Advanced Encryption Standard AES

Krypto II

VL25

Datum: 10.01.2019

Buch des Tages

Titel: A Course in Number Theory and Cryptography (2nd Edition)

Autor(en): Neil Koblitz

Verlag: Springer, 1994

Umfang: ca. 230 Seiten

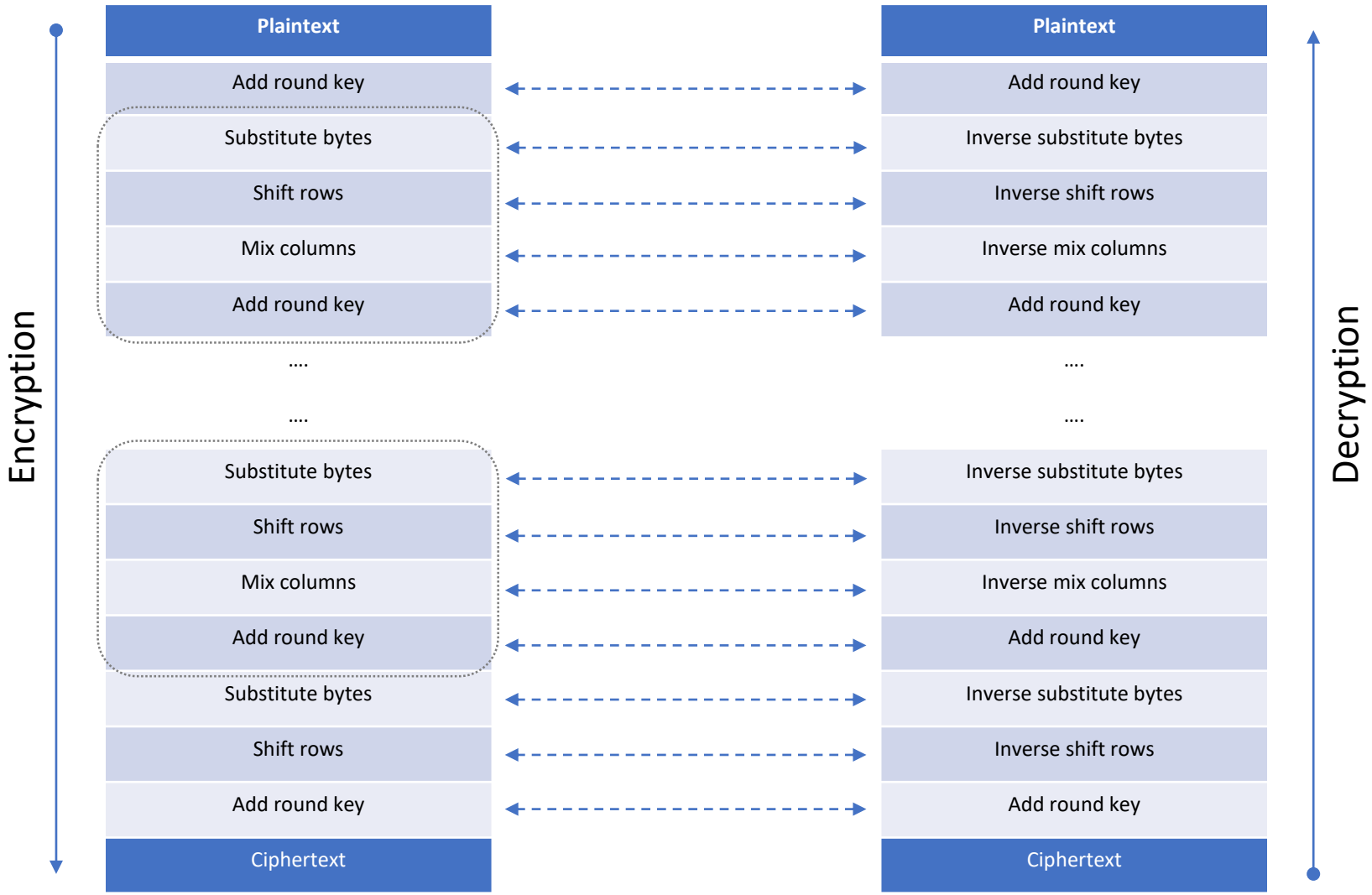
Hinweise zum Inhalt:

Dieses Buch ist ein Klassiker und wird an zahlreichen Hochschulen noch immer als Lehrbuch eingesetzt. Die Darstellung ist mathematisch, aber vergleichsweise gut verständlich und auch für Einsteiger empfehlenswert, insbesondere zum besseren Verständnis der Grundlagen für Public Key Verfahren.

Symmetrische Verfahren kommen im Buch nicht vor.

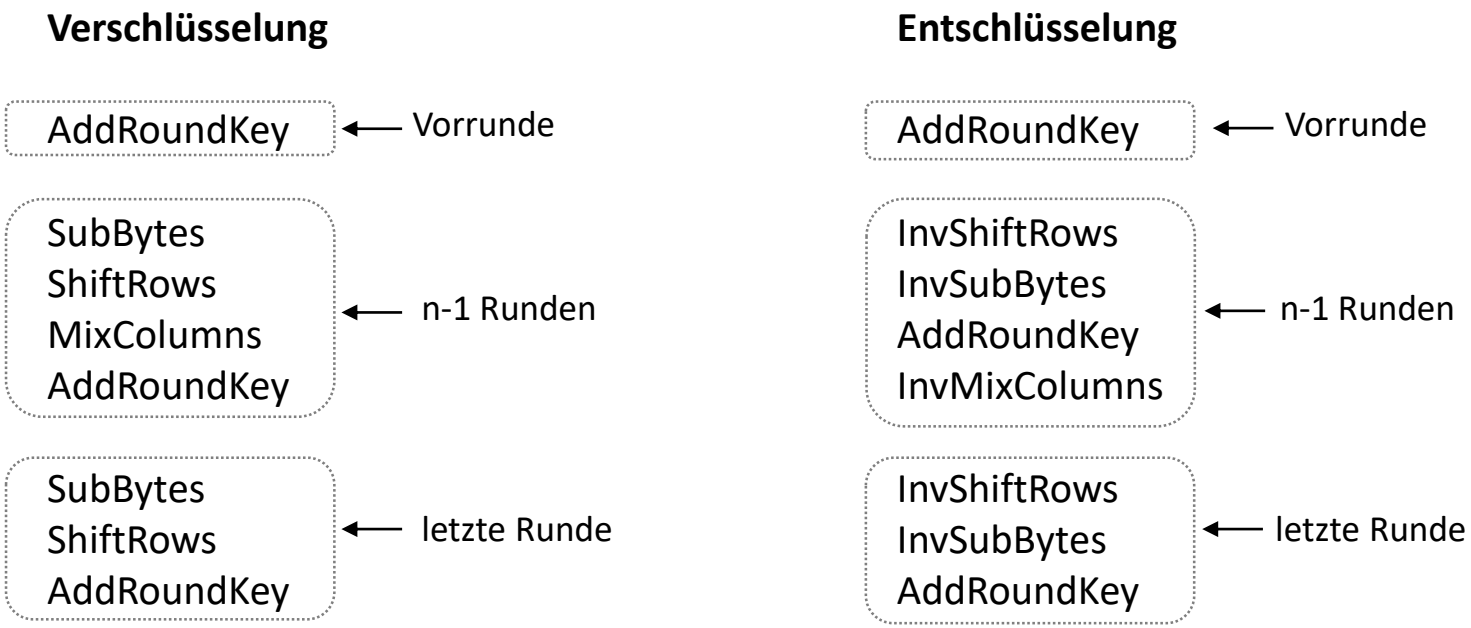
Mathematische Grundlagen ändern sich nur sehr behutsam im Laufe der Zeit, insofern ist das Erscheinungsdatum 1994 kein Hinderungsgrund, um dieses Lehrbuch als Ergänzung zu einer allgemeineren Einführung zu empfehlen.

Entschlüsselung und Verschlüsselung



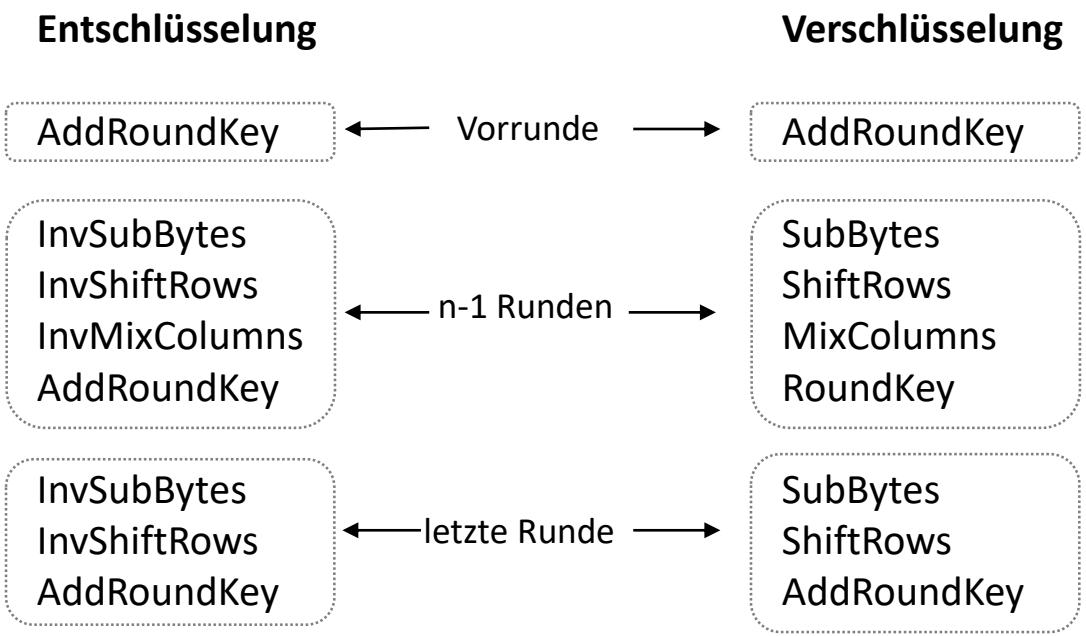
Entschlüsselung (1)

In der letzten Graphik haben wir die Entschlüsselung von unten nach oben ausgeführt, zur besseren Gegenüberstellung mit der Verschlüsselung. Stellen wir die derart dargestellte Entschlüsselung vom Kopf auf die Beine, erhalten wir zunächst folgende Darstellung:



Entschlüsselung (2)

Wie man zeigen kann, lassen sich die Operationen InvShiftRows und InvSubBytes anpassen und in ihrer Reihenfolge vertauschen. Gleiches gilt für die Operationen AddRoundKey und InvMixColumns, sofern wir u.a. die Konstruktion der Rundenschlüssel etwas anpassen. Hierdurch erhalten wir:



Wie wir sehen, haben Entschlüsselung und Verschlüsselung nun eine identische Rundenstruktur. Nur die einzelnen Operationen weichen noch teilweise voneinander ab.

State- und Square Array

- Nicht nur Plaintext und Ciphertext bestehen aus 128-Bit Vektoren. Vor und nach jedem Zwischenschritt entstehen ebenfalls 128-Bit Vektoren. Im Design von AES wird ein solcher Vektor jeweils als ***State*** bezeichnet.
- AES teilt den 128-Bit Input zur Ausführung der Verschlüsselungsoperationen auf in 16 Bytes der Länge 8. Die meisten internen Operationen erfolgen dann auf solchen 8-Bit Vektoren. Diese werden angeordnet als 4x4 Matrix.
- Die 16 entstehenden Bytes bezeichnen wir mit B_1, …, B_{16}. Jedes Byte B besteht aus 8 Bits (b_7, …, b_0).
- Die Anordnung der 16 Bytes als sogenanntes ***Square Array*** (oder manchmal auch ***State Array***) wird in AES wie folgt vorgenommen:

B_0	B_4	B_8	B_{12}
B_1	B_5	B_9	B_{13}
B_2	B_6	B_{10}	B_{14}
B_3	B_7	B_{11}	B_{15}

Substitute Bytes

- Die ***Forward substitute bytes transformation***, kurz ***Substitute Bytes*** oder auch ***SubBytes***, nimmt einen einfachen Table lookup vor. Substitute Bytes ist also eine Tabelle mit 256 Einträgen. Eine solche Substitutionstabelle nennen wir auch ***S-Box***.
- Beim AES ist diese S-Box als 16x16 Matrix angeordnet.
- Jedem Wert (8-Bit Byte) wird ein Funktionswert wie folgt zugeordnet:

  Das Byte wird in zwei Hälften geteilt. Die ersten vier Bits beschreiben einen Wert zwischen 0 und 15 und adressieren eine Zeile der Matrix, die zweiten vier Bits adressieren eine Spalte der Matrix.

Beispiel: Das Byte 10100011 führt zur 12. Zeile (1010 = 12 dezimal bzw. C hexadezimal) und dort zur 3. Spalte (0011 = 3 dezimal bzw. 3 hexadezimal).

S-Box in Substitute Bytes

	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	F0	AD	D4	A2	AF	9C	A4	72	C0
2	B7	FD	93	26	36	3F	F7	CC	34	A5	E5	F1	71	D8	31	15
3	04	C7	23	C3	18	96	05	9A	07	12	80	E2	EB	27	B2	75
4	09	83	2C	1A	1B	6E	5A	A0	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	D0	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	DB
A	E0	32	3A	0A	49	06	24	5C	C2	D3	AC	62	91	95	E4	79
B	E7	C8	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	1F	4B	BD	8B	8A
D	70	3E	B5	66	48	03	F6	0E	61	35	57	B9	86	C1	1D	9E
E	E1	F8	98	11	69	D9	8E	94	9B	1E	87	E9	CE	55	28	DF
F	8C	A1	89	0D	BF	E6	42	68	41	99	2D	0F	B0	54	BB	16

Beispiel: das Byte 10100011 wird interpretiert als hexadezimal C3: da 1010 = C und 0011 = 3. Dies führt zu Zeile C und dort Spalte 3. Im Feld C3 finden wir den Eintrag 2E. Somit wird C3 $\rightarrow$ 2E abgebildet.

Konstruktionsprinzip der S-Box

- Natürlich stellt man sich die Frage, warum die durch die S-Box realisierte Permutation genau so gewählt wurde und nicht anders.
- Die Entwickler haben darauf geachtet, dass das Design der S-Box dazu beiträgt, bekannte kryptoanalytische Methoden zu verhindern (Details hierzu würden einige Vorlesungen in Anspruch nehmen. Später etwas mehr hierzu).
- Umgekehrt möchte man gerne überzeugt sein, dass die Entwickler durch die spezielle Wahl der S-Box nicht absichtlich eine Schwachstelle eingebaut haben. Diese Befürchtung war stets ein schwierig auszuräumender Diskussionspunkt bei der Betrachtung von DES, dem Vorgänger von AES.
- Oben genannte Gründe führten dazu, dass die Entwickler nicht etwa durch systematisches Ausprobieren eine S-Box konstruiert haben. Vielmehr ergeben sich die Tabelleneinträge der S-Box aus einer auf alle Inputwerte gleichermaßen angewendeten arithmetischen Operation.
- Die zugehörigen Berechnungen erfolgen im endlichen Körper $GF(2^8)$.

Konstruktion der S-Box (1)

- Zu Beginn der SubBytes Transformation initialisieren wir die S-Box, indem wir alle 128 möglichen Werte für 8-Bit Vektoren spaltenweise in die 16x16 Matrix eintragen:

  Wir beginnen in der ersten Spalte mit hexadezimal {01}, {02}, …, {0F}, in der zweiten Spalte mit {11}, {12}, …, {1F}, bis zur letzten Spalte {F1}, {12}, …, {FF}.

- **Schritt 1**: wir modifizieren die Matrix nun wie folgt: wir ersetzen jedes Element durch sein multiplikatives Inverses in $GF(2^8)$. {00} hat kein multiplikatives Inverses, und wir bilden {00} auf sich selbst ab.

- **Schritt 2**: die Elemente der so erhaltenen Matrix mit den multiplikativen Inversen interpretieren wir für die nächste Operation wieder als Bytes in Binärform: $(b_7, \ldots, b_0)$, mit $b_i \in \{0,1\}$. Also beispielsweise {F1} = 11110001.

Nun ersetzen wir jedes Matrixelement $(b_7, \ldots, b_0) \rightarrow (y_7, \ldots, y_0)$:

$$y_i = b_i \oplus b_{(i+4) \bmod 8} \oplus b_{(i+5) \bmod 8} \oplus b_{(i+6) \bmod 8} \oplus b_{(i+7) \bmod 8} \oplus 01100011$$

Konstruktion der S-Box (2)

Schritt 2 (vorherige Folie) in Matrixdarstellung:

$$\begin{pmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \\ y_6 \\ y_7 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 \end{pmatrix} \begin{pmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \\ b_4 \\ b_5 \\ b_6 \\ b_7 \end{pmatrix} + \begin{pmatrix} 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \\ 0 \end{pmatrix}$$

- Oben beschriebene Schritte 1 und 2 wenden wir auf jeden der 16x16 Matrixeinträge an. Das Resultat ist die dargestellte S-Box.
- Diese S-Box nutzen wir zum Table lookup und ersetzen jedes unserer 16 Bytes des Input Square Arrays mit dem in der S-Box hinterlegten Byte des Output Square Arrays.

InvSubBytes (1)

Zur Ausführung der Entschlüsselung benötigen wir zu jedem Teilschritt die zugehörige inverse Operation. Zur Invertierung von SubBytes (abgekürzt ***InvSubBytes***) müssen wir die Inverse S-Box konstruieren. Diese dient dann wieder zum Table lookup. Zur Konstruktion der inversen S-Box müssen wir die zwei Schritte aus SubBytes in der Reihenfolge umkehren und beide Schritte invertieren:

- **Schritt 1'**: wir benötigen die Umkehrung der affinen Transformation (Schritt 2 aus SubBytes). Dies ermöglicht die folgende Formel:

  Wir ersetzen jedes Matrixelement $(y_7, \ldots, y_0) \rightarrow (b_7, \ldots, b_0)$ wie folgt:

  $b_i = y_{(i+2) \bmod 8} \oplus y_{(i+5) \bmod 8} \oplus y_{(i+7) \bmod 8} \oplus 00000101$

- **Schritt 2'**: die Umkehrung des ersten Schrittes aus SubBytes müssen wir einfach nur wieder jedes Element der Matrix ersetzen durch sein multiplikatives Inverses in $GF(2^8)$. Als Ergebnis bekommen wir die Inverse S-Box.

InvSubBytes (2)

Schritt 1’ aus InvSubBytes in Matrixdarstellung:

$$\begin{pmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \\ b_4 \\ b_5 \\ b_6 \\ b_7 \end{pmatrix} = \begin{pmatrix} 0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 \\ 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 1 & 0 \end{pmatrix} \begin{pmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \\ y_6 \\ y_7 \end{pmatrix} + \begin{pmatrix} 1 \\ 0 \\ 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{pmatrix}$$

Auf den hier beschriebenen Schritt 1’ folgt Schritt 2’. Das Ergebnis ist die gesuchte inverse S-Box.

Die inverse S-Box nutzen wir wieder zum Table lookup und ersetzen jedes unserer 16 Bytes des Input Square Arrays mit dem in der S-Box hinterlegten Byte des Output Square Arrays.

InvSubBytes (3)

Wir müssen noch prüfen, ob die in InvSubBytes angegebene affine Transformation tatsächlich die Operation aus SubBytes invertiert. Bezeichnen wir den Inputvektor (b_7, …, b_0) mit b, die Transformationsmatrix mit M, den zu addierenden konstanten Vektor 01000011 mit c, und das Ergebnis (y_7, …, y_0) mit y. Es ist also y = M b + c.

Ebenso bezeichnen wir die Transformationsmatrix in InvSubBytes mit M’ und den konstanten Vektor 00000101 mit d, so berechnen wir b’ = M’ y + d.

b’ = M’ y + d = M’ (M b + c) + d = M’M b + M’c + d

Wir müssen verifizieren, dass b’ = b:

$$\begin{pmatrix} b'_0 \\ b'_1 \\ b'_2 \\ b'_3 \\ b'_4 \\ b'_5 \\ b'_6 \\ b'_7 \end{pmatrix} = \begin{pmatrix} 0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 \\ 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 1 & 0 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 \end{pmatrix} \begin{pmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \\ b_4 \\ b_5 \\ b_6 \\ b_7 \end{pmatrix} + \begin{pmatrix} 0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 \\ 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 1 & 0 \end{pmatrix} \begin{pmatrix} 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \\ 0 \end{pmatrix} + \begin{pmatrix} 1 \\ 0 \\ 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{pmatrix}$$

Inverse S-Box in Substitute Bytes

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	52	09	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	00	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	DF	6E
A	47	F1	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	7F	A9	19	B5	4A	0D	2D	E5	7A	9F	93	C9	9C	EF
E	A0	E0	3B	4D	AE	2A	F5	B0	C8	EB	BB	3C	83	53	99	61
F	17	2B	04	7E	BA	77	D6	26	E1	69	14	63	55	21	0C	7D

Beispiel: im vorherigen Beispiel haben wir gesehen, dass C3 $\rightarrow$ 2E abgebildet wird. In der inversen S-Box sehen wir folgerichtig, dass beispielsweise Zeile 2 Spalte E zum Eintrag C3 führt, also 2E $\rightarrow$ C3 abgebildet wird.

Aufbau der Inversen S-Box (2)

Schritt 2 in Matrixdarstellung:

$$\begin{pmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \\ y_6 \\ y_7 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 \end{pmatrix} \begin{pmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \\ b_4 \\ b_5 \\ b_6 \\ b_7 \end{pmatrix} + \begin{pmatrix} 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \\ 0 \end{pmatrix}$$

Eigenschaften der S-Box

Einige Merkmale der S-Box, auf die beim Design Wert gelegt wurde:

- niedrige Korrelation zwischen Inputbits und Outputbits
- der Output ist keine lineare Funktion des Inputs; die Nichtlinearität wurde bewirkt durch Nutzung der multiplikativen Inversen in $GF(2^8)$
- der konstante Vektor der affin linearen Transformation wurde so gewählt, dass die S-Box keine Fixpunkte besitzt (d.h. kein Element auf sich selbst abgebildet wird); ebenso besitzt die S-Box keine "Opposite fixed points", das heisst, kein Element wird auf sein 1-Komplement (Negation aller Bits) abgebildet

ShiftRows

Nach SubBytes folgt die Operation ***ShiftRows***. Hierfür betrachten wir den 128-Bit Input State angeordnet als 4x4 Square Array, bestehend aus 16 Bytes B_i.

Jede der vier Zeilen des Square Array wird dann einer Shift Operation unterzogen.

Für i = 0 bis 3 rotieren wir die i-te Zeile um i Positionen nach links:

B_0	B_4	B_8	B_{12}
B_1	B_5	B_9	B_{13}
B_2	B_6	B_{10}	B_{14}
B_3	B_7	B_{11}	B_{15}

B_0	B_4	B_8	B_{12}
B_5	B_9	B_{13}	B_1
B_{10}	B_{14}	B_2	B_6
B_{15}	B_3	B_7	B_{11}

InvShiftRows

Die Umkehrung ***InvShiftRows*** zur Operation ShiftRows ist offensichtlich. Wir kehren die Linksshifts um in Rechtsshifts:

Für i = 0 bis 3 rotieren wir die i-te Zeile um i Positionen nach rechts:

B'_0	B'_4	B'_8	B'_{12}
B'_1	B'_5	B'_9	B'_{13}
B'_2	B'_6	B'_{10}	B'_{14}
B'_3	B'_7	B'_{11}	B'_{15}

B'_0	B'_4	B'_8	B'_{12}
B'_{13}	B'_1	B'_5	B'_9
B'_{10}	B'_{14}	B'_2	B'_6
B'_7	B'_{11}	B'_{15}	B'_3

MixColumns (1)

Nachdem die Operation ShiftRows die Zeilen des Square Arrays manipuliert hat, operiert MixColumns nun auf den Spalten des Square Arrays.

MixColumns verwendet Multiplikation von Elementen in GF(2^8). Die 16 Bytes des State Arrays entsprechen also jeweils einem Polynom $f_{i,j}$ vom Grad < 8 mit Koeffizienten aus {0,1}:

$f_{i,j} = f_{i,j,7}x^7 + \ldots + f_{i,j,1}x + f_{i,j,0} = (f_{i,j,7}, \ldots, f_{i,j,0})$, mit $f_{i,j,k} \in \{0,1\}$. Den 8-Bit Vektor stellen wir nicht binär sondern hexadezimal mit zwei Zeichen dar, z.B. 00000010 = **02**.

Das 4x4 State Array multiplizieren wir mit einer für AES fest vorgegebenen 4x4 Matrix M wie folgt:

02	03	01	01
01	02	03	01
01	01	02	03
03	01	01	02

M

$\times$

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

Input State Array

$=$

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

Output State Array

MixColumns (2)

Schreiben wir MixColumns statt obiger Matrixdarstellung Term für Term aus, erhalten wir folgende Gleichungen: für j = 0, …, 3:

$g_{0,j} = (02\ f_{0,j}) \oplus (03\ f_{1,j}) \oplus f_{2,j} \oplus f_{3,j}$

$g_{1,j} = f_{0,j} \oplus (02\ f_{1,j}) \oplus (03\ f_{2,j}) \oplus f_{3,j}$

$g_{2,j} = f_{0,j} \oplus f_{1,j} \oplus (02\ f_{2,j}) \oplus (03\ f_{3,j})$

$g_{3,j} = (03\ f_{0,j}) \oplus f_{1,j} \oplus f_{2,j} \oplus (02\ f_{3,j})$

wobei $g_{i,j}$, $f_{i,j}$ Polynome (Elemente) aus GF(2^8).

Anmerkung: die Koeffizienten der obigen Matrix M wurden einem sogenannten linearen Code mit bestimmten Eigenschaften nachempfunden (lineare Codes betrachten wir hier nicht weiter).

Anmerkung: Die Konstruktion von MixColumns wird mathematisch gelegentlich auch anders dargestellt, allerdings mit identischem Ergebnis.

InvMixColumns

Zur Invertierung von MixColumns genügt es, wenn wir in unserer Berechnung die Matrix M ersetzen durch eine zu M inverse Matrix M^{-1}. Wir sparen uns die aufwendige Berechnung und geben M^{-1} direkt an. Wir können verifizieren, dass M^{-1} M die Einheitsmatrix ergibt:

0E	0B	0D	09
09	0E	0B	0D
0D	09	0E	0B
0B	0D	09	0E

$\times$

02	03	01	01
01	02	03	01
01	01	02	03
03	01	01	02

$=$

01	00	00	00
00	01	00	00
00	00	01	00
00	00	01	01

folglich gilt

0E	0B	0D	09
09	0E	0B	0D
0D	09	0E	0B
0B	0D	09	0E

$\times$

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

$=$

$=$

0E	0B	0D	09
09	0E	0B	0D
0D	09	0E	0B
0B	0D	09	0E

$\times$

02	03	01	01
01	02	03	01
01	01	02	03
03	01	01	02

$\times$

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

$=$

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

AddRoundKey

- Die Operation AddRoundKey tut genau das, was ihr Name sagt: sie addiert den 128-Bit Rundenschlüssel bitweise per XOR auf das State array.
- Die inverse Operation InvAddRoundKey ist identisch, die zweimalige XOR-Operation hebt sich auf.
- Im Falle eines n-Runden AES benötigen wir n+1 Rundenschlüssel. Diese werden alle abgeleitet aus dem 128-Bit AES Key.
- Nun müssen wir noch betrachten, wie diese Schlüsselexpansion funktioniert.

AES Key Expansion: Basics

- Die Erzeugung der benötigten Rundenschlüssel hängt davon ab, welche AES-Version wir einsetzen. Zur Erinnerung: wir haben drei Optionen:
  - 128 Bit Schlüssellänge und 10 Runden $\Rightarrow$ 11 Round Keys mit 128 Bit erforderlich
  - 192 Bit Schlüssellänge und 12 Runden $\Rightarrow$ 13 Round Keys mit 128 Bit erforderlich
  - 256 Bit Schlüssellänge und 14 Runden $\Rightarrow$ 15 Round Keys mit 128 Bit erforderlich

Anmerkung: die XOR-Verknüpfung des Input State mit einem Rundenschlüssel zu Beginn der Verschlüsselung nennt man manchmal auch ***Key Whitening***.

- Wir haben bei den im AES enthaltenen Operationen meist auf 8-Bit Vektoren (Bytes) gearbeitet. Einen 128-Bit Inputstring haben wir also in 16 Bytes zerteilt, und diese Bytes je nach Anwendung als 4x4 Square array angeordnet.
- Für die Rundenfunktionen nehmen wir nun auch eine Aufteilung des 128-Bit State arrays in vier 32-Bit Strings vor, sogenannte ***Words*** W[i].

128-Bit AES Key Expansion: Algorithmus

- Wir betrachten zunächst den Key schedule für den Fall des AES mit 128 Bit Schlüssellänge.
- Insgesamt benötigen wir dann 11x128 Bit Rundenschlüssel, das entspricht 11 x 4 Words W[i] der Länge 32 Bit, also W[0] bis W[43].
- Zu Beginn füllen wir unsere 128 Schlüsselbits in die Worte W[0] bis W[3]. Zur Erzeugung der Worte W[4] … W[43] gehen wir wie folgt vor:

Für i = 1, .., 10 und

für j = 1,..., 3:

$W[4i + j] = W[4i + j - 1)] \oplus W[4(i-1) + j]$

für j = 0:

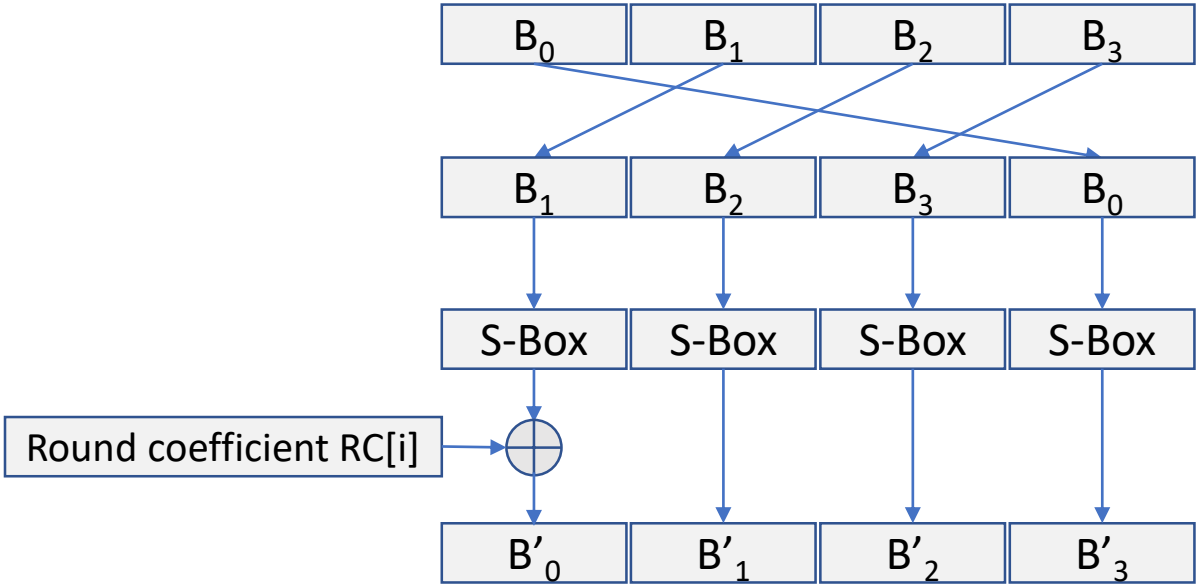
$W[4i] = W[4(i-1)] \oplus g(W[4i - 1])$

- Drei von vier Worten des folgenden Rundenschlüssels werden also gebildet, indem zwei frühere Worte per XOR verknüpft werden, und jeweils ein Wort in der Reihe wird auf kompliziertere Weise gebildet, unter Einbeziehung einer nichtlinearen Funktion g: $\{0,1\}^{32} \rightarrow \{0,1\}^{32}$.

AES Key Expansion: die Funktion g()

Die auf 32-Bit Worten operierende nichtlineare Funktion g ist folgendermaßen definiert: den Inputvektor teilen wir auf in vier 8-Bit Vektoren (Bytes) $B_0,\ldots, B_3$.

Die vier Bytes $B_0,\ldots, B_3$ werden wie folgt abgebildet auf $B'_0,\ldots, B'_3$:



Die verwendete S-Box kennen wir bereits. Betrachten wir also noch die RC[i].

AES Key Expansion: Round Coefficients

Der erste Rundenschlüssel besteht aus unserem originalen AES Key. Für die insgesamt 11 erforderlichen Rundenschlüssel müssen wir also noch 10 erzeugen. Dazu benötigen wir 10 Runden, und somit 10 sogenannte ***Round coefficients*** RC[0] bis RC[9].

Zur Berechnung interpretieren wir die 8-Bit Werte RC[i] als Polynome aus $GF(2^8)$, die wir modulo m(x) reduzieren, wobei $m(x) = x^8 + x^4 + x^3 + x + 1$ das irreduzible "AES-Polynom" ist.

Die RC[i] sind wie folgt definiert: $RC[i] = x^i \mod m(x)$, für alle i = 0,...,9. Wir erhalten:

$RC[0] = x^0 = 00000001_{bin} = 01_{hex}$

$RC[1] = x^1 = 00000010_{bin} = 02_{hex}$

$RC[2] = x^2 = 00000100_{bin} = 04_{hex}$

$RC[3] = x^3 = 00001000_{bin} = 08_{hex}$

$RC[4] = x^4 = 00010000_{bin} = 10_{hex}$

$RC[5] = x^5 = 00100000_{bin} = 20_{hex}$

$RC[6] = x^6 = 01000000_{bin} = 40_{hex}$

$RC[7] = x^7 = 10000000_{bin} = 80_{hex}$

$RC[8] = x^8 = x^8 + m(x) = x^8 + (x^8 + x^4 + x^3 + x + 1) = x^4 + x^3 + x + 1 = 00011011_{bin} = 1B_{hex}$

$RC[9] = x^9 = x \cdot x^8 = x \cdot (x^4 + x^3 + x + 1) = x^5 + x^4 + x^2 + x = 00110110_{bin} = 36_{hex}$

192-Bit AES Key Expansion (1)

- Betrachten wir jetzt AES mit 192 Bit Schlüssellänge. Nun benötigen wir insgesamt 13x128 Bit Rundenschlüssel (ein Whitening key zu Beginn und 12 Round keys) . Dies entspricht 13x4 = 52 WordsW[0] bis W[51]. Das Verfahren ist nahezu identisch mit jenem für 128 Bit Schlüssellänge. Nur hat der Key expansion Algorithmus diesmal nicht 4 Worte sondern 6 Worte in einer Reihe.
- Auch hier wird wieder auf das Byte links außen in der Reihe die Funktion g() angewandt.
- Wir starten mit unserem 192-Bit Cipher key und füllen diesen in die ersten 6 Words W[0] bis W[5].
- Pro Rundenschlüssel benötigen wir 4 Words. Den ersten (Whitening key) bilden wir mit W[0] bis W[3]. Den zweiten Round key beginnen wir mit dem Rest des Cipher keys W[4] und W[5]. Alle weiteren W[i] erhalten wir durch wiederholte Ausführung des Key expansion Algorithmus. Hierzu genügen 8 Iterationen, dann haben wir 6 + 6x8 = 54 Words (die beiden letzten Words benötigen wir also nicht).

192-Bit AES Key Expansion (2)

In Pseudocode sieht die Prozedur wie folgt aus:

Für i = 1, .., 8 und

für j = 1, ..., 5:

$W[6i + j] = W[6i + j - 1)] \oplus W[6(i-1) + j]$

für j = 0:

$W[6i] = W[6(i-1)] \oplus g(W[6i - 1])$

- 5 von 6 Worten der folgenden Iteration werden also gebildet, indem zwei frühere Worte per XOR verknüpft werden, und ein Wort der Reihe wird auf kompliziertere Weise gebildet, unter Einbeziehung der nichtlinearen Funktion $g: \{0,1\}^{32} \rightarrow \{0,1\}^{32}$.

256-Bit AES Key Expansion (1)

- Bei AES mit 256 Bit Schlüssellänge verändert sich das Verfahren geringfügig.
- Wir benötigen insgesamt 15x128 Bit Rundenschlüssel (ein Whitening key zu Beginn und 14 Round keys) .
- Dies entspricht 15x4 = 60 WordsW[0] bis W[59].
- Der Key expansion Algorithmus hat jetzt 8 Words in einer Reihe.
- Mit dieser aus 8 Words bestehende Reihe wird wie in den beiden vorherigen Fällen verfahren.
- Die Funktion g() wird jetzt allerdings nicht nur auf das Byte links außen angewandt, sondern auch auf das Byte 4 Positionen weiter rechts.
- Wir starten mit unserem 256-Bit Cipher key und füllen diesen in die ersten 8 Words W[0] bis W[7].

256-Bit AES Key Expansion (2)

- Pro Rundenschlüssel benötigen wir 4 Words.
- Den ersten (Whitening key) bilden wir mit W[0] bis W[3].
- Den zweiten Round key füllen wir mit der zweiten Hälfte des Cipher keys W[4] bis W[7].
- Alle weiteren W[i] erhalten wir durch wiederholte Ausführung des Key expansion Algorithmus.
- Hierzu genügen 7 Iterationen, dann haben wir 8 + 7x8 = 64 Words (die letzten 5 Words benötigen wir also eigentlich nicht und berechnen diese folglich auch nicht). In Pseudocode sieht die Prozedur wie folgt aus:
- [Hausaufgabe]
- 6 von 8 Words der folgenden Iteration werden also gebildet, indem zwei frühere Words per XOR verknüpft werden, und zwei Words der Reihe werden gebildet unter Einbeziehung der nichtlinearen Funktion g().

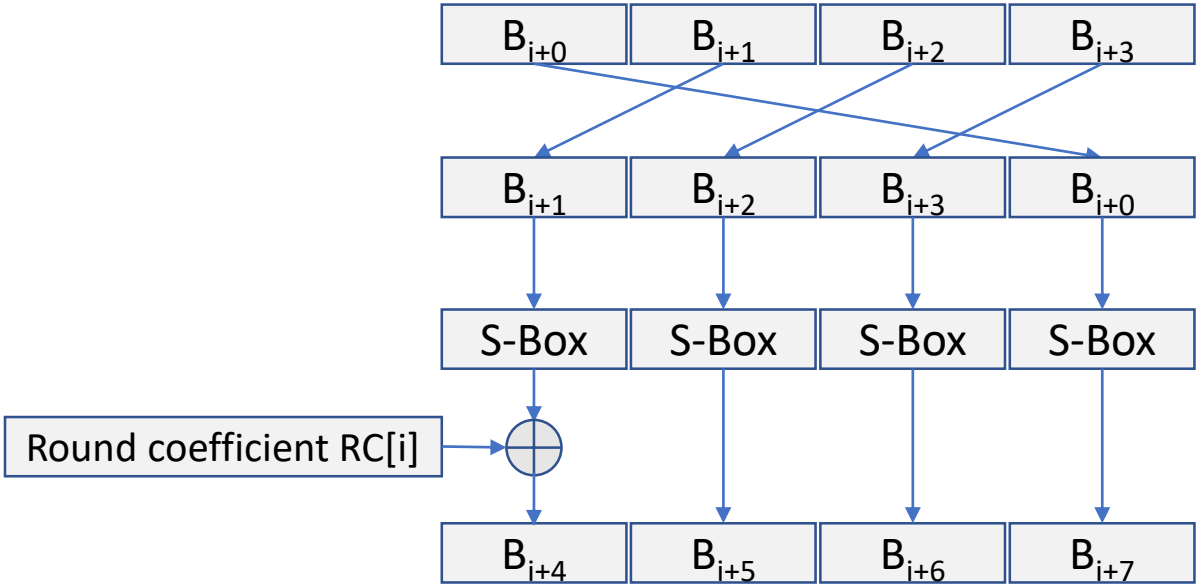
AES Key Expansion: Designkriterien

Die Erzeugung von Rundenschlüsseln aus einem Verschlüsselungsschlüssel kann zu Schwachstellen im Verschlüsselungsverfahren führen, wenn man beim Design der Key Expansion nicht sehr aufpasst. Im Falle von AES wurden explizit u.a. folgende Kriterien benannt:

- einfache Beschreibbarkeit des Verfahrens
- der rundenabhängige Round coefficient RC[i] soll etwaige Symmetrien beseitigen in der Erzeugung von Rundenschlüsseln
- kennt man einen Teil des Cipher keys oder eines Round keys, so soll dies nicht ermöglichen, viele (???) weitere Schlüsselbits zu errechnen
- hohe Ausführungsgeschwindigkeit
- hohe Diffusion, d.h. jedes Bit im Cipher key beeinflusst möglichst viele Bits im Round key
- ausreichende Nichtlinearität um zu verhindern, dass aus Cipher key Differenzen geschlossen werden kann auf Round key Differenzen

Ableitung Cipher Key aus Round Key

Eigentlich strebt man beim Design an, dass die Kenntnis eines Round keys nicht ermöglicht, den Cipher key zu ermitteln. Anderenfalls ist man anfällig z.B. gegen Side Channel Attacks mit dem Ziel, einen Rundenschlüssel zu extrahieren. Schauen wir nach, wie sich dies bei AES verhält (analog für Schlüssellängen 192 und 256):



Beispiel: angenommen wir haben B_{i+4} , B_{i+5} , B_{i+6} , B_{i+7} für ein $i \geq 0$. Alle in obiger Grafik dargestellten Operationen und Parameter sind bekannt und erlauben uns die Berechnung von B_i, B_{i+1}, B_{i+2}, B_{i+3}. Rekursiv können wir somit zurückrechnen bis B_0, B_1, B_2, B_3. Dies ist der Cipher key.

Rolle der AES-Teiloperationen (1)

Betrachten wir die einzelnen Operationen, aus denen sich die AES-Runden zusammensetzen, so sehen wir, dass jede einzelne unverzichtbar ist:

- **AddRoundKey**:

... ist die einzige schlüsselabhängige Operation. Ohne AddRoundKey bräuchte AES keinen Schlüssel...

- **KeyExpansion**:

... verhindert, dass jede Runde denselben Schlüssel verwendet und AES anfällig wird gegen eine sogenannte Slide Attack.

- **SubBytes**:

... gewährleistet Nichtlinearität. Ohne SubBytes könnte man AES als einfach lösbares, großes System linearer Gleichungen darstellen...

Rolle der AES-Teiloperationen (2)

- **ShiftRows**:

... sorgt dafür, dass die 4 Spalten des Square arrays durcheinander gemischt werden. Ohne ShiftRows bliebe jede Spalte (4 zusammenhängende Bytes im State array) von Anfang bis Ende der Verschlüsselung an derselben Position im Square array. Außerhalb der Spalte gäbe es keine Veränderung. Für die 2^{32} möglichen Vektoren einer jeden Spalte könnten wir eine große Tabelle ("Codebook") erstellen und zu jedem der 2^{32} = 4.294.967.296 Eingangswerte den zugehörigen Ausgangswert erfassen.

- **MixColumns**:

... ohne MixColumns würden Änderungen innerhalb eines Input Bytes immer nur Änderungen innerhalb eines einzelnen Output Bytes bewirken. Für jedes der insgesamt 16 Input Bytes könnten wir daher mittels Chosen Plaintext Attacke eine eigene Tabelle anlegen mit jeweils 2^8 = 256 Einträgen. Danach könnten wir jeden 128-Bit Eingabewert entschlüsseln.