Algorithmische Anwendungen

Prof. Dr. Heinrich Klocke

Algorithmisches Projekt
im Wintersemester 2005/06

# *Advanced Encryption Standard (AES)*
# *Rijndael – Algorithmus*

Gruppe: C_gelb

Thomas Boddenberg 11032925
Marcel Mörchen 11033153

# Inhalt(1)

Grober Überblick

Die Struktur
- Blockchiffre
  - Blockaufbau und Größen

Die Vorbereitung
- Key expansion
- Die S-Box
- Key addition

# Inhalt(2)

Die Rundentransformation

- byteSub()
- shiftRow()
- mixColumn()
- keyAddition()

Verschlüsselung

- Schlussrunde

Entschlüsselung

- Inverse Transformationsfunktionen

Design of AES

Erklärungen

# Grober Überblick

**Blockchiffre**

Rijndael ist ein symmetrischer Blockchiffre. Er ver- und entschlüsselt also blockweise Daten mit einem Schlüssel. Die Blockgröße und die Schlüssellänge können dabei unabhängig voneinander 128, 192 oder 256 Bit lang sein.

Jeder Block wird durch eine zweidimensionale Tabelle abgebildet, die immer aus 4 Zeilen besteht und je nach Größe der Daten aus 4 (128 Bit) bis 8 (256 Bit) Spalten. Jede Zelle dieser Tabelle ist genau 1 Byte (8 Bit) groß.
Datenblock:

| $A_{0,0}$ | $A_{0,1}$ | $A_{0,2}$ | $A_{0,3}$ |
|---|---|---|---|
| $A_{1,0}$ | $A_{1,1}$ | $A_{1,2}$ | $A_{1,3}$ |
| $A_{2,0}$ | $A_{2,1}$ | $A_{2,2}$ | $A_{2,3}$ |
| $A_{3,0}$ | $A_{3,1}$ | $A_{3,2}$ | $A_{3,3}$ |

Ein Block kann somit aus 16, 24 oder 32 Bytes (4, 6 oder 8 Worte) bestehen.

# Grober Überblick

Auf jedem dieser Blöcke werden nun, je nach länge des Schlüssels und der Blockgröße, Transformationen durchgeführt. Der Durchgang, der der diese Transformationen durchführt, wird mit „Runde“ beschrieben. Ein Block kann bis zu 14 Runden durchlaufen (min. 10).

Die Anzahl *r* dieser Runden variiert und ist von der Schlüssellänge *k* und Blockgröße *b* abhängig:
Je nach Blocklänge b und Schlüssellänge k wird die Anzahl der Runden bestimmt (10, 12 oder 14)

| r | b=128 | b=192 | b=256 |
|---|---|---|---|
| **k=128** | 10 | 12 | 14 |
| **k=192** | 12 | 12 | 14 |
| **k=256** | 14 | 14 | 14 |

# Grober Überblick

Die Anzahl der Möglichkeiten $2^n$ Eingabedaten zu permutieren beträgt: $(2^n)!$

Die Anzahl der Schlüsselbits errechnet sich aus :

$$\left\lceil \log_2 \left( (2^n)! \right) \right\rceil$$

| n | Permutationen | Schlüsselbits |
|---|---|---|
| 1 | 2! = 2 | 1 |
| 2 | 4! = 24 | 5 |
| 3 | 8! = 40320 | 16 |
| 4 | 16! = 20922789888000 | 45 |
| 8 | $256! \approx 10^{507}$ | 1684 |
| 16 | $65536! \approx 10^{287194}$ | 954037 |
| 32 | $(2^{32})! \approx 10^{3,95 \cdot 10^{10}}$ | $1,31 \cdot 10^{11}$ |
| 64 | $(2^{64})! \approx 10^{3,47 \cdot 10^{20}}$ | $1,15 \cdot 10^{21}$ |
| 128 | $(2^{128})! \approx 10^{1,29 \cdot 10^{40}}$ | $4,3 \cdot 10^{40}$ |

(Für eine Verschlüsselung von nur 8 Bit wäre somit ein Schlüssel von 1684 Bits notwendig, um alle Permutationen realisieren zu können!)

Vergleich zu DES: $2^{128-56} = 2^{72}$ ,
AES macht also das Durchprobieren aller Schlüssel um $2^{72}$ Mal aufwendiger als bei DES.

# Grober Überblick

Ablauf der Verschlüsselung:

Schlüsselexpansion
Vorrunde
KeyAddition ()
Verschlüsselungsrunden (wiederhole solange runde<r)
Substitution()
ShiftRow()
MixColumn()
KeyAddition()
Schlussrunde
Substitution()
ShiftRow()
KeyAddition()

S-Box:
Eine Substitutionsbox (S-Box) dient als Basis für eine monoalphabetische Verschlüsselung. Sie ist meist als Array implementiert und gibt an, welches Byte wie getauscht wird.

# Grober Überblick

Schlüsselexpansion:
Aufteilen des Schlüssels in r+1 Teilschlüssel (Rundenschlüssel).

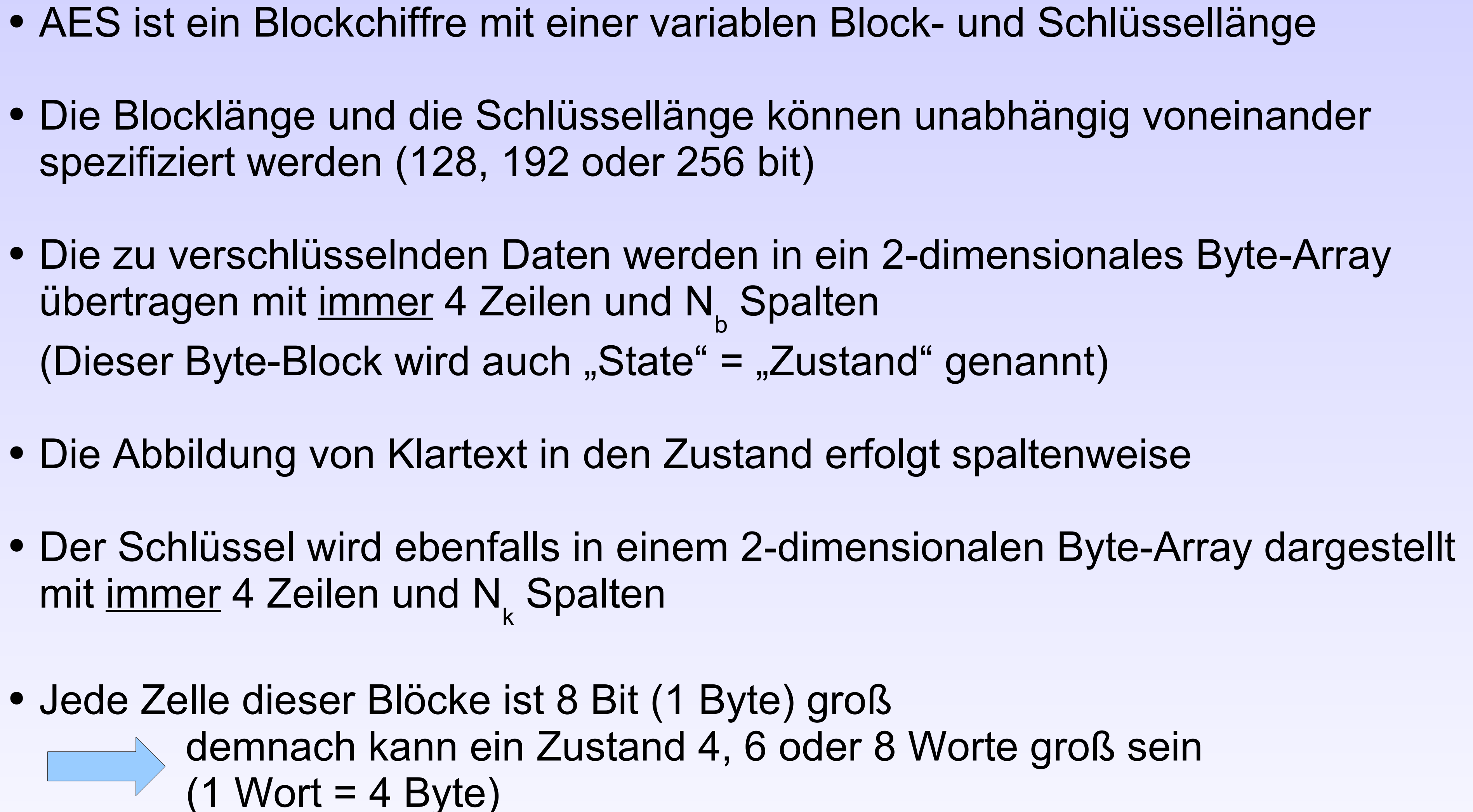
KeyAddition:
In der Vorrunde und nach jeder weiteren Verschlüsselungsrunde wird die KeyAddition ausgeführt. Hierbei wird eine bitweise XOR-Verknüpfung zwischen dem Block und dem aktuellen Rundenschlüssel vorgenommen.

Substitution:
Im ersten Schritt jeder Runde wird für jedes Byte im Block ein Äquivalent in der S-Box gesucht. Somit werden die Daten monoalphabetisch verschlüsselt.

ShiftRow:
In diesem Schritt werden die Zeilen um eine bestimmte Anzahl von Spalten nach links verschoben. Überlaufende Zellen werden von rechts fortgesetzt. Die Anzahl der Verschiebungen ist zeilen- und blocklängenabhängig

MixColumn:
Mischen der Spalten unter den einzelnen Blöcken. Es wird zunächst jede Zelle einer Spalte mit einer Konstanten multipliziert und anschließend die Ergebnisse XOR verknüpft.

# Die Struktur(1)



# Die Struktur(2)

Beispiel:

Zelle ist immer 1 Byte groß

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

Zustand mit $N_b = 6$

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

Schlüssel mit $N_k = 4$

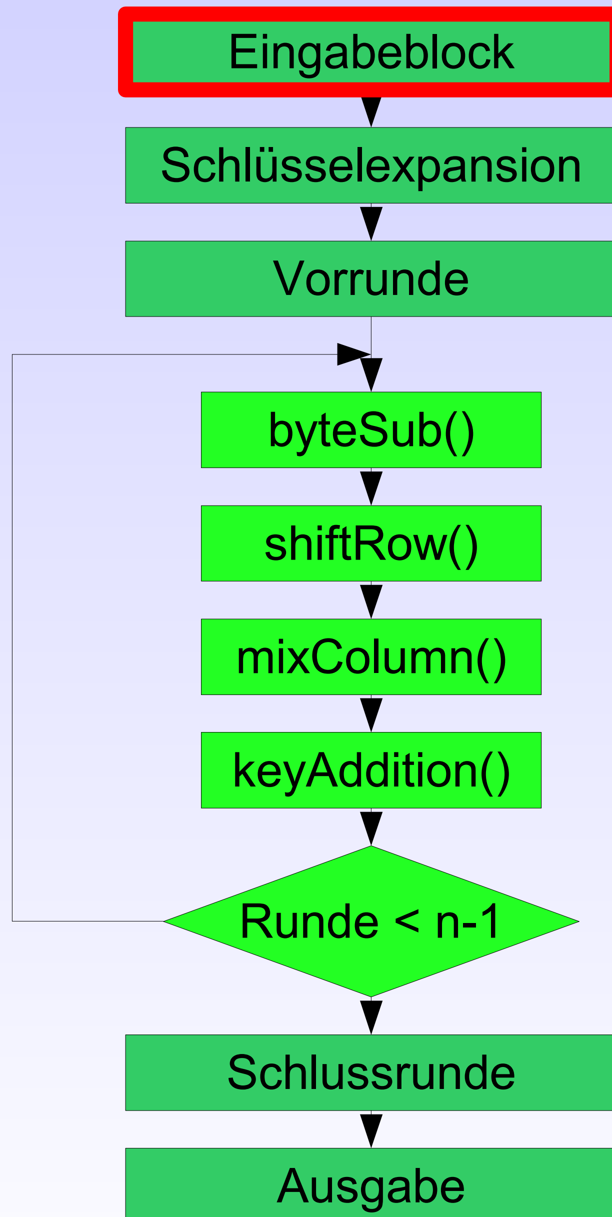
- Das Füllen der Arrays (Zustand und Schlüssel) erfolgt spaltenweise, also $a_{0,0}$, $a_{1,0}$, $a_{2,0}$, $a_{3,0}$, $a_{0,1}$, $a_{1,1}$, ...
- Die Anzahl der Runden $N_r$ (Transformationsdurchläufe) ist abhängig von $N_b$ und $N_k$

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

Table 1: Number of rounds (Nr) as a function of the block and key length.

Abbildung 1 - Rundentabelle
Quelle: [DJRV_99], Kapitel 4, Seite 10

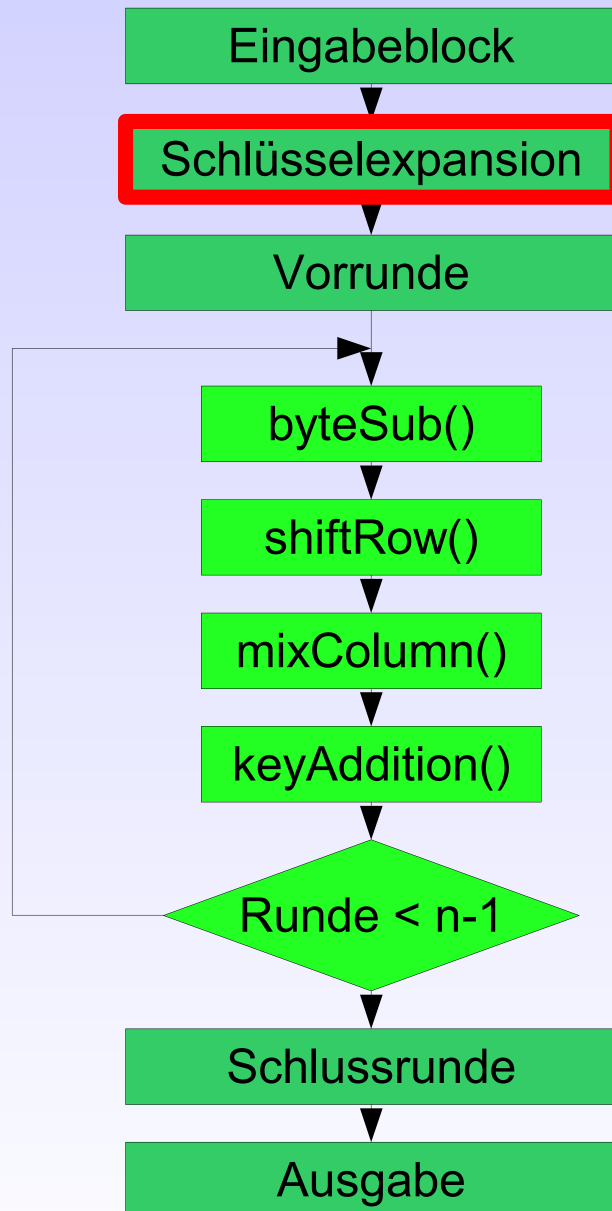
# Die Vorbereitung




**Der Eingabeblock**

Hier wird wie gerade gesprochen das 2-dimensionale Array mit Daten aufgefüllt

# Die Vorbereitung




**Die Schlüsselexpansion (1)**

Der expandierte Schlüssel ist ein 4-Byte Wort-Array mit $W [N_b * ( N_r + 1 )]$ .

Zur Schlüsselexpansion gilt:

$Anz_{Exp}$ = Blockgröße (in Worte) * (Rundenzahl + 1)
konkret:

$Anz_{Exp} = 6 * ( 12 + 1 )$

$Anz_{Exp}$ = <u>78 Worte</u> ( = 2496 Bits)

Die ersten $N_k$ (bei uns $N_k = 4$) Worte im expandierten Schlüssel beinhalten den „Hauptschlüssel“. Die restlichen 74 Worte werden nun rekursiv berechnet, wobei der jeweils vorherige Rundenschlüssel dem folgenden als Berechnungsgrundlage dient.
(Der erste Rundenschlüssel ist demzufolge der Hauptschlüssel.)

## Die Schlüsselexpansion (2)

Darstellung der Berechnung der weiteren Rundenschlüssel anhand eines Beispiels:

1. Rundenschlüssel (= “Hauptschlüssel“)

| | | | |
|---|---|---|---|
| A9 | 5B | FF | 4 |
| 8E | 8F | 5A | B8 |
| 3D | 6F | A5 | 33 |
| 9E | 17 | 9C | 2E |

2. Rundenschlüssel (abhängig vom 1. Rundenschlüssel)

| | | | |
|---|---|---|---|
| C4 | | | |
| | | | |
| | | | |
| | | | |

Das 1. Byte eines jeden Rundenschlüssels wird mit dem 3 Zeilen zuvor stehenden Byte der letzten Spalte XOR verknüpft, nachdem dieses Byte zuvor in der S-Box substituiert wurde. Zudem wird das 1. Byte auch mit einem Eintrag aus der rcon-Tabelle XORiert.

| | | | |
|---|---|---|---|
| Eintrag | | A9 | 10101001 |
| B8 | S-Box | 6C | 01101100 |
| Eintrag | rcon[0] | 1 | 00000001 |
| Ergebnis | | C4 | 11000100 |

## Die Schlüsselexpansion (3)

Darstellung der Berechnung der weiteren Rundenschlüssel anhand eines Beispiels:

1. Rundenschlüssel (= “Hauptschlüssel“)

| | | | |
|---|---|---|---|
| A9 | 5B | FF | 4 |
| 8E | 8F | 5A | B8 |
| 3D | 6F | A5 | 33 |
| 9E | 17 | 9C | 2E |

2. Rundenschlüssel (abhängig vom 1. Rundenschlüssel)

| | | | |
|---|---|---|---|
| C4 | 71 | | |
| | | | |
| | | | |
| | | | |

Die nächsten Bytes des Wortes werden berechnet, indem das nächste Byte des 1. Rundenschlüssels mit dem gerade berechneten Byte des 2. Rundenschlüssels XORiert wird. Dieses geschieht bis das neue Wort erzeugt wurde.
Das erste Byte des neuen Wortes wird äquivalent zu Schlüsselexpansion (2) erzeugt, jedoch ohne den Eintrag der rcon-Tabelle.

| | | |
|---|---|---|
| Eintrag | B5 | 10110101 |
| Eintrag | C4 | 11000100 |
| Ergebnis | 71 | 01110001 |

# Die Vorbereitung

## Die Schlüsselexpansion (4)

Bei einer Schlüsselexpansion von Nk > 6 wird jedes 4. Byte eines Wortes innerhalb des neu zu erstellenden Rundenschlüssels noch einmal durch die S-Box substituiert. Dies bewirkt eine zusätzliche Durchmischung bei grossen Schlüsseln und verhindert das Entstehen von neuen Regelmässigkeiten innerhalb des expandierten Schlüssels.

## Rechenoperationen im AES

**Addition ⊕**

Die Addition bei AES ist gleichzusetzen mit einer XOR-Verknüpfung auf byte-level.

Beispiel:

57 + 83 = D4 $( x^6 + x^4 + x^2 + x + 1 ) + ( x^7 + x + 1 ) = x^7 + x^6 + x^4 + x^2$

```
   01010111
+  10000011
   11010100
```

**Multiplikation ●** (später)

## Die S-Box

Die S-Box (Substitutions-Box) ist eine Tabelle, in der byteweise Daten ersetzt werden.
Die Daten in der S-Box lassen sich wie folgt erzeugen:

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

Abbildung 2 – affine Transformation
Quelle: [DRRV_99], Kapitel 4, Seite 7

Beispiel:

Substituieren der hex. Zahl 00

0*1 + 0*0 + 0*0 + ... = 0
0*1 + 0*1 + 0*0 + ... = 0
0*1 + 0*1 + 0*1 + ... = 0
...

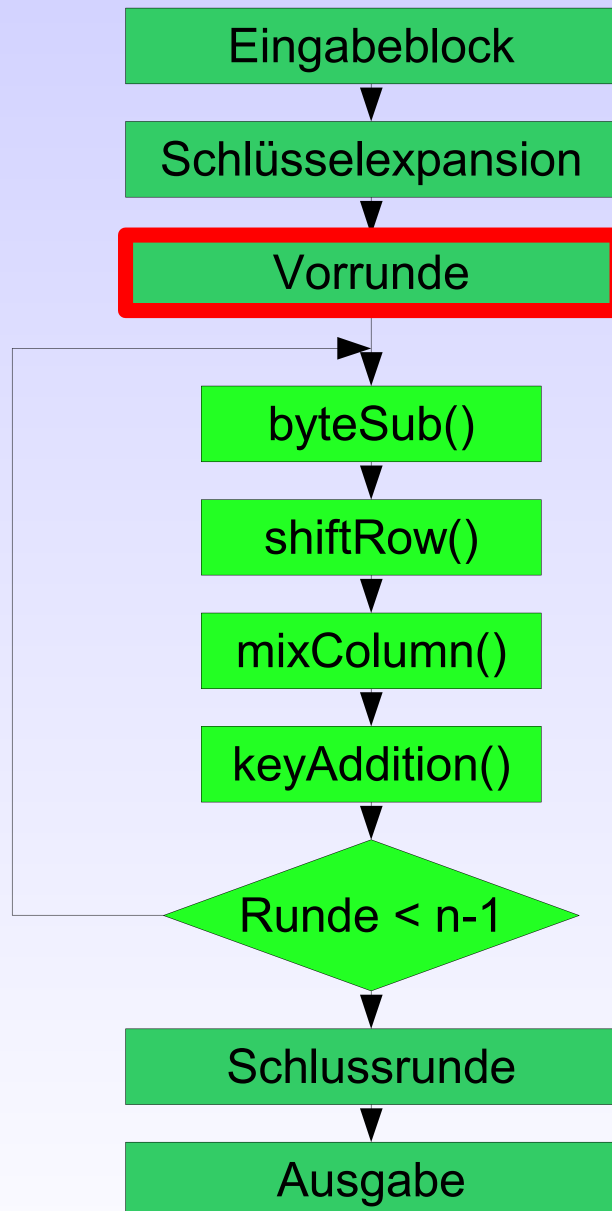
Ergebnis $= (00000000)_2 + (01100011)_2$
$= (00)_{16} + (63)_{16}$
$\Rightarrow (63)_{16}$

## Die S-Box

| | 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 | 1 | 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 | 4 | C7 | 23 | C3 | 18 | 96 | 5 | 9A | 7 | 12 | 80 | E2 | EB | 27 | B2 | 75 |
| 4 | 9 | 83 | 2C | 1A | 1B | 6E | 5A | A0 | 52 | 3B | D6 | B3 | 29 | E3 | 2F | 84 |
| 5 | 53 | D1 | 0 | ED | 20 | FC | B1 | 5B | 6A | CB | BE | 39 | 4A | 4C | 58 | CF |
| 6 | D0 | EF | AA | FB | 43 | 4D | 33 | 85 | 45 | F9 | 2 | 7F | 50 | 3C | 9F | A8 |
| 7 | 51 | A3 | 40 | 8F | 92 | 9D | 38 | F5 | BC | B6 | DA | 21 | 10 | FF | F3 | D2 |
| 8 | CD | C | 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 | B | DB |
| A | E0 | 32 | 3A | A | 49 | 6 | 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 | 8 |
| C | BA | 78 | 25 | 2E | 1C | A6 | B4 | C6 | E8 | DD | 74 | 1F | 4B | BD | 8B | 8A |
| D | 70 | 3E | B5 | 66 | 48 | 3 | F6 | E | 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 | D | BF | E6 | 42 | 68 | 41 | 99 | 2D | F | B0 | 54 | BB | 16 |

Abbildung 3 - Substitutions-Box

# Die Vorbereitung




## Die Vorrunde

In der Vorrunde wird die Funktion keyAddition() ausgeführt. Hierbei wird eine bitweise XOR - Verknüpfung zwischen dem Block und dem aktuellen Rundenschlüssel vorgenommen.

| 20 | 47 | 72 | 75 |
|---|---|---|---|
| 70 | 70 | 65 | 20 |
| 43 | 5F | 67 | 65 |
| 6C | 62 | 20 | 20 |

Block (hex)

|  | G | r | u |
|---|---|---|---|
| p | p | e |  |
| C | _ | g | e |
| l | b |  |  |

Klartext

| C4 | 97 | b2 | 55 |
|---|---|---|---|
| BC | 25 | DA | B1 |
| D0 | 17 | AA | 64 |
| E8 | F7 | 46 | 64 |

Schlüssel

| E4 | D0 | C0 | 20 |
|---|---|---|---|
| CC | 55 | BF | 91 |
| 93 | 48 | CD | 1 |
| 84 | 95 | 66 | 44 |

Verschlüsselt

| ä | Ð | À |  |
|---|---|---|---|
| Ì | U | Ú | ± |
| " | H | Í | ^A |
| „ | • | f | D |

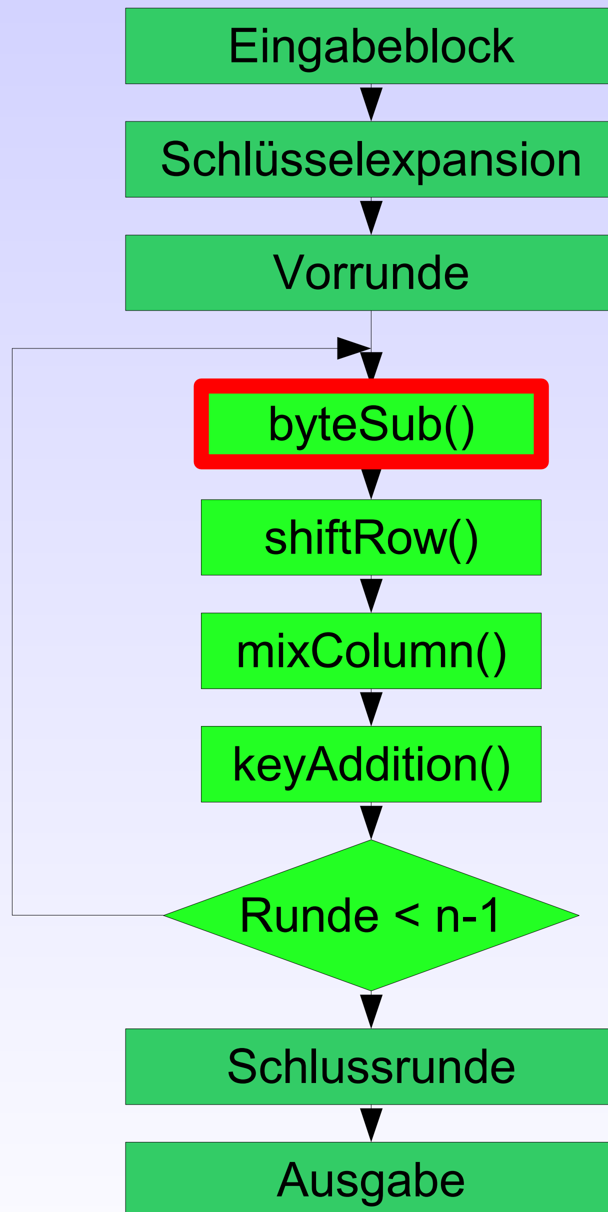
Klartext

Beispiel:

$$
\begin{array}{lll}
 & 00100000 & (20)_{16} \\
\underline{\text{XOR}} & \underline{11000100} & (C4)_{16} \\
 & 11100100 & (E4)_{16}
\end{array}
$$

Die Vorrunde soll gegen *known-plaintext-attacks* schützen. Dabei ist dem Gegner ein zusammen - gehöriges Paar Klartext/Geheimtext aus dem er den Schlüssel zurückführen kann.

# byteSub() Transformation




In der Funktion byteSub() werden alle Zellen eines Blocks mit der S-Box byteweise substituiert.

| | 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 | 1 | 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 | 4 | C7 | 23 | C3 | 18 | 96 | 5 | 9A | 7 | 12 | 80 | E2 | EB | 27 | B2 | 75 |
| 4 | 9 | 83 | 2C | 1A | 1B | 6E | 5A | A0 | 52 | 3B | D6 | B3 | 29 | E3 | 2F | 84 |
| 5 | 53 | D1 | 0 | ED | 20 | FC | B1 | 5B | 6A | CB | BE | 39 | 4A | 4C | 58 | CF |
| 6 | D0 | EF | AA | FB | 43 | 4D | 33 | 85 | 45 | F9 | 2 | 7F | 50 | 3C | 9F | A8 |
| 7 | 51 | A3 | 40 | 8F | 92 | 9D | 38 | F5 | BC | B6 | DA | 21 | 10 | FF | F3 | D2 |
| 8 | CD | C | 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 | B | DB |
| A | E0 | 32 | 3A | A | 49 | 6 | 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 | 8 |
| C | BA | 78 | 25 | 2E | 1C | A6 | B4 | C6 | E8 | DD | 74 | 1F | 4B | BD | 8B | 8A |
| D | 70 | 3E | B5 | 66 | 48 | 3 | F6 | E | 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 | D | BF | E6 | 42 | 68 | 41 | 99 | 2D | F | B0 | 54 | BB | 16 |

Abbildung 1 - Substitutions-Box

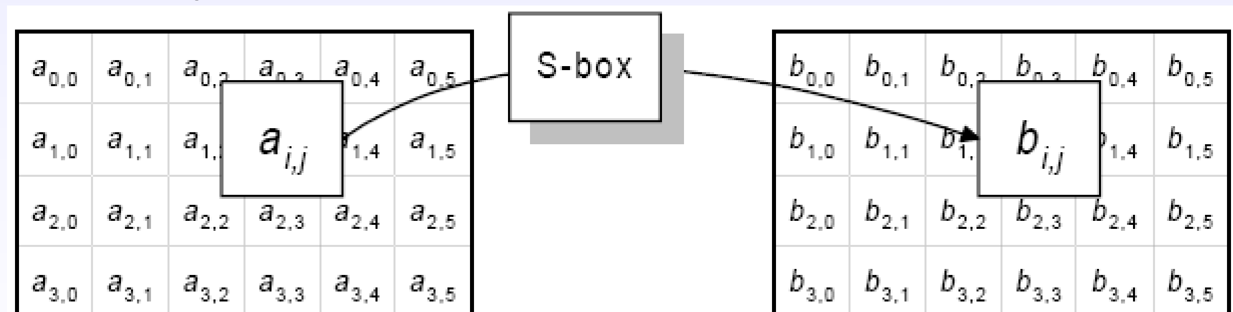
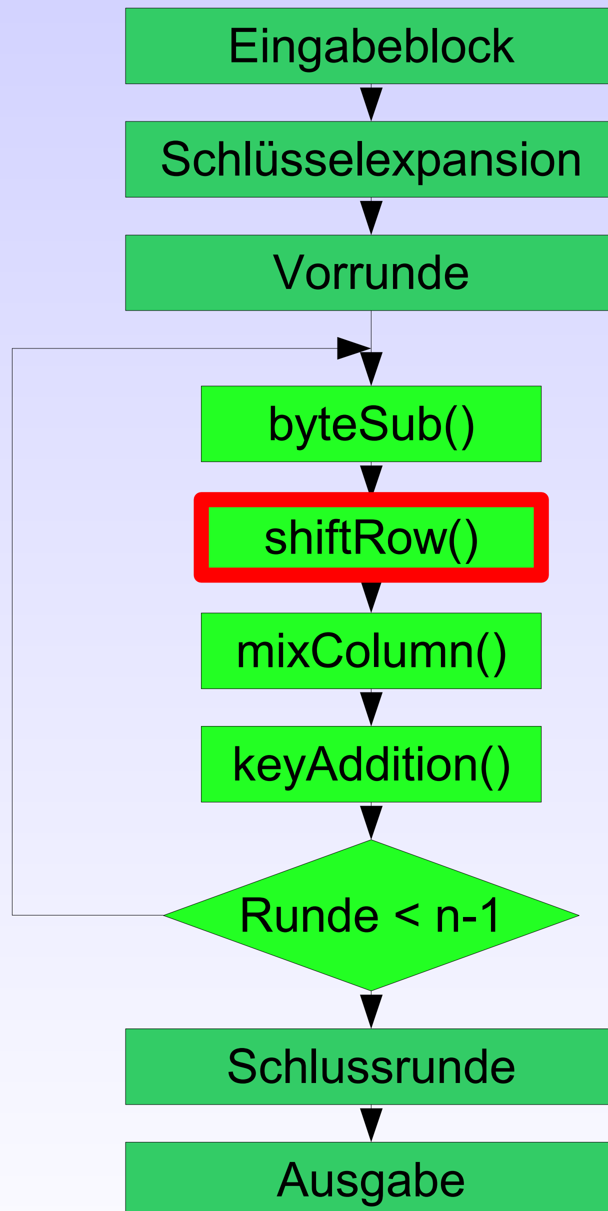



Abbildung 2 – Vorgang von byteSub()
Quelle: [DRRV_99], Kapitel 4.2.1, Seite 11

# shiftRow() Transformation




In der Funktion shiftRow() werden die Zeilen eines Blocks zyklisch verschoben. Um wieviel Zellen sich eine Zeile verschiebt, hängt von $N_b$ ab.

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

Abbildung 3 – Shift-Tabelle
Quelle: [DRRV_99], Kapitel 4.2.2, Seite 12

Die Zeilen werden beim Verschieben nach links und nicht nach rechts verschoben. Die erste Zeile eines Blocks wird hier allerdings niemals verschoben!

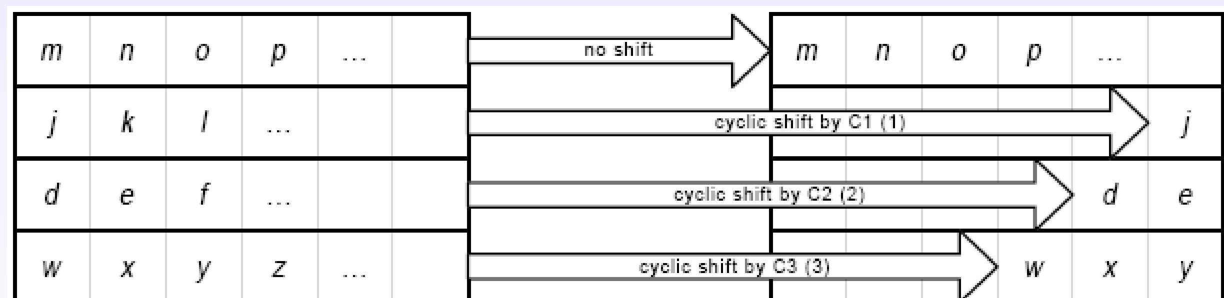



Abbildung 4 – shiftRow() Vorgang
Quelle: [DRRV_99], Kapitel 4.2.2, Seite 12

Die Multiplikation:

Die Multplikation erfolgt auf Basis einer Polynom-Multiplikation mit anschließender modulo Operation  mit einem festgelegten Polynom 8. Grades:

**Example:** '57' • '83' = 'C1', or:

$$(x^6 + x^4 + x^2 + x + 1)(x^7 + x + 1) = x^{13} + x^{11} + x^9 + x^8 + x^7 + x^7 + x^5 + x^3 + x^2 + x + x^6 + x^4 + x^2 + x + 1$$

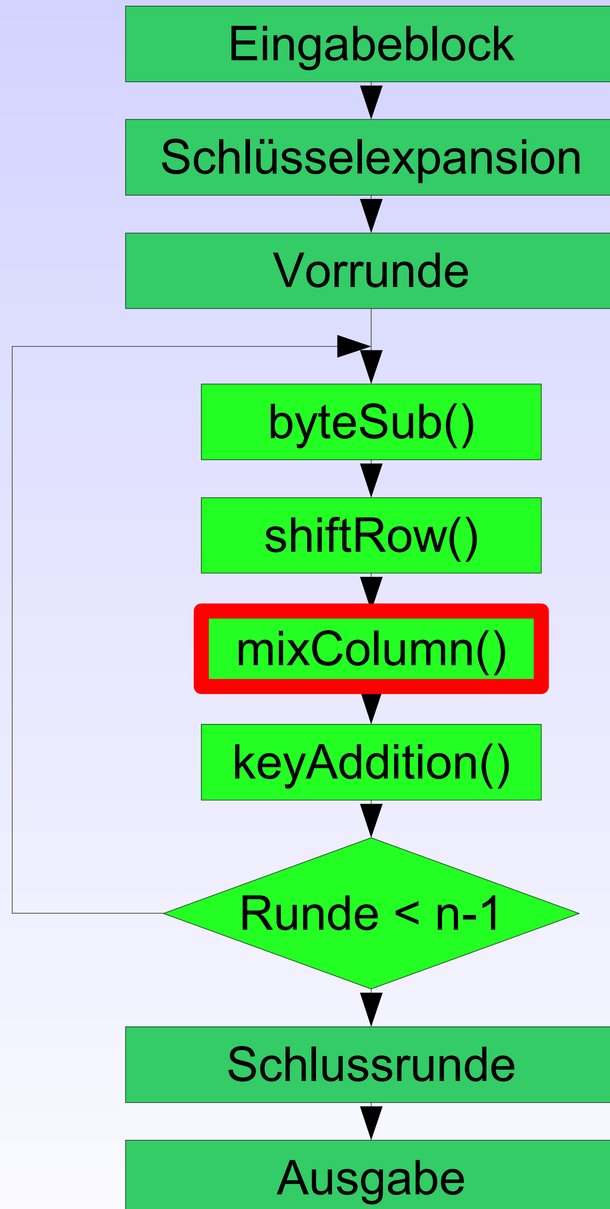
$$= x^{13} + x^{11} + x^9 + x^8 + x^6 + x^5 + x^4 + x^3 + 1$$

$$x^{13} + x^{11} + x^9 + x^8 + x^6 + x^5 + x^4 + x^3 + 1 \text{ modulo } x^8 + x^4 + x^3 + x + 1$$

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

Abbildung 5 –Beispiel einer Multiplikation
Quelle: [DRRV_99], Kapitel 2.1.2, Seite 5

# mixColumn() Transformation




MixColumn() durchmischt die Zellen einer Spalte bitweise, indem die Einträge in einer Spalte mit einer Konstanten multipliziert werden, und das Ergebnis anschließend mit den vorherigen Ergebnis XORiert wird.

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

($a_0$ bis $a_3$ sind Spalteneinträge)

Abbildung 6 –Matrix Multiplikation bei mixColumn()
Quelle: [DRRV_99], Kapitel 4.2.3, Seite12

# mixColumn() Transformation

Beispiel:

| D1 | C3 | 68 | 0B |
| --- | --- | --- | --- |
| 21 | 09 | 4B | 45 |
| 5F | A3 | 59 | 9B |
| 21 | CC | F2 | F9 |

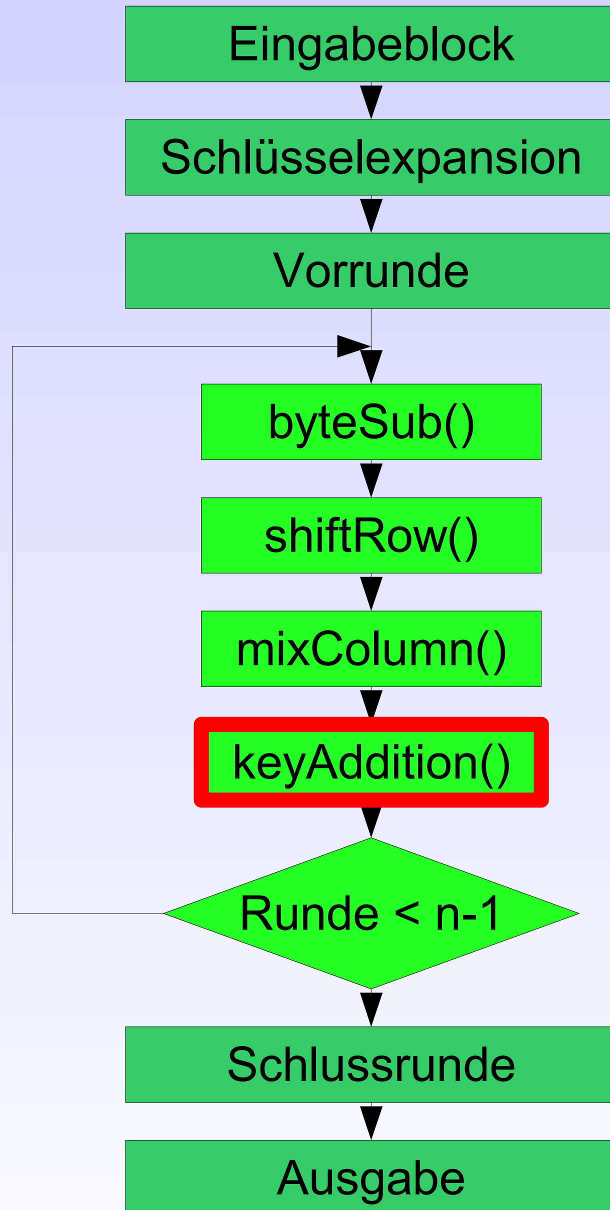
| | | | | |
| --- | --- | --- | --- | --- |
| 11010001 | * | 00000010 | 10111001 | |
| 00100001 | * | 00000011 | 01100011 | XOR |
| | | | 11011010 | |
| 01011111 | * | 00000001 | 01011111 | XOR |
| | | | 10000101 | |
| 00100001 | * | 00000001 | 00100001 | XOR |
| | | | 10100100 | |

| A4 | | | |
| --- | --- | --- | --- |
| | | | |
| | | | |
| | | | |

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

Abbildung 7 –Matrix Multiplikation bei mixColumn()
Quelle: [DRRV_99], Kapitel 4.2.3, Seite12

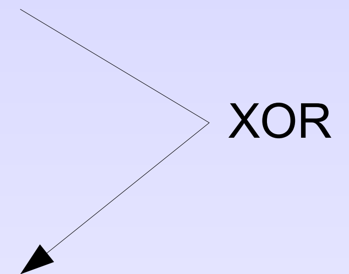
# keyAddition() Transformation




In keyAddition() wird er jeweilige Rundenschlüssel mit den Blockeinträgen XORiert.

| C8 | 72 | 3F | 52 |
|---|---|---|---|
| F4 | 8B | 1F | 8E |
| 04 | 01 | 51 | 6 |
| 1A | 5B | 8E | 3D |

Blockdaten



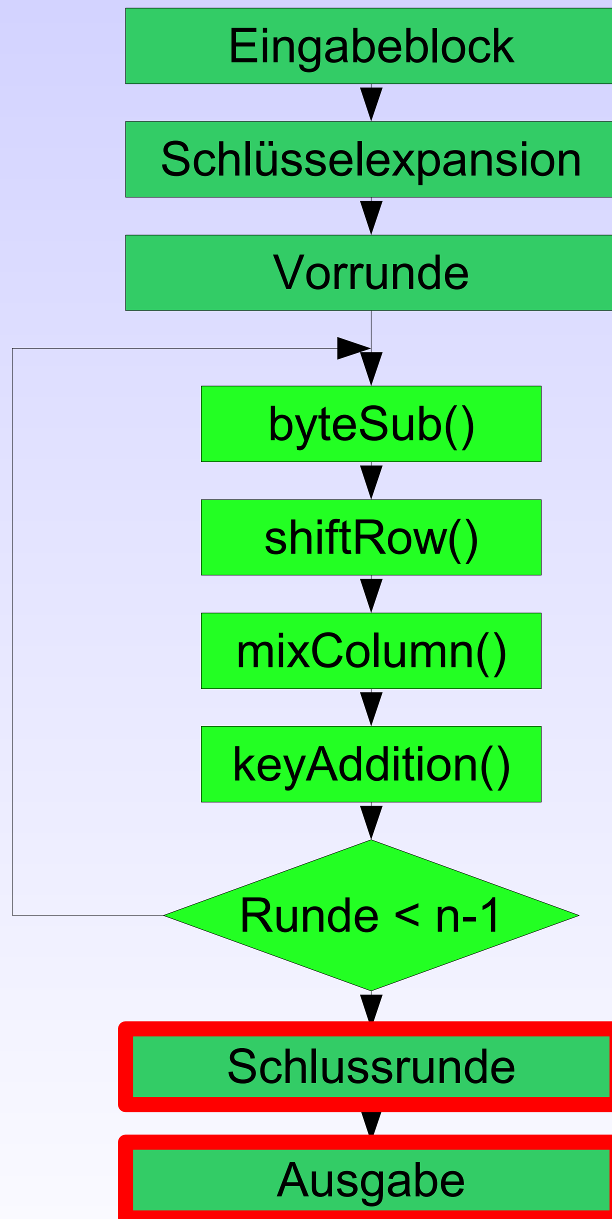

| D9 | 96 | 1C | 18 |
|---|---|---|---|
| A8 | 14 | F3 | 74 |
| 6C | 39 | AA | B8 |
| 60 | 9C | F4 | 1B |

Rundenschlüssel

| 11 | E4 | 23 | 4A |
|---|---|---|---|
| 5C | 9F | EC | FA |
| 68 | 38 | FB | BE |
| 7A | C2 | 7A | 26 |

Blockdaten nach XOR

# Schlussrunde




Die Schlussrunde ist ebenfalls eine Transformationsrunde, wobei allerdings nicht alle Funktionen durchgeführt werden.

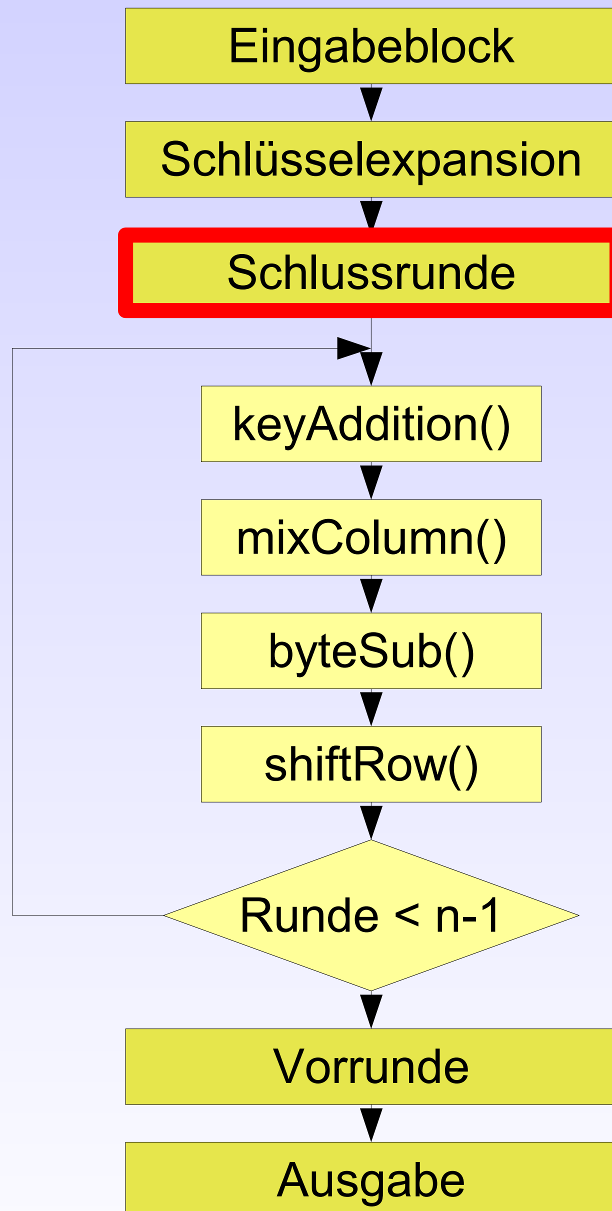
Aufruf der Funktionen:

- byteSub()
- shiftRow()
- keyAddition()

Die Durchführung der aufgerufenen Funktionen erfolgt analog zu den der Rundentransformationen.

Die einzelnen verschlüsselten Blöcke werden wieder zu einer Datei zusammengefasst und ausgegeben. Danach lässt sich die Datei nur mit dem gleichen Algorithmus und Schlüssel wieder in den Orginalzustand bringen.

# Entschlüsselung



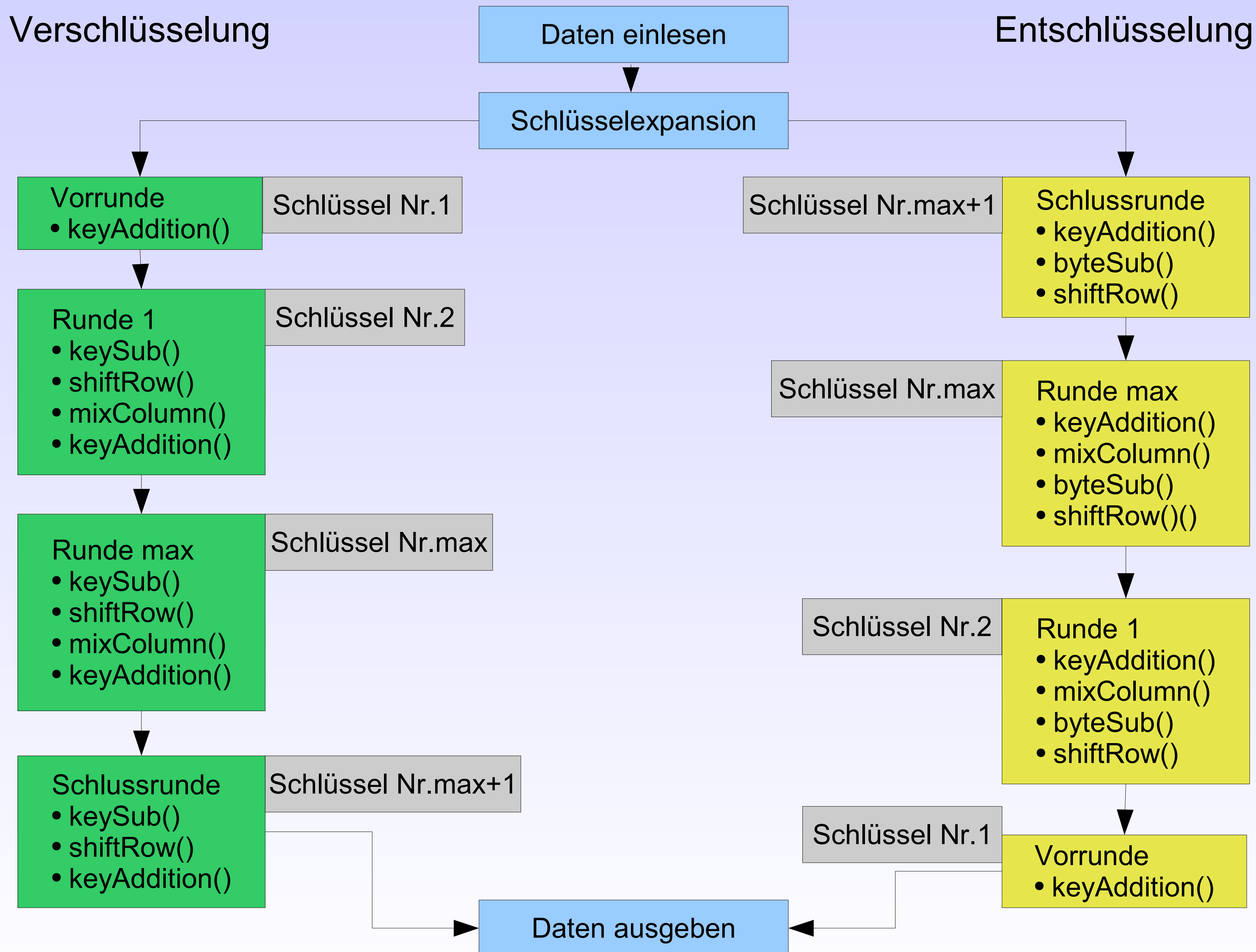

Bei der Entschlüsselung von Daten wird genau rückwärts vorgegangen. Die Daten werden zunächst wieder in zweidimensionale Tabellen gelesen und die Rundenschlüssel generiert. Allerdings wird nun mit der Schlussrunde angefangen und alle Funktionen in jeder Runde in der umgekehrten Reihenfolge aufgerufen. Die Reihenfolge von shiftRow() und byteSub() ist gleichgültig:
shiftRow() arbeitet auf die Zeilen der Blöcke ohne ihre Werte zu ändern. ByteSub() arbeitet auf die Blockeinträge, unabhängig von deren Position.

Jedoch müssen vor Entschlüsselungsbeginn sämtliche Rundenschlüssel vorliegen, da mit dem letzten Rundenschlüssel begonnen wird und dieser sich aus dem Vorhergehenden berechnet.

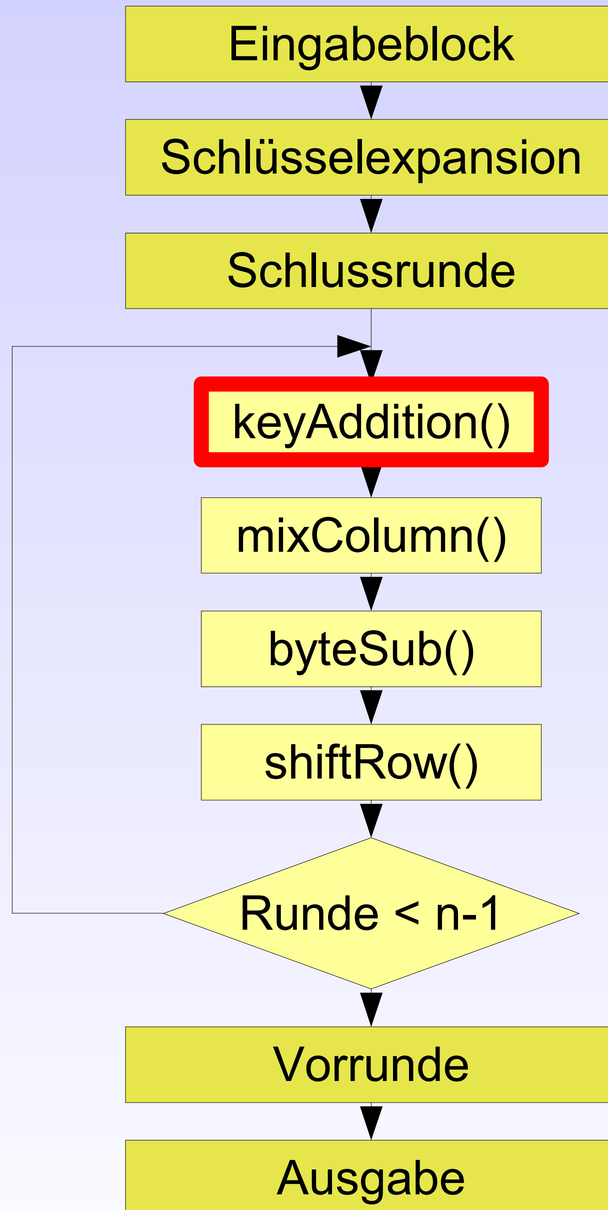
Die Dürchführung der Schlussrunde erfolgt in der Reihenfolge:

- keyAddition()
- byteSub()
- shiftRow()

# Entschlüsselung



# Inverse Transformationsfunktionen




Jeder Schritt der Verschlüsselung wird von hinten nach vorne rückgängig gemacht.
Um die Daten nun zu dechiffrieren müssen die Transformationsfunktionen ihre Funktionalität invers durchführen. Da es hier viele XOR Verknüpfungen gibt, müssen nur wenige grundlegende Änderungen an den „inversen" Funktionen stattfinden.

| C8 | 72 | 3F | 52 |
|---|---|---|---|
| F4 | 8B | 1F | 8E |
| 04 | 01 | 51 | 6 |
| 1A | 5B | 8E | 3D |

Blockdaten nach inverser XOR Operation

| D9 | 96 | 1C | 18 |
|---|---|---|---|
| A8 | 14 | F3 | 74 |
| 6C | 39 | AA | B8 |
| 60 | 9C | F4 | 1B |

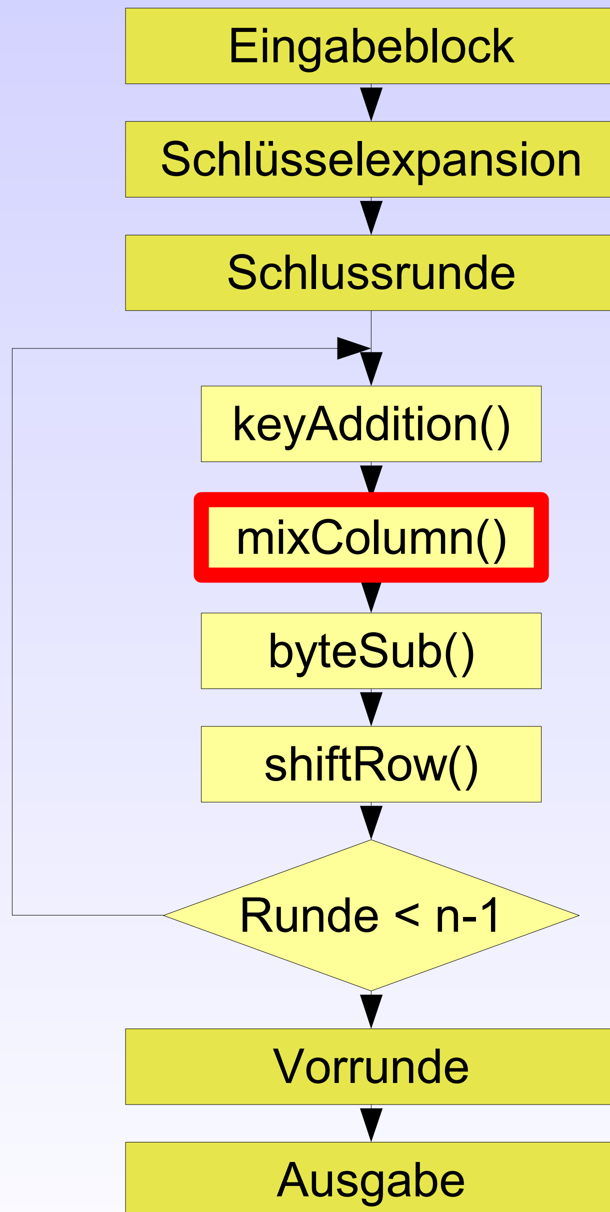
Rundenschlüssel

XOR invers

| 11 | E4 | 23 | 4A |
|---|---|---|---|
| 5C | 9F | EC | FA |
| 68 | 38 | FB | BE |
| 7A | C2 | 7A | 26 |

Blockdaten

# Inverse Transformationsfunktionen




Bei der inversen Funktion mixColumn() muss das inverse Polynom '0B'$x^3$+'0D'$x^2$+'09'x+'0E' verwendet werden, um eine Rückführung der Daten zu ermöglichen!
Da hier offensichtlich größere Koeffizienten als beim Verschlüsseln vorliegen, ist das Entschlüsseln aufwendiger !
Beispiel:

| | | | |
|---|---|---|---|
| 52 | 8C | 2c | EA |
| 46 | 3F | E8 | DA |
| 8E | 18 | 40 | 98 |
| 58 | 55 | E9 | C4 |

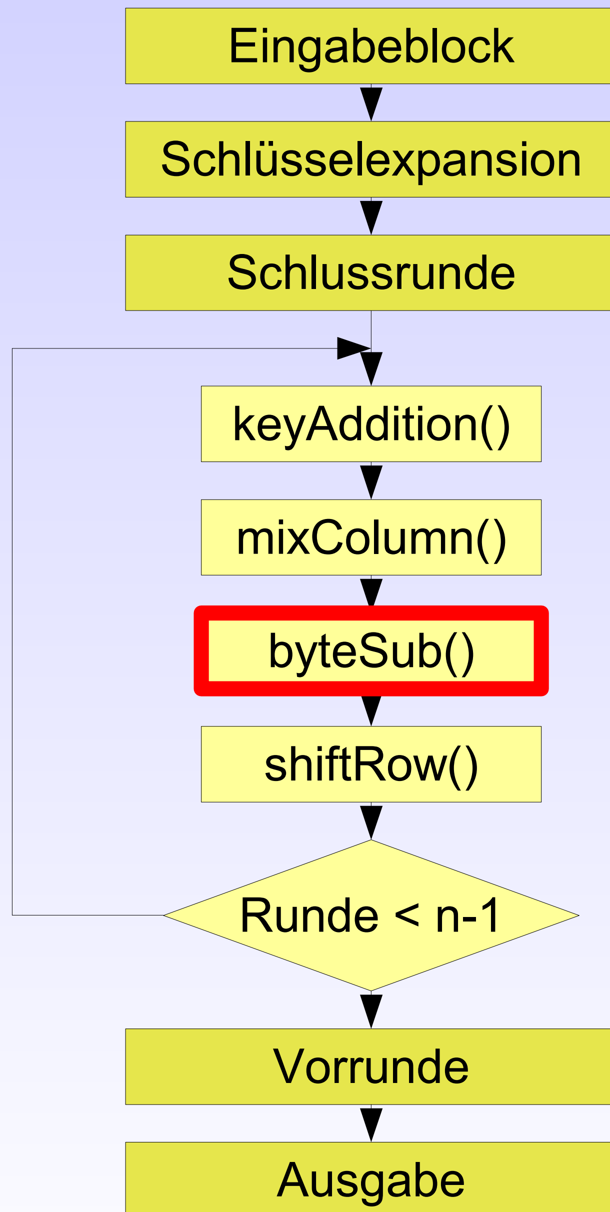
Blockdaten vor inverser mixColumn()

| | | | | |
|---|---|---|---|---|
| 01010010 | * | 00001110 | 01010001 | |
| | | | | XOR |
| 01000110 | * | 00001011 | 11001100 | |
| | | | 10011101 | |
| 10001110 | * | 00001101 | 10011100 | XOR |
| | | | 0000001 | XOR |
| 01011000 | * | 00001001 | 10101110 | |
| | | | 10101111 | |

| | | | |
|---|---|---|---|
| AF | | | |
| | | | |
| | | | |
| | | | |

Blockdaten nach inverser mixColumn()

# Inverse Transformationsfunktionen




In der inversen Funktion byteSub() werden alle Zellen eines Blocks mit der inversen S-Box byteweise substituiert.

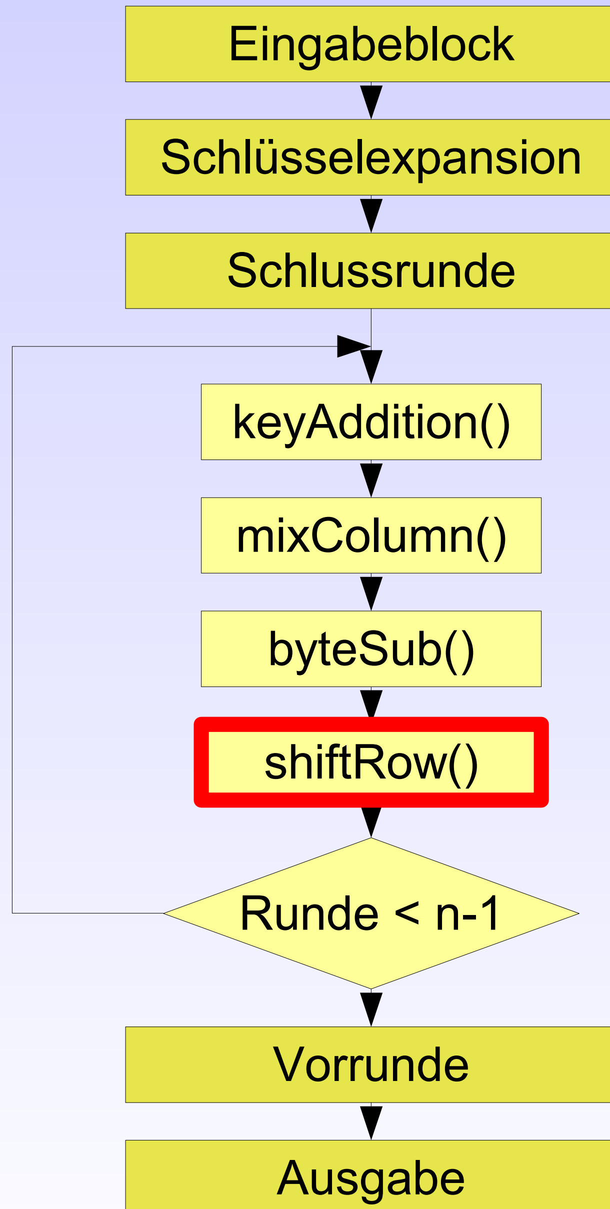
| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | a | b | c | d | e | f |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 00 | 63 | 7c | 77 | 7b | f2 | 6b | 6f | c5 | 30 | 01 | 67 | 2b | fe | d7 | ab | 76 |
| 10 | ca | 82 | c9 | 7d | fa | 59 | 47 | f0 | ad | d4 | a2 | af | 9c | a4 | 72 | c0 |
| 20 | b7 | fd | 93 | 26 | 36 | 3f | f7 | cc | 34 | a5 | e5 | f1 | 71 | d8 | 31 | 15 |
| 30 | 04 | c7 | 23 | c3 | 18 | 96 | 05 | 9a | 07 | 12 | 80 | e2 | eb | 27 | b2 | 75 |
| 40 | 09 | 83 | 2c | 1a | 1b | 6e | 5a | a0 | 52 | 3b | d6 | b3 | 29 | e3 | 2f | 84 |
| 50 | 53 | d1 | 00 | ed | 20 | fc | b1 | 5b | 6a | cb | be | 39 | 4a | 4c | 58 | cf |
| 60 | d0 | ef | aa | fb | 43 | 4d | 33 | 85 | 45 | f9 | 02 | 7f | 50 | 3c | 9f | a8 |
| 70 | 51 | a3 | 40 | 8f | 92 | 9d | 38 | f5 | bc | b6 | da | 21 | 10 | ff | f3 | d2 |
| 80 | cd | 0c | 13 | ec | 5f | 97 | 44 | 17 | c4 | a7 | 7e | 3d | 64 | 5d | 19 | 73 |
| 90 | 60 | 81 | 4f | dc | 22 | 2a | 90 | 88 | 46 | ee | b8 | 14 | de | 5e | 0b | db |
| a0 | e0 | 32 | 3a | 0a | 49 | 06 | 24 | 5c | c2 | d3 | ac | 62 | 91 | 95 | e4 | 79 |
| b0 | e7 | c8 | 37 | 6d | 8d | d5 | 4e | a9 | 6c | 56 | f4 | ea | 65 | 7a | ae | 08 |
| c0 | ba | 78 | 25 | 2e | 1c | a6 | b4 | c6 | e8 | dd | 74 | 1f | 4b | bd | 8b | 8a |
| d0 | 70 | 3e | b5 | 66 | 48 | 03 | f6 | 0e | 61 | 35 | 57 | b9 | 86 | c1 | 1d | 9e |
| e0 | e1 | f8 | 98 | 11 | 69 | d9 | 8e | 94 | 9b | 1e | 87 | e9 | ce | 55 | 28 | df |
| f0 | 8c | a1 | 89 | 0d | bf | e6 | 42 | 68 | 41 | 99 | 2d | 0f | b0 | 54 | bb | 16 |

Abbildung 1 - S-Box

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | a | b | c | d | e | f |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 00 | 52 | 09 | 6a | d5 | 30 | 36 | a5 | 38 | bf | 40 | a3 | 9e | 81 | f3 | d7 | fb |
| 10 | 7c | e3 | 39 | 82 | 9b | 2f | ff | 87 | 34 | 8e | 43 | 44 | c4 | de | e9 | cb |
| 20 | 54 | 7b | 94 | 32 | a6 | c2 | 23 | 3d | ee | 4c | 95 | 0b | 42 | fa | c3 | 4e |
| 30 | 08 | 2e | a1 | 66 | 28 | d9 | 24 | b2 | 76 | 5b | a2 | 49 | 6d | 8b | d1 | 25 |
| 40 | 72 | f8 | f6 | 64 | 86 | 68 | 98 | 16 | d4 | a4 | 5c | cc | 5d | 65 | b6 | 92 |
| 50 | 6c | 70 | 48 | 50 | fd | ed | b9 | da | 5e | 15 | 46 | 57 | a7 | 8d | 9d | 84 |
| 60 | 90 | d8 | ab | 00 | 8c | bc | d3 | 0a | f7 | e4 | 58 | 05 | b8 | b3 | 45 | 06 |
| 70 | d0 | 2c | 1e | 8f | ca | 3f | 0f | 02 | c1 | af | bd | 03 | 01 | 13 | 8a | 6b |
| 80 | 3a | 91 | 11 | 41 | 4f | 67 | dc | ea | 97 | f2 | cf | ce | f0 | b4 | e6 | 73 |
| 90 | 96 | ac | 74 | 22 | e7 | ad | 35 | 85 | e2 | f9 | 37 | e8 | 1c | 75 | df | 6e |
| a0 | 47 | f1 | 1a | 71 | 1d | 29 | c5 | 89 | 6f | b7 | 62 | 0e | aa | 18 | be | 1b |
| b0 | fc | 56 | 3e | 4b | c6 | d2 | 79 | 20 | 9a | db | c0 | fe | 78 | cd | 5a | f4 |
| c0 | 1f | dd | a8 | 33 | 88 | 07 | c7 | 31 | b1 | 12 | 10 | 59 | 27 | 80 | ec | 5f |
| d0 | 60 | 51 | 7f | a9 | 19 | b5 | 4a | 0d | 2d | e5 | 7a | 9f | 93 | c9 | 9c | ef |
| e0 | a0 | e0 | 3b | 4d | ae | 2a | f5 | b0 | c8 | eb | bb | 3c | 83 | 53 | 99 | 61 |
| f0 | 17 | 2b | 04 | 7e | ba | 77 | d6 | 26 | e1 | 69 | 14 | 63 | 55 | 21 | 0c | 7d |

Abbildung 2 - inverse S-Box

# Inverse Transformationsfunktionen



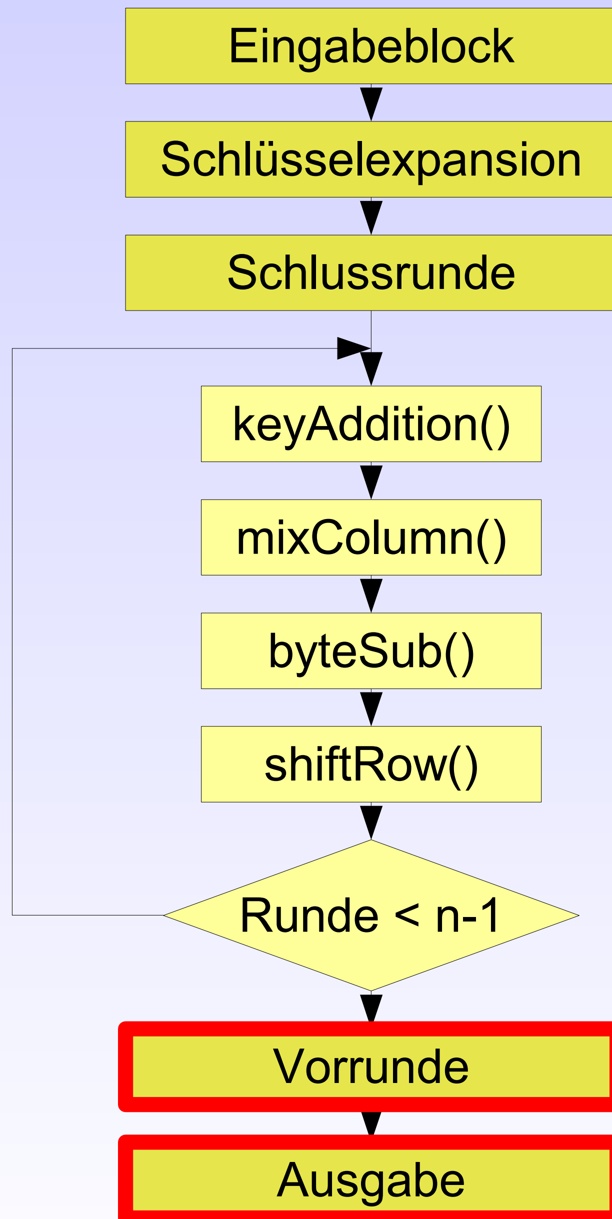

In der inversen Funktion shiftRow() werden analog zur Funktion shiftRow() die Zeilen eines Blocks in Abhängigkeit zu $N_b$ verschoben.

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

Abbildung 3 – Shift-Tabelle
Quelle: [DRRV_99], Kapitel 4.2.2, Seite 12

Jetzt allerdings werden die Zeilen nicht nach links, sondern zurück nach rechts verschoben. Die Regel, das die erste Zeile nicht verschoben wird, bleibt nach wie vor bestehen!

# Entschlüsselung




Im letzten Schritt der Entschlüsselung erfolgt die Durchführung der Vorrunde. Hier wird die Funktion keyAddition() ein letztes Mal aufgerufen, um eine XOR Verknüpfung zwischen den Blockdaten und dem Rundenschlüssel bitweise vorzunehmen.

Nach der Vorrunde lassen sich die einzelnen dechiffrierten Blöcke nun wieder zu einer Datei zusammenfassen.

# Design of AES

Was bewirken die einzelnen Verschlüsselungsschritte ?

**byteSub()**
Die Substitution ist eine nicht-lineare Transformation, in der Blockeinträge mit Einträgen in der S-Box ersetzt werden. Diese Funktion ist eine stark nicht-lineare Operation, die dem AES nahezu einen idealen Schutz vor linearer und differentieller Kryptoanalyse bietet, wie zum Beispiel die „Known Plaintext“ Attacke.

**shiftRow()**
Bei shiftRow() handelt es sich um eine lineare Transformation, bei der Zeilen des Blocks um i Zellen verschoben werden. Dieser Vorgang sorgt für eine große Diffusion, die wiederum notwendig ist, um Regelmäßigkeiten in den Blöcken um meherere Zeichen hinweg zu verteilen, so dass diese schwieriger aufzufinden sind. In unserer 2-Dimensionalen Darstellung der Daten in einer Tabelle, ist diese Funktion zur Verschiebung der Zellen auf der x-Achse zuständig.

**mixColumn()**
Diese Funktion führt wie shiftRow() eine lineare Transformation der Daten aus, bei der jedoch die einzelnen Zellen der Spalten durchmischt werden, indem diese mit einer Matrix multiplizeirt werden und anschliessend mit dem vorhergehenden Ergebnis XORiert werden. Auch diese Funktion sorgt für bessere Diffusion, um Regelmäßigkeiten schwerer erkennbar zu machen. MixColumn() arbeitet nach unserem Abbild auf der y-Achse.

# Design of AES

**keyAddition()**
Hier werden die Zellen des Blocks mit dem Schlüssel XORiert um eine Erhöhung der Konfusion zu erreichen. Nicht nur innerhalb der Runden wird jedesmal eine keyAddition() ausgeführt, sondern auch noch vor den Rundendurchläufen, um der schon gerade genannten „Known-Paintext“ Attacke entgegenzuwirken.

# Erklärungen

**Differentielle Kryptoanalyse**
Es handelt sich bei der Differentiellen Kryptoanalyse um einen sogenannten Angriff mit gewähltem Klartext, das heißt, der Angreifer konnte sich zu selbst gewählten Klartexten die Geheimtexte beschaffen. Bei der differenziellen Kryptoanalyse benutzt man Klartextpaare, die eine bestimmte feste Differenz aufweisen und hofft, in den passenden Kryptotexten ein nichtzufälliges Muster zu entdecken, das einen Rückschluss auf den Schlüssel beziehungsweise Teile davon zulässt.

**Lineare Kryptoanalyse**
Die lineare Kryptoanalyse basiert auf statistischen linearen Relationen zwischen Klartext, Geheimtext und Schlüssel. Der Kryptoanalyst erhält beliebig viele (gestohlene) Klartexte und die zugehörigen, mit dem unbekannten Schlüssel chiffrierten (abgefangenen) Geheimtexte. Er kann aber die internen Abläufe, die wechselnden Registerinhalte etc., nicht beobachten.

**Konfusion**
Unter Konfusion versteht man die Verschleierung des Zusammenhangs zwischen Klartextzeichen und Geheimtextzeichen. Nach Claude Shannon gehört die Konfusion zusammen mit der Diffusion zu den zwei Grundprinzipien der Chiffrierung.

## Weblinks mit Informationen zur Sicherheit

- http://www.heise.de/security/artikel/61803
- http://www.cipherbox.de/

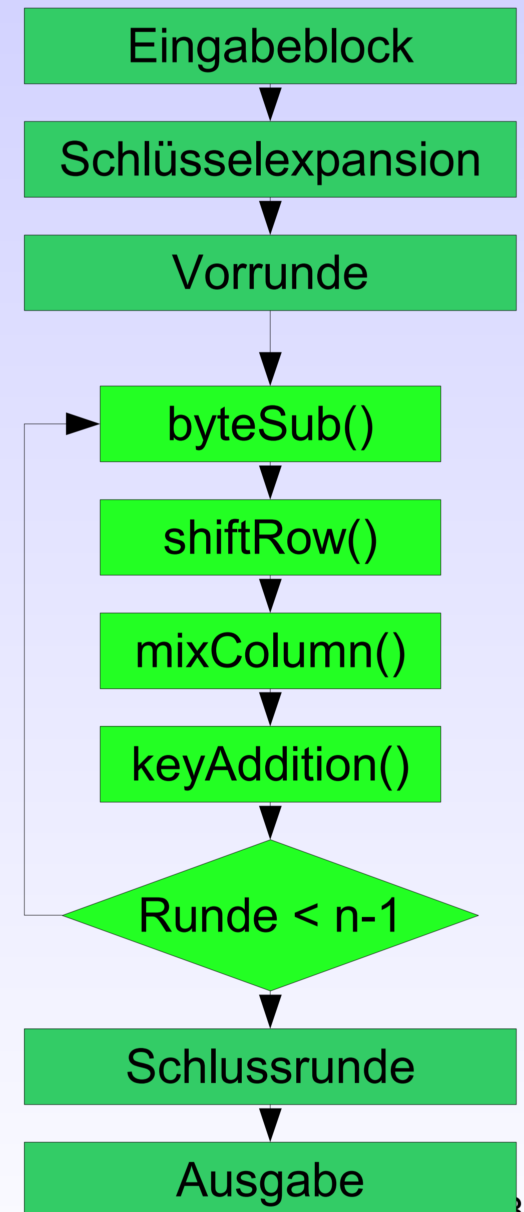
## Asymptotische Laufzeit Analyse

Die Laufzeit des Rijndael Algorithmus ergibt sich aus den Laufzeiten der einzelnen Teiloperationen des Algorithmus.

Die Operationen byteSub(), shiftRow(), mixColumn() und keyAddition() werden dabei 10 bis 14 mal durchlaufen, abhängig von der Schlüssellänge.
D.h. die Laufzeit ist auch geringfügig von der Schlüssellänge abhängig, ebenso wie von der verwendeten Blockgröße.
Ein Block kann aus 4 bis 8 Spalten bestehen.

Dies berücksichtigen wir hier aber nicht da es sich bei der Rundenzahl ebenso wie bei der Blockgröße um Konstante Werte handelt.

Im ingesamten ergibt sich daraus eine Laufzeit von **O(n)**.



## Asymptotische Laufzeit Analyse der Schlüsselexpansion

Die expandierte Schlüssel (W) besteht aus einem linearen 4-byte Wortarray mit der Größe von $N_b * (N_r +1)$. Die ersten $N_k$ Worte in dem Array beinhalten den Originalschlüssel; alle anderen folgenden Schlüssel werden rekursiv in Abhängigkeit des vorherigen Schlüssels erzeugt. Beim Expandieren des Schlüssels werden drei Schritte durchgeführt: eine Substitution und zwei XOR Verknüpfungen (die erste mit einem bestehende Wert, die zweite mit einem Wert aus der rcon-Tabelle). Alle Schritte werden in konstanter Zeit ausgeführt.
Die entsprechenden Rundenschlüssel, die für jede einzelne Runde zum Verschlüsseln der Blockdaten benötigt werden, müssen die gleiche Größe ausweisen wie die Blockgröße der zu verschlüsselnden Blockdaten.
Da die notwendigen Schlüssel aus dem linearen Array $W = (N_b * (N_r +1))$ entnommen werden und alle Expansionsschritte in konstanter Zeit erfolgen, beträgt die Laufzeit: $O(N_b * N_r)$

## Asymptotische Laufzeit Analyse der Funktion byteSub()

- Die Funktion byteSub() besteht aus zwei Operationen,

  1. zum einen aus der Berechnung des **multiplikativen inversen Elements** ($x := x^{-1}$) im endlichen Feld $GF(2^8)$ ($x^{-1} \bmod x^8+x^4+x^3+x+1$)

  2. und zum anderen aus einer **affinen Transformation**, ebenfalls über dem endlichen Feld $GF(2^8)$.

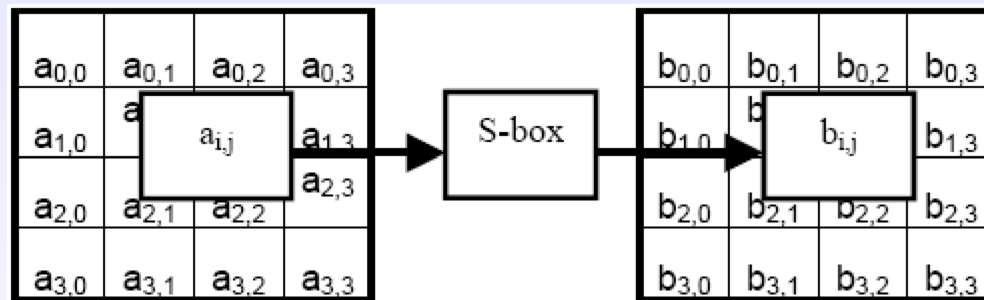
$$\begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \\ y_6 \\ y_7 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\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

affine transformation über $GF(2^8)$

# Laufzeitanalyse

## Asymptotische Laufzeit Analyse der Funktion byteSub()

- Ergebnisse beider Operationen liegen im gleichen Zahlenraum wie Ursprünglich zu tranformierender Operand
- Kann über Tabelle oder Matrix durch austauschen (Substitution) realisiert werden
- Substitution aller Werte eines Blocks geschieht also in Konstanter Zeit.



# Laufzeitanalyse

## Asymptotische Laufzeit Analyse der Funktion shiftRow()

- Verschieben aller Wert einer Zeile um einen bestimmten Wert abhängig von der Blockgröße

- Jedes n wird i konstanter Laufzeit verschoben, somit ergibt sich eine Laufzeit von **O(n)**, oder besser gesagt eine Konstante Laufzeit für das Verschieben eines Blocks.

| | Row 0 | Row 1 | Row 2 | Row 3 |
|---|---|---|---|---|
| Nb = 4 | 0 | 1 | 2 | 3 |
| Nb = 6 | 0 | 1 | 2 | 3 |
| Nb = 8 | 0 | 1 | 3 | 4 |

## Asymptotische Laufzeit Analyse der Funktion mixColumn()

- Behandelt eine Spalte wie einen Vector und multipliziert jede Spalte mit dem Polynom '03'$x^3$+'01'$x^2$+'01'x+'02'. Dies läßt sich auch als Matrizenmultiplikation darstellen.

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

- Die Teilergebnisse der Matrizenmultiplikation (Beispiel: $a_0$ * 02) werden noch untereinander XORiert.
- Verwendete Matrix und Anzahl der XOR - Operationen, Konstant, da eine Spalte immer 4 Zeilen hat
- Somit benötigt diese Funktion eine Konstante Laufzeit für das Vermischen der Spalten eines Blocks.

## Asymptotische Laufzeit Analyse der Funktion keyAddition()

- XORiert den Rundenschlüssel mit dem zu verschlüsselnden Block

$$\begin{bmatrix} a_{0,0} & a_{0,1} & a_{0,2} & a_{0,3} & a_{0,4} & a_{0,5} \\ a_{1,0} & a_{1,1} & a_{1,2} & a_{1,3} & a_{1,4} & a_{1,5} \\ a_{2,0} & a_{2,1} & a_{2,2} & a_{2,3} & a_{2,4} & a_{2,5} \\ a_{3,0} & a_{3,1} & a_{3,2} & a_{3,3} & a_{3,4} & a_{3,5} \end{bmatrix} \oplus \begin{bmatrix} k_{0,0} & k_{0,1} & k_{0,2} & k_{0,3} & k_{0,4} & k_{0,5} \\ k_{1,0} & k_{1,1} & k_{1,2} & k_{1,3} & k_{1,4} & k_{1,5} \\ k_{2,0} & k_{2,1} & k_{2,2} & k_{2,3} & k_{2,4} & k_{2,5} \\ k_{3,0} & k_{3,1} & k_{3,2} & k_{3,3} & k_{3,4} & k_{3,5} \end{bmatrix} = \begin{bmatrix} b_{0,0} & b_{0,1} & b_{0,2} & b_{0,3} & b_{0,4} & b_{0,5} \\ b_{1,0} & b_{1,1} & b_{1,2} & b_{1,3} & b_{1,4} & b_{1,5} \\ b_{2,0} & b_{2,1} & b_{2,2} & b_{2,3} & b_{2,4} & b_{2,5} \\ b_{3,0} & b_{3,1} & b_{3,2} & b_{3,3} & b_{3,4} & b_{3,5} \end{bmatrix}$$

Beispiel: $a_{0,0} * k_{0,0} = b_{0,0}$

- Verschlüsselt einen Blocks in einer konstanten Zeit

# Laufzeitanalyse

## Asymptotische Laufzeit Analyse

- Schlußrunde ist gleich einer normalen Runde, ohne die Funktion mixColumn()
- Vorrunde besteht nur aus einer keyAddition()
- Kompletter Algorithmus besteht aus keyExpansion(), byteSub(), shiftRow(), mixColumn() und keyAddition()
- Alle Funktionen werden in konstanter Zeit durchgeführt
- Jedes Zeichen wird mindestens einmal ausgetauscht oder XORiert
- Dadurch ergibt sich eine Gesamtlaufzeit von **O(n)**.