, со смещением вправо. Число сдвигов равно номеру строки (0, 1, 2 и 3) в матрицы состояний.

ShiftRows- и InvShiftRows-преобразования инверсны друг другу.

Алгоритм

Программа на псевдокоде для преобразования ShiftRows

```
{
   for (r = 1 to 3)
      shiftrow (Sr, r)                    // Sr r-тая строка}

shiftrow (row, n)   // n - число байтов, на которое должен быть сделан сдвиг

{
   CopyRow (row, t) for (c = 0 to 3)   //t - временная строка
   for (c = 0 to 3)
      row (c-n) mod4 ← tc
}
```

кой, мы используем процедуру, называемую *shiftrow*, которая сдвигает байт в единственной строке. Мы вызываем эту процедуру три раза. Процедура *shiftrow* сначала копирует строку во временную матрицу строки (матрица t), а потом сдвигает строку.

Пример

Рисунок 7.10 показывает, как, используя преобразование ShiftRows, преобразуется матрица состояний. Рисунок также иллюстрирует, как преобразование InvShiftRows создает первоначальную матрицу состояний.

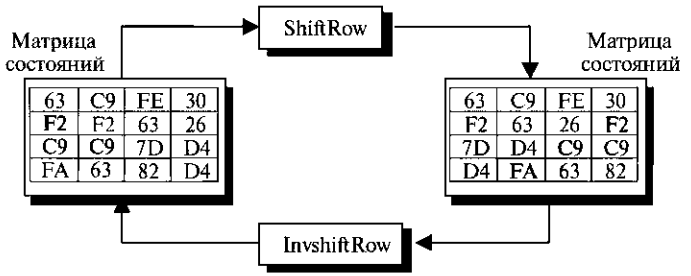


Рис. Пример преобразования *ShiftRow* в примере 7.4

Смешивание

Подстановка, которая делается преобразованием SubByte, изменяет значение байта, основанное только на первоначальном значении и входе в таблице; процесс не включает соседние байты. Мы можем сказать, что SubByte — *внутрибайтовое* преобразование. Перестановка, которая делается ShiftRows-преобразованием, обменивает местами байты, не переставляя биты в байтах. Мы можем сказать, что ShiftRows — преобразование *обмена байтами*. Теперь нам нужно внутрибайтовое преобразование, изменяющее биты в байте и основанное на битах в соседних байтах. Мы должны смешать байты, чтобы обеспечить рассеивание на разрядном уровне.

Преобразование смешивания изменяет содержание каждого байта, преобразовывая четыре байта одновременно и объединяя их, чтобы получить четыре новых байта. Чтобы гарантировать, что каждый новый байт будет отличаться от другого (даже если все четыре байта те же самые), процесс сначала умножает каждый байт на различный набор констант и затем смешивает их. Смешивание может быть обеспечено матричным умножением. Как мы обсуждали в лекции 2, когда мы умножаем квадратную матрицу на матрицу-столбец, результат — новая матрица-столбец. После того как матрица умножена на значения строки в матрице констант, каждый элемент в новой матрице зависит от всех четырех элементов старой матрицы. Рисунок 7.11 иллюстрирует эту идею.

AES определяет преобразование, называемое MixColumns. Для применения такого преобразования вводится также обратное преобразование, называемое InvMixColumns. Рисунок 7.12 показывает матрицу констант, используемую для

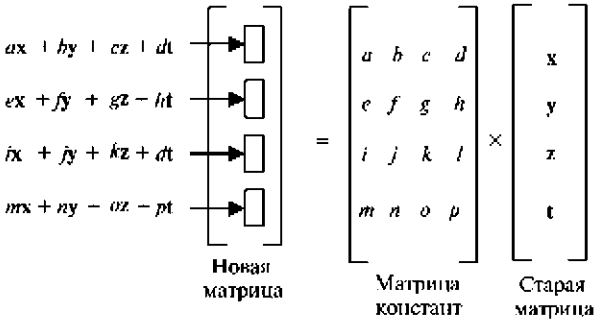


Рис. Смешивание байтов с использованием смешивающей матрицы

этих преобразований. Эти две матрицы инверсны друг другу, когда элементы интерпретируются как слова из 8 ми битов (или полиномы) с коэффициентами в GF (2^8). Доказательство мы оставляем как упражнение.

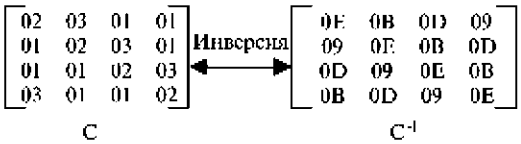


Рис. Матрица констант, используемая MixColumns и InvMixColumns

MixColumns

Преобразование **MixColumns** работает на уровне столбца; оно преобразовывает каждый столбец матрицы состояний в новый столбец. Это преобразование — фактически матричное умножение столбца матрицы состояний и квадратной матрицы констант. Байты в столбце матрицы состояний и в матрице констант интерпретируются как слова по 8 битов (или полиномы) с коэффициентами в GF (2). Умножение байтов выполняется в GF(2^8) по модулю (10001101) или ($x^8 + x^4 + x^3 + x + 1$). Сложение — это применение операции ИСКЛЮЧАЮЩЕЕ ИЛИ (XOR) к словам по 8 бит. Рисунок 7.13 показывает преобразование MixColumns.

InvMixColumns

Преобразование **InvMixColumns** похоже на MixColumns-преобразование. Если две матрицы констант инверсны друг другу, то легко доказать, что эти два преобразования также инверсны друг другу.

Преобразования MixColumns и InvMixColumns инверсны друг другу.

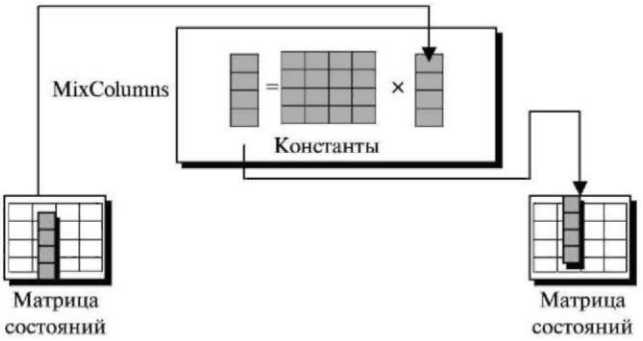


Рис. Преобразование MixColumns

Алгоритм Программа на псевдокоде для преобразования MixColumns

```
MixColumns (S)
{
   for (c = 0 to 3)
      mixcolumn (sc)
}

mixcolumn (col)
{
   CopyColumn (col, t)        //t is a temporary column

   col0 ← (0x02)• t0 ⊕ (0x03)• t1 ⊕ t2 ⊕ t3
   col1 ← t0 ⊕ (0x02)• t1 ⊕ (0x03)• t2 ⊕ t3
   col2 ← t0 ⊕ t1 ⊕ (0 × 02)• t2 ⊕ (0x03)• t3
   col3 ← (0x03)• t0) ⊕ t1 ⊕ t2 ⊕ (0x02) • t3
}
```

Алгоритмы MixColumns и InvMixColumns включают умножение и сложение в поле GF(2^8). Как мы видели в лекции 4, есть простой и эффективный алгоритм для умножения и сложения в этом поле. Однако чтобы показать характер алгоритма (преобразование одного столбца одновременно), мы используем процедуру, называемую *mixcolumn*. Она может быть вызвана алгоритмом четыре раза. Процедура *mixcolumn* просто умножает строки матрицы констант на столбец в матрицы состояний. В вышеупомянутом алгоритме оператор (•), используемый в процедуре *mixcolumn*, — умножение в поле GF(2^8). Оно может быть заменено простой процедурой, как это уже рассматривалось в лекции 4. Программу для InvMixColumns оставляем как упражнение.