Security

- LV 4121 und 4241 -

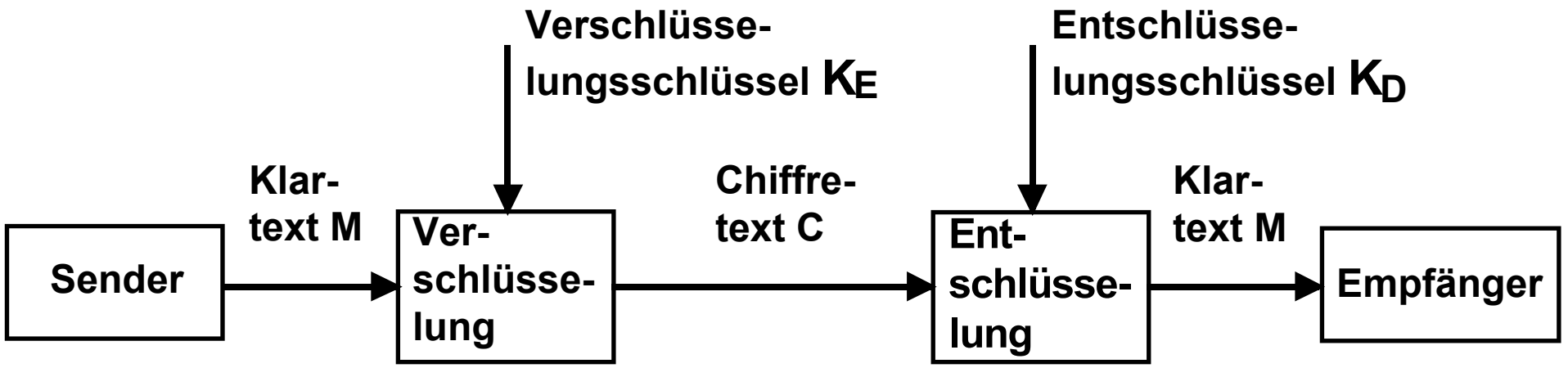
Symmetrische Verfahren und moderne Blockchiffren

- Gegenüberstellung symmetrische und asymmetrische Kryptoverfahren
- Symmetrische Blockverschlüsselung und **DES-Algorithmus**
- Advanced Encryption Standard **(AES)**
- **Betriebsarten** für blockorientierte Verschlüsselungsalgorithmen
- Symmetrische Bitstromverschlüsselung (one time pad)
- Grundlegende Aspekte des Schlüssel- und Sicherheitsmanagements
- **DH-Schlüsselaustausch** (gegenseitige Schlüsselabsprache)
- Schlüsselhierarchie und Schlüsselklassen

Kap. 4: Symmetrische Verfahren und moderne Blockchiffren

Teil 1: Symmetrische Blockverschlüsselung

- Schlüsselgesteuerte Transformation
- Gegenüberstellung der Chiffrierverfahren
- Data Encryption Standard (DES-Algorithmus)
- Advanced Encryption Standard (AES-Algorithmus)


Verschlüsse-
lungsschlüssel K_E
Entschlüsse-
lungsschlüssel K_D
Klar-
text M
Chiffre-
text C
Klar-
text M
Sender
Ver-
schlüsse-
lung
Ent-
schlüsse-
lung
Empfänger

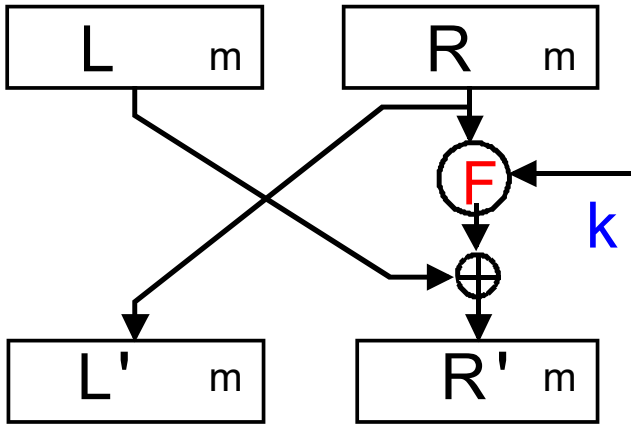
Symmetrische vs. asymmetrische Chiffrierverfahren:

Symmetrische Verfahren	Asymmetrische Verfahren
Vorteile: • Sie sind schnell, d. h. sie haben einen hohen Datendurchsatz. • Die Sicherheit ist im wesentlichen durch die Schlüssellänge festgelegt.	**Vorteile:** • Jeder Teilnehmer muß nur seinen eigenen privaten Schlüssel geheimhalten. • Sie bieten elegante Lösungen für die Schlüsselverteilung in Netzen.
Nachteile: • Jeder Teilnehmer muß sämtliche Schlüssel seiner Kommunikationspartner geheimhalten. • Es ist ein komplexeres Schlüsselmanagement erforderlich.	**Nachteile:** • Sie sind langsam, d. h. sie haben im allgemeinen einen deutlich geringeren Datendurchsatz als symmetrische Verfahren. • Es gibt wesentlich bessere Attacken als das Durchprobieren aller Schlüssel.

Idee einer Feistel-Chiffre

Transformation und Rekonstruktion:

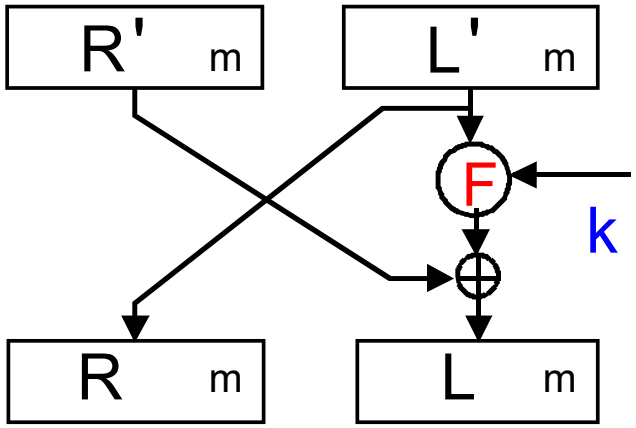
Verschlüsselung



Transformation der Eingangsblöcke L und R in Ausgangsblöcke L' und R', wobei $k \in K$ der Schlüssel ist.

$L' = R$ und $R' = F(R, k) \oplus L$

Entschlüsselung



Rekonstruktion der Eingangsblöcke L und R aus den Ausgangsblöcken L' und R', wobei $k \in K$ der Schlüssel ist.

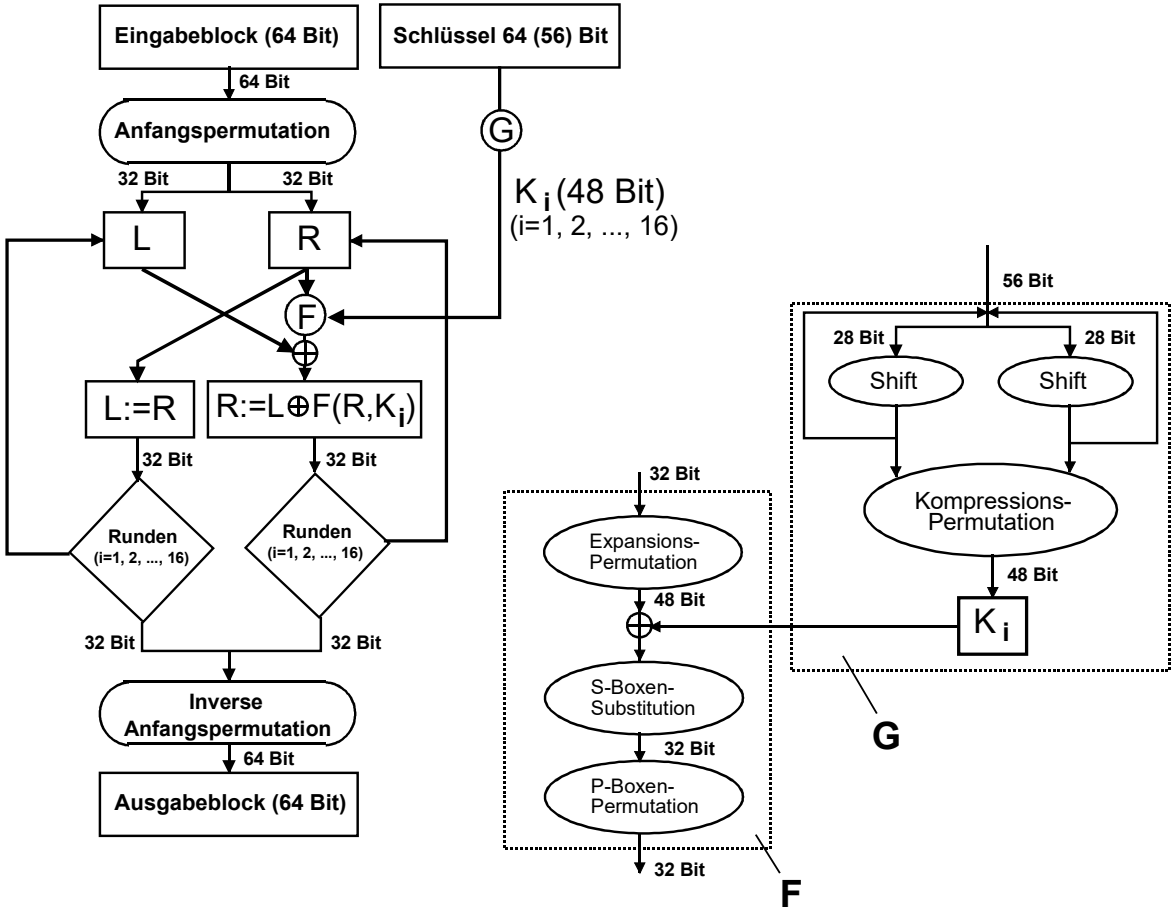
$R = L'$ und $L = F(L', k) \oplus R'$

Idee einer Feistel-Chiffre

- Eines der bedeutesten Hilfsmittel für den Entwurf heutiger Blockchiffren ist das von Horst Feistel entwickelte Konstruktionsprinzip einer **Feistel-Chiffre**.
- Der wesentliche Aspekt besteht darin, dass eine beliebige Funktion $F : \{0, 1\}^m \times k \rightarrow \{0, 1\}^m$ zum Einsatz kommen kann. Die Funktion F muss nicht einmal umkehrbar sein.
- Soll eine Feistel-Chiffre sowohl zur Ver- als auch Entschlüsselung verwendet werden, so ist es notwendig, die beiden Ausgabeblöcke zu vertauschen (vgl. DES).
- Wendet man das Konstruktionsprinzip wiederholt auf die sich ergebenden Ausgangsblöcke an, so können **sichere Verschlüsselungssysteme** konstruiert werden.

Data Encryption Standard

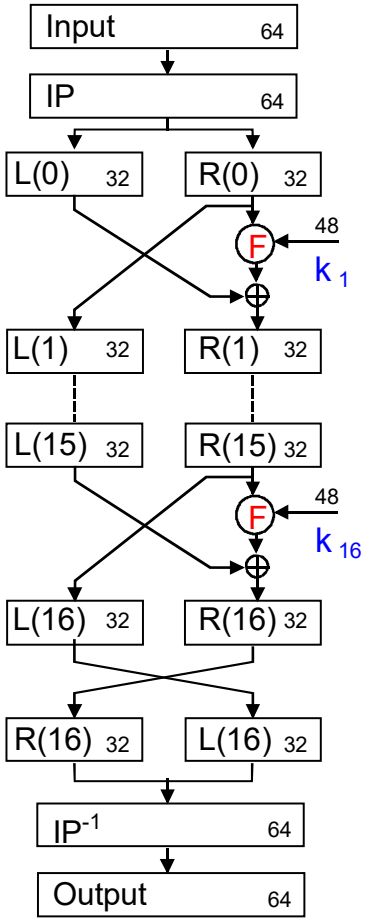
Prinzipieller Aufbau



DES:

- 1974 veröffentlicht
- ANSI-Standard (USA)
- Blocklänge 64 Bit
- Schlüssellänge 56 Bit
- ca. $7{,}2 \cdot 10^{16}$ Schlüssel
- Anfangspermutation
- Zwei Hälften L und R
- 16 Runden Iteration
- Rundenschlüssel 48 Bit
- Substitution (nichtlinear)
- Transposition
- Ver-/ Entschlüsselung
- Abschlusspermutation

Data Encryption Standard



Initialpermutation

1. Runde
Output: L(1) und R(1)

.
.
.

16. Runde
Output: L(16) und R(16)

Preoutput

Abschlusspermutation

DES:

- 16 Teilschlüssel k_1 bis k_{16} aus 56-Bit-Schlüssel mittels **Schlüsselauswahl-funktion**
- Algorithmus so ausgelegt, dass er für Ver- und Entschlüsselung identisch ist
- Verschlüsselung $k_1, k_2, ..., k_{16}$ und Entschlüsselung $k_{16}, k_{15}, ..., k_1$
- IP^{-1} zu IP inverse Permutation sind beide ohne Sicherheitsbedeutung
- $IP(X) = (x_{58}, x_{50}, x_{42}, \ldots, x_{15}, x_7)$
- $IP^{-1}(Y) = (y_{40}, y_8, y_{48}, \ldots, y_{57}, y_{25})$

Initialpermutation IP

58	50	42	34	26	18	10	2
60	52	44	36	28	20	12	4
62	54	46	38	30	22	14	6
64	56	48	40	32	24	16	8
57	49	41	33	25	17	9	1
59	51	43	35	27	19	11	3
61	53	45	37	29	21	13	5
63	35	47	39	31	23	15	7

heißt:
schreibe 1. Bit an Position 58 und
Bit 2 an Position 50.

Abschlusspermutation IP^{-1}

40	8	48	16	56	24	64	32
39	7	47	15	55	23	63	31
38	6	46	14	54	22	62	30
37	5	45	13	53	21	61	29
36	4	44	12	52	20	60	28
35	3	43	11	51	19	59	27
34	2	42	10	50	18	58	26
33	1	41	9	49	17	57	25

entsprechend:
schreibe 58. Bit an Position 1 bzw.
Bit 50 an Position 2 zurück.

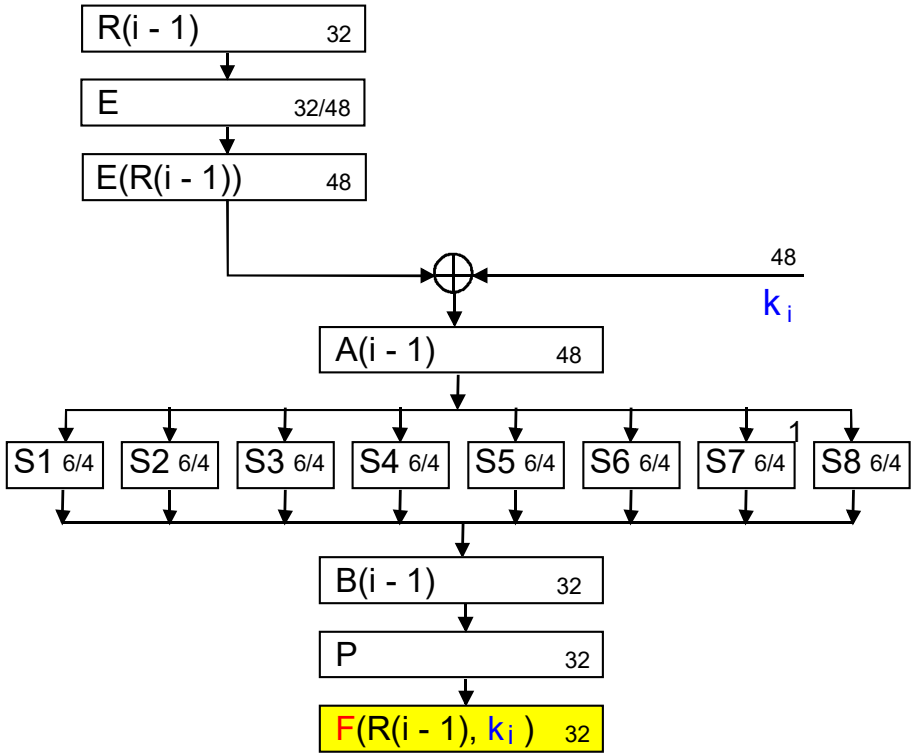
Iteration der Verschlüsselung:

Im Anschluss an die **Initialpermutation IP** wird der **Block IP(X)** in einen linken **Block L** und einen rechten **Block R** zerlegt (L || R) = IP(X). Beide Blöcke sind 32 Bit lang. Auf **L** = (x_{58}, x_{50}, …, x_8) und **R** = (x_{57}, x_{49}, …, x_7) wird folgende Operation angewandt:

```
L(0) := L ;
R(0) := R ;
for i := 1 to 16 do
    begin
        L(i) := R(i - 1) ;
        R(i) := L(i - 1) ⊕ F(R(i - 1), k_i) ;
    end ;
```

Die Funktion F(R(i - 1), k_i):

Kern des gesamten Verfahrens $F : \{0, 1\}^{32} \times \{0, 1\}^{48} \rightarrow \{0, 1\}^{32}$.



Input: R(i - 1)

Expansionsabbildung E

i. Rundenschlüssel k_i

$A(i - 1) = E(R(i - 1)) \oplus k_i$
$:= (A_1, A_2, ..., A_8)$

Sicherheitsboxen Si

Permutation P

i. Runde
Output: $F(R(i -1), k_i)$

Expansionsabbildung E

32	1	2	3	4	5
4	5	6	7	8	9
8	9	10	11	12	13
12	13	14	15	16	17
16	17	18	19	20	21
20	21	22	23	24	25
24	25	26	27	28	29
28	29	30	31	32	1

heißt:

$R(i-1) = (r_1, r_2, r_3, \ldots, r_{31}, r_{32}) \Rightarrow$

$E(R(i-1)) = (r_{32}, r_1, r_2, \ldots, r_{32}, r_1)$

Indextabelle der Permutation P

16	7	20	21
29	12	28	17
1	15	23	26
5	18	31	10
2	8	24	14
32	27	3	9
19	13	30	6
22	11	4	25

entsprechend:

$B(i-1) = (b_1, b_2, b_3, \ldots, b_{32}) \Rightarrow$

$P(B(i-1)) = (b_{16}, b_7, b_{20}, \ldots, b_{25})$

Die S-Box S1:

- Das Ergebnis der bitweisen XOR-Bildung **A** bildet den Input für die **S-Boxen** S1 bis S8, wobei A_j der zu Sj gehörende Input ist.
- Jede S-Box kann vier unterschiedliche Substitutionen realisieren, wobei die einzelnen S-Boxen durch eine 4 x 16-Matrix festgelegt sind.
- Die Zeilen werden mit **0**, **1**, **2**, **3** und die Spalten mit **0**, **1**, ..., **14**, **15** bezeichnet (im Beispiel ist die S-Box S1 widergegeben).

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

Die S-Box S1:

- Jede Zeile der S-Box Sj stellt eine Substitution $S_{j,k} : \{0, 1\}^4 \rightarrow \{0, 1\}^4$ dar, wobei k die Zeilennummer bezeichnet.
- Auf diese Weise können mit acht S-Boxen insgesamt 32 verschiedene Substitutionen realisiert werden.
- Ist $A_j = (a_{j1}, a_{j2}, a_{j3}, a_{j4}, a_{j5}, a_{j6})$ ein 6-Bit-Block.
  Dann bestimmen die Bits $a_{j1}a_{j6}$ – als Binärzahl gelesen – die Zeilennummer und $a_{j2}a_{j3}a_{j4}a_{j5}$ – ebenfalls als Binärzahl aufgefasst – die Spaltennummer der S-Box Sj.
- Der zugehörige Matrixeintrag $S_{j,aj1aj6}$ ($a_{j2}a_{j3}a_{j4}a_{j5}$) legt das Substitutionsergebnis eindeutig fest, wobei jeder Eintrag als Dezimalzahl eine Folge von **4 Bit** ergibt (→ Output der S-Box Sj).

Die DES Substitutionsboxen:

S1:

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

S2:

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

Die DES Substitutionsboxen:

S3:

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

S4:

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

Die DES Substitutionsboxen:

S5:

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

S6:

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

Die DES Substitutionsboxen:

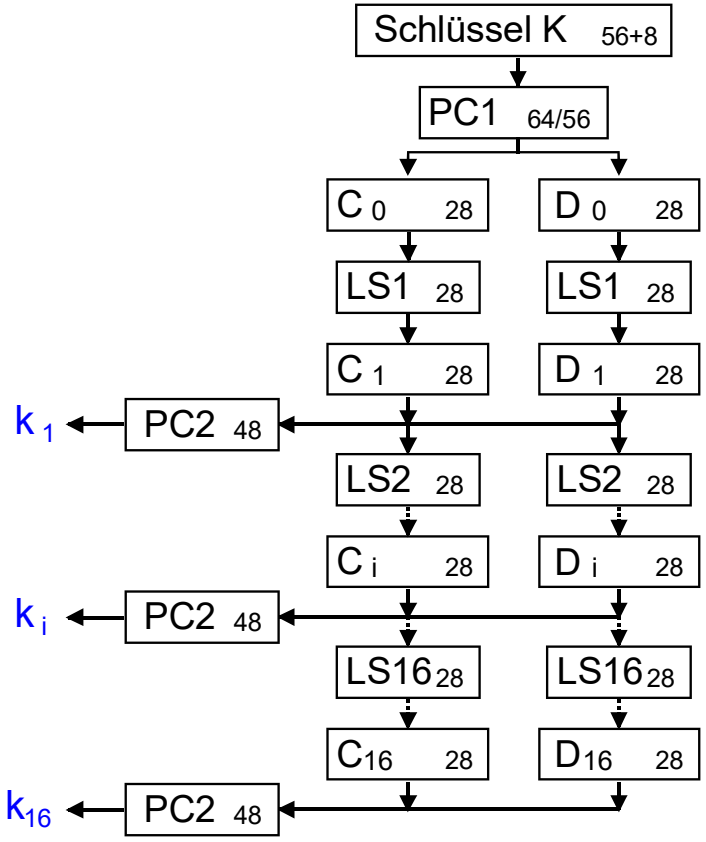
S7:

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

S8:

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

Die Schlüsselauswahlfunktion:



Gegeben: $K = (k_1, k_2, ..., k_{64})$

Permuted Choice 1
(Schlüsselreduktion und Permutation)

Zirkuläre Linksshifts

Permuted Choice 2
1. Rundenschlüssel (Konkatenation)

.
.
.

i. Rundenschlüssel (Konkatenation)

.
.
.

16. Rundenschlüssel (Konkatenation)

Die Schlüsselauswahlfunktion:

- Die bei jeder Iteration benutzten **Rundenschlüssel k_i** werden aus dem gegebenen **Schlüssel K** ermittelt.
- Aus Sicherheitsgründen sollten alle Rundenschlüssel k_i verschieden sein (unterschiedliche Teilmengen aus **K**).
- Gegeben sei der vorgegebene Schlüssel **K** = (k_1, k_2, ..., k_{64}) mit den **Paritätsbits k_i** an den Stellen **i** = 8(8)64.
- PC1 entfernt alle Paritätsbits und reduziert den Schlüssel **K** auf 56 **aktive** Schlüsselbits, die zudem permutiert werden.
- Der resultierende Wert PC1(**K**) ergibt sich zu (k_{57}, k_{49}, ..., k_{12}, k_4).

Permutation PC1

57	49	41	33	25	17	9
1	58	50	42	34	26	18
10	2	59	51	43	35	27
19	11	3	60	52	44	36
63	55	47	39	31	23	15
7	62	54	46	38	30	22
14	6	61	53	45	37	29
21	13	5	28	20	12	4

heißt:

$PC1(\mathbf{K}) = (k_{57}, k_{49}, k_{41}, \ldots, k_{12}, k_4)$

$\Rightarrow C_0 = (k_{57}, k_{49}, \ldots, k_{36})$ und

$\Rightarrow D_0 = (k_{63}, k_{55}, \ldots, k_4)$

Permutation PC2

14	17	11	24	1	5
3	28	15	6	21	10
23	19	12	4	26	8
16	7	27	20	13	2
41	52	31	37	47	55
30	40	51	45	33	48
44	49	39	56	34	53
46	42	50	36	29	32

entsprechend:

$C_1 = (k_{49}, k_{41}, \ldots, k_{44}, k_{36}, k_{57})$ und

$D_1 = (k_{55}, k_{47}, \ldots, k_{12}, k_4, k_{63}) \Rightarrow$

$k_1 = (k_{10}, k_{51}, \ldots, k_{13}, k_{62}, k_{55}, k_{31})$

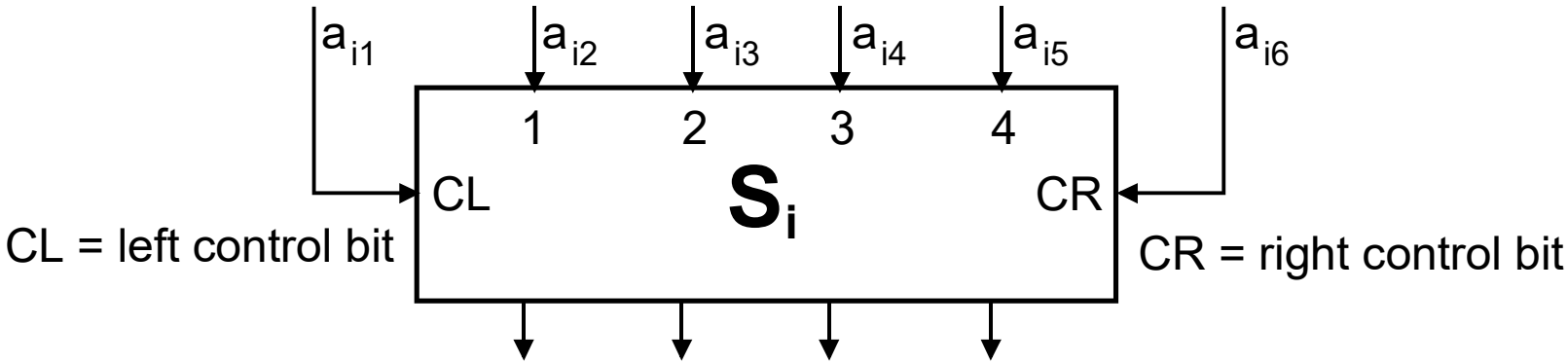
- Der permutierte Wert PC1(**K**) wird in eine linke Hälfte $C_0 = (k_{57}, k_{49}, ..., k_{36})$ und eine rechte Hälfte $D_0 = (k_{63}, k_{55}, ..., k_4)$ aufgeteilt.
- Die Vektoren $C_i \,||\, D_i$ (Konkatenation) werden für i = 1, 2, ..., 16 rekursiv aus $C_{i-1} \,||\, D_{i-1}$ durch **zirkuläres Linksshiften** LSi der Hälften C_{i-1} und D_{i-1} um eine oder zwei Bitpositionen abgeleitet.
- Die Hälften werden dabei getrennt geshiftet. $C_1 = LS1(C_0)$ bzw. $D_1 = LS1(D_0)$ um eine Position zirkulär nach links.

Nummer der Iteration	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Anzahl der Linksshifts	1	1	2	2	2	2	2	2	1	2	2	2	2	2	2	1

- Damit ergeben sich beispielsweise die beiden Hälften:
  $C_1 = (k_{49}, k_{41}, ..., k_{36}, k_{57})$ und $D_1 = (k_{55}, k_{47}, ..., k_4, k_{63})$ aufgeteilt.
- PC2 bestimmt schließlich für i = 1, 2, ..., 16 aus den Konkatenationen $C_i \,||\, D_i$ den Rundenschlüssel k_i.
- Hierzu werden zuerst die Bits von $C_i \,||\, D_i$ auf den Positionen 9, 18, 22, 25, 35, 38, 43 und 54 entfernt.
- Die verbleibenden 48 Bits werden abschließend der Permutation PC2 unterworfen.

- Der **DES** wurde von der Firma IBM entwickelt und auf Empfehlung des National Bureau of Standards, Washington D. C., 1977 genormt.
- Seine offizielle Beschreibung erfährt der DES in **FIPS PUB 46** (Federal Information Processing Standards Publication).
- DES wurde explizit in Übereinstimmung mit den Shannonschen Design-Prinzipien bezüglich Konfusion und Diffusion entwickelt.
- Lokale Diffusion und Konfusion wird durch die im hohen Grad **nicht-lineare** Funktion $F_i = F(R(i - 1), k_i)$ innerhalb jeder Runde i erzeugt, wobei die tatsächliche Nichtlinearität in den **S-Boxen** verankert ist.
- Weitere Diffusion wird durch Transposition bzw. Swapping in zwei Hälften L bzw. R innerhalb jeder Runde (mit Ausnahme der letzten) erzeugt.

Die Substitutionsboxen (S-Boxen) des DES:



- Für jede der vier Kombinationen der beiden Steuerbits CL und CR liefert die S-Box S_i eine unterschiedliche **Permutation** in Abhängigkeit der vier Input-Bits a_{i2} a_{i3} a_{i4} a_{i5} (4-Tuple).
- In dieser **Unterschiedlichkeit** ist die **Nichtlinearität** der S-Boxen und letzten Endes die hohe Sicherheit des DES begründet.

Design-Regeln für die S-Boxen (Empfehlung):

(1) Für jede Kombination der beiden Steuerbits CL und CR sollten die S-Boxen eine über GF(2) **nichtlineare Transormation** von dem Input-4-Tuple in das Output-4-Tuple darstellen.

(2) Eine jegliche Änderung der 6 Input-Bits sollte **mindestens zwei** Output-Bits ändern.

(3) Wenn ein der 6 Input-Bits konstant gehalten wird, dann sollten die sich für die Output-4-Tuples ergebenden $2^5 = 32$ Möglichkeiten eine gute Balance von 0 und 1 erzielen, falls die übrigen 5 Input-Bits variiert werden.

Jedoch sind die tatsächlichen Design-Prinzipien für die S-Boxen des U.S.-Government **nie** veröffentlicht worden (U.S. „classified“ information).

Schwache und semi-schwache Schlüssel:

Definition:

K ist ein schwacher (oder sogenannter self-dualer) Schlüssel, wenn:

$$DES_K(.) = DES^{-1}{}_K(.),$$

d. h. wenn die Verschlüsselungsfunktion für **K** mit der Entschlüsselungsfunktion übereinstimmt, da in diesem Fall die abgeleiteten Teilschlüssel k_i nicht alle voneinander verschieden wären.

- DES hat mindestens vier schwache Schlüssel, die es unbedingt zu vermeiden gilt.
- Es ist sehr wahrscheinlich, dass es außer diesen vier Schlüsseln keine weiteren schwache Schlüssel gibt.

Begründung:

0000 0001	···	0000 0001	0000 0001	···	0000 0001
1111 1110	···	1111 1110	1111 1110	···	1111 1110
1110 0000	···	1110 0000	1111 0001	···	1111 0001
0001 1111	···	0001 1111	0000 1110	···	0000 1111
Byte 1		Byte 4	Byte 5		Byte 8

- Aufgrund der Permutation **PC1** machen diese 4 Schlüssel C_0 entweder zu (00 ··· 0) oder (11 ··· 1) **und** C_1 entweder zu (00 ··· 0) oder (11 ··· 1), so dass folgt: $k_1 = k_2 = \cdots = k_{16}$.
- Hieraus folgt $(k_1, k_2, \cdots, k_{16}) = (k_{16}, k_{15}, \cdots, k_1)$, so dass

$$DES_K(.) = DES^{-1}_K(.).$$

Schwache und semi-schwache Schlüssel:

Definition:

K ist ein semi-schwacher (oder sogenannter dualer) Schlüssel, wenn:

$$DES_{K'}(.) = DES^{-1}_{K}(.),$$

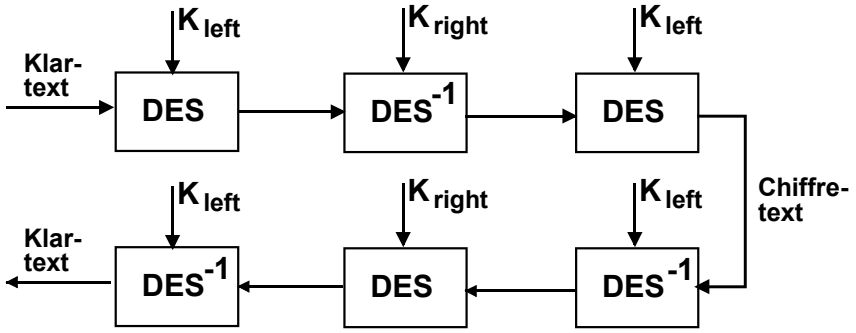
d. h. wenn die Verschlüsselungsfunktion für unterschiedliche Schlüssel **K'** und **K** mit der Entschlüsselungsfunktion übereinstimmt, da in diesem Fall die abgeleiteten Teilschlüssel k_i nicht alle voneinander verschieden wären.

- DES hat mindestens 12 semi-schwache Schlüssel.
- Semi-schwache Schlüssel erscheinen immer paarweise.
- Es ist sehr wahrscheinlich, dass es außer diesen 12 Schlüsseln keine weiteren semi-schwache Schlüssel gibt.

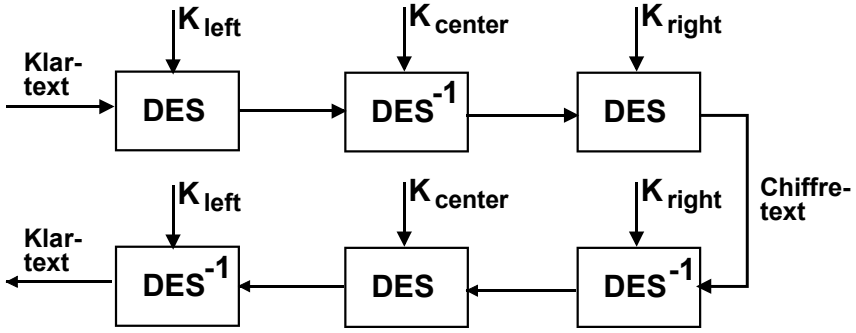
Begründung:

- Ein Schlüssel **K'**, der C_0 = (1010 ··· 10) **und** D_0 entweder zu (00 ··· 0) oder (11 ··· 1) oder (1010 ··· 10) oder (0101 ··· 01) liefert, ist dual zum Schlüssel **K**, der C_0 = (0101 ··· 01) **und** D_0 entweder zu (00 ··· 0) oder (11 ··· 1) oder (0101 ··· 01) oder (1010 ··· 10) ergibt.
- Eine ähnliche Situation ergibt sich für C_0 = (0101 ··· 01) **und** D_0 = (1010 ··· 10) oder D_0 = (0101 ··· 01).
- Hieraus resultieren insgesamt 12 unterschiedliche Fälle.
- Die Hauptschwäche des DES (im sogenannten ECB-Mode betrieben) besteht jedoch in der zu kurzen Schlüssellänge von lediglich 56 Bit.

Triple DES mit **doppelter** Länge K := K_{left} // K_{right}



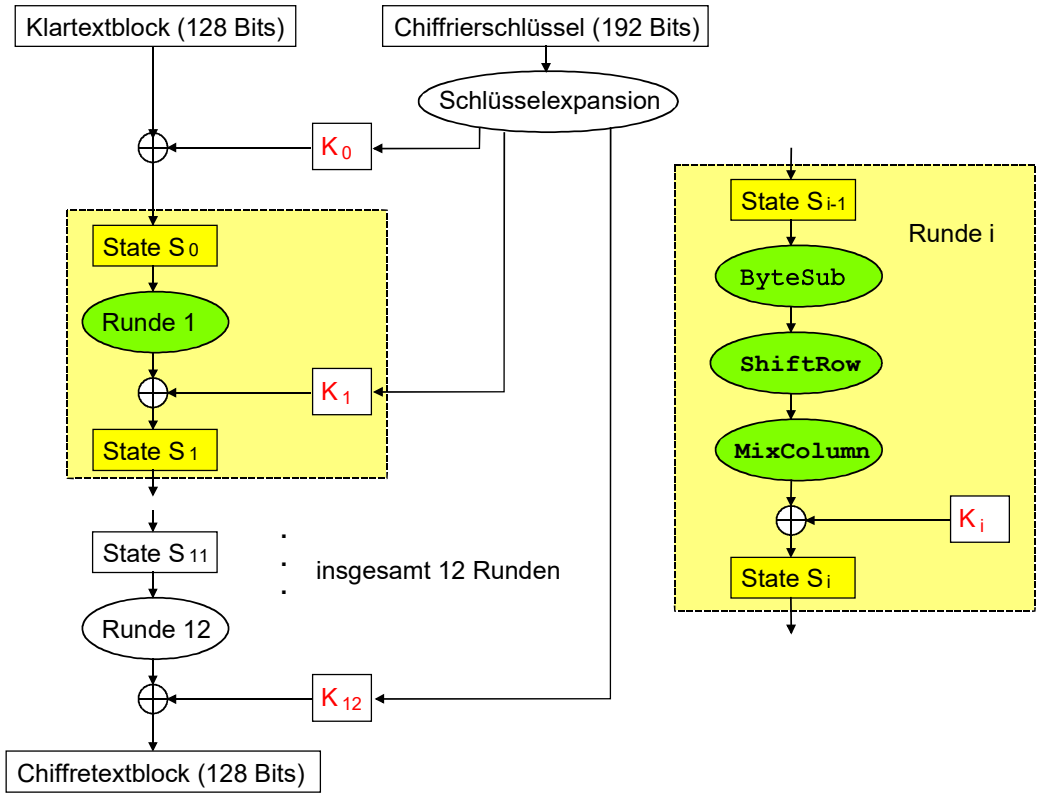
Triple DES mit **dreifacher** Länge K := K_{left} // K_{center} // K_{right}



Anforderungen, Funktionalitäten und Designkriterien:

- Im Jahr 1997 vom U.S.-amerikanischen NIST (National Institut of Standards and Technology) als Nachfolger für DES initiiert.
- Der Algorithmus von AES heißt **Rijndael** und wurde in Belgien von den Kryptologen **Joan Daemen** und **Vincent Rijmen** entwickelt.
- AES arbeitet auf einer Blöckgröße von 128 Bit mit Schlüssellängen von 128, 192 und 256 Bit.
- Spezielle Anforderungen an den Standard betrafen die Sicherheit, Einfachheit, Flexibilität, Effizienz und die Implementierung.
- Im Dezember 2001 wurde AES offiziell zum **FIPS 197** (Federal Information Processing Standards) erklärt.

Schematischer Ablauf:



AES (Rijndael):

- Symmetrische Blockchiffre
- Blocklänge **b** (hier: 128 Bit)
- Schlüssellänge **k**
  (128, 192 oder 256 Bit)
- Variable Rundenzahl **r**
  (zwischen 10 und 14)
- Schlüsselexpansion erzeugt r + 1 **Rundenschlüssel** K_0, K_1, …, K_r
- Zwischenergebnisse des Verschlüsselungsprozesses werden **Zustand** S_i genannt
- Rundenschlüssel haben gleiche Länge wie der jeweilige Zustand

Zusammenhang zwischen r, b und k:

Rundenzahl **r**	Blocklänge		
Schlüssellänge	**b** = 128	**b** = 192	**b** = 256
k = 128	**10**	**12**	**14**
k = 192	**12**	**12**	**14**
k = 256	**14**	**14**	**14**

Zustandsmenge und Schlüssel:

Jedes Element $a_{m,n}$ bzw. $k_{m,n}$ **1 Byte**

Zustände bzw. Klartext (b = 128)

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

Schlüssel (k = 192)

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

Verschlüsselungsprozedur:

- Vor der ersten und nach jeder Runde **i** wird der Rundenschlüssel (hier K_i = 128 Bit) **XOR-verknüpft** mit dem aktuellen Zustand S_i.
- Das Ergebnis dient als Eingabe für die nächste Runde **i** + **1** bzw. als Chiffretext nach der letzten Runde.
- Jede Runde (mit Ausnahme der letzten) besteht aus den Funktionen:

  `ByteSub` → nichtlineare S-Boxen (Substitutionsschritt)

  `ShiftRow` → zyklisches Verschieben der Zustandsmatrix

  `MixColumn` → invertierbare Matrixmultiplikation
- Alle vorgenannten Transformationen (außer XOR-Verknüpfung) sind schlüssel**un**abhängig.

Die `ByteSub`-Transformation:

- Diese Transformation stellt die **nichtlineare S-Box** von AES dar.
- Sie wird auf jedes Byte $a_{m,n}$ des Zustands angewandt und wird als „table-lookup“ implementiert (siehe u. a. Folie Nr. 42).
- Sie entspricht dem Berechnen der **multiplikativen Inversen** in **GF(2^8)** bzw. **mod m(x)** gefolgt von der nachfolgenden affinen Transformation.
- Die einzelnen Multiplikationen und Additionen der Komponenten sind **modulo zwei** zu berechnen.
- Die Umkehrung von **`ByteSub`** erfolgt durch Anwendung der inversen affinen Transformation gefolgt von der multiplikativen Inversen in GF(2^8).

Die `ByteSub`-Transformation (Fortsetzung):

→ affinen Transformation (kommt genau **16 mal** zur Anwendung!)

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

y und **x** jeweils **1 Byte** lang!

Mathematische Beschreibung:

$$\mathbf{y} = A \cdot \mathbf{x} + \mathbf{b}$$

$\rightarrow$

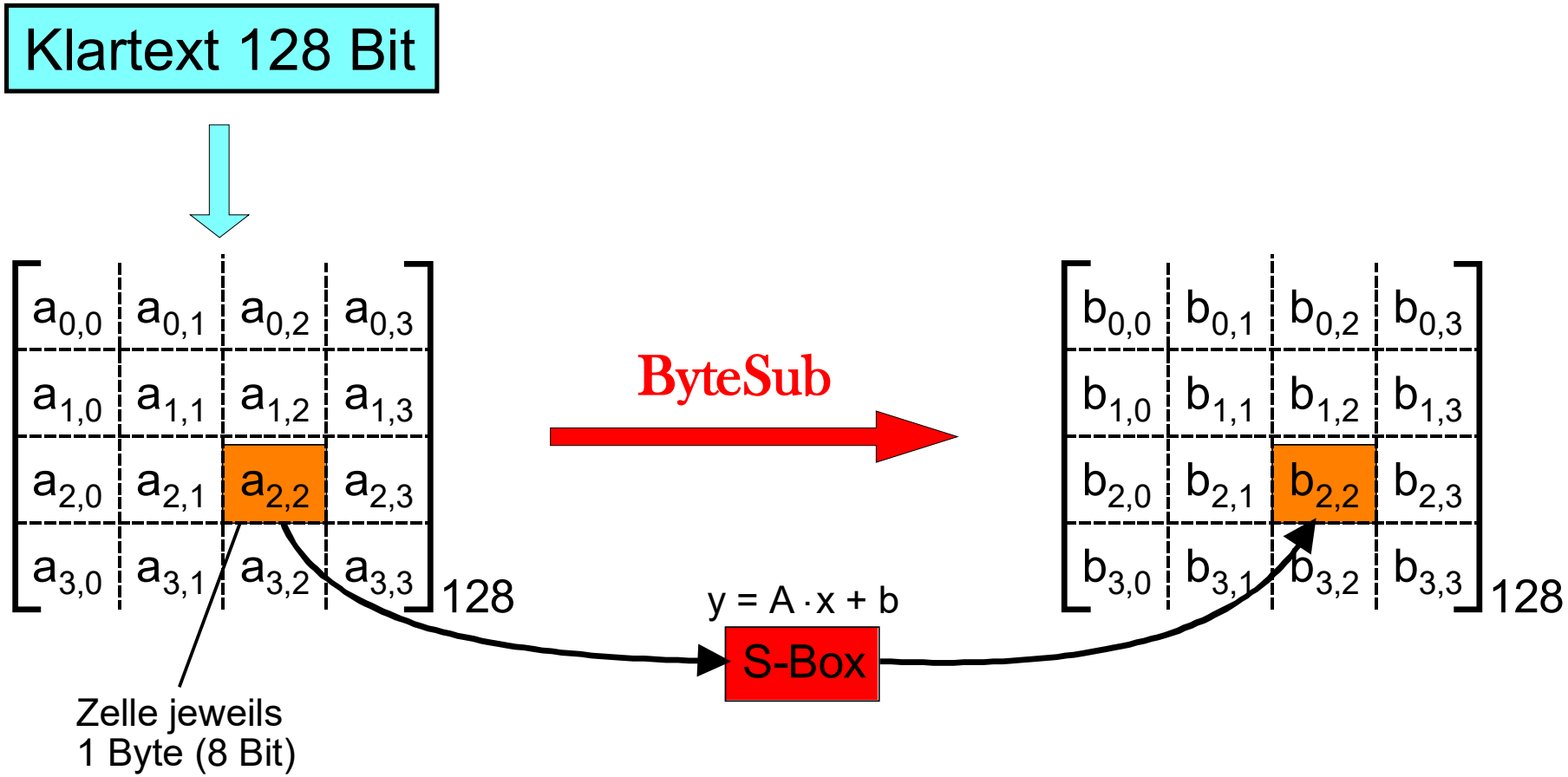
$$\mathbf{y} - \mathbf{b} = A \cdot \mathbf{x}$$
$$A^{-1}(\mathbf{y} - \mathbf{b}) = \mathbf{x}$$

$\rightarrow$ Umkehrfunktion

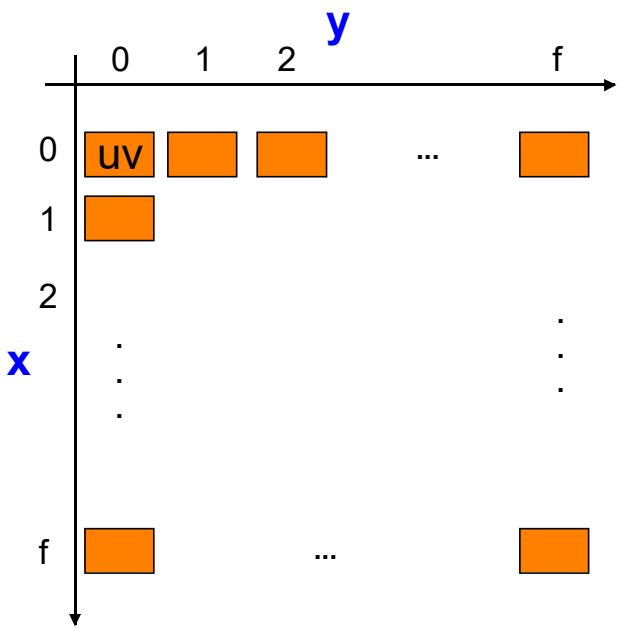
$$\mathbf{x} = B \cdot \mathbf{y} + \mathbf{c}$$

mit

$$B = A^{-1} \pmod 2 \quad \text{und} \quad \mathbf{c} = B \cdot \mathbf{b} \pmod 2$$


Klartext 128 Bit
a0,0 a0,1 a0,2 a0,3
a1,0 a1,1 a1,2 a1,3
a2,0 a2,1 a2,2 a2,3
a3,0 a3,1 a3,2 a3,3
128
ByteSub
y = A · x + b
S-Box
b0,0 b0,1 b0,2 b0,3
b1,0 b1,1 b1,2 b1,3
b2,0 b2,1 b2,2 b2,3
b3,0 b3,1 b3,2 b3,3
128
Zelle jeweils
1 Byte (8 Bit)

$\mathbf{y} = A \cdot \mathbf{x} + \mathbf{b}$



$\mathbf{x} = x_7\, x_6\, x_5\, x_4\, x_3\, x_2\, x_1\, x_0$ ist 8 Bit lang $\rightarrow$ 256 Möglichkeiten

- Jedes Byte **x** (8 Bit) wird in Hexadezimaldarstellung als

  $\mathbf{x} = xy_{hex}$

  geschrieben.
- Damit insgesamt 256 Werte

  $\mathbf{y} = uv_{hex}$,

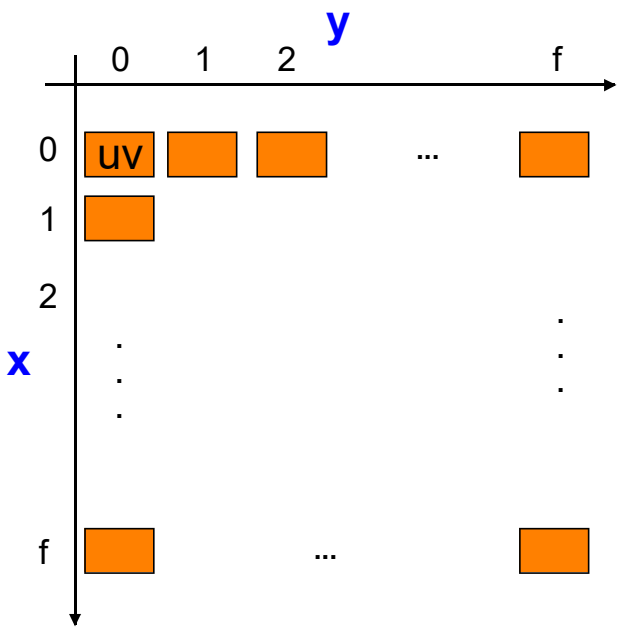
  die ebenfalls hexadezimal interpretiert werden.

$\rightarrow$ **S-Box**, Substitutionswerte uv_{hex} für das Byte xy_{hex}

Substitutionswerte:

x \ y	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

$\mathbf{x} = B \cdot \mathbf{y} + \mathbf{c}$



$\mathbf{y} = y_7\, y_6\, y_5\, y_4\, y_3\, y_2\, y_1\, y_0$ ist 8 Bit lang $\rightarrow$ 256 Möglichkeiten

- Jedes Byte **y** (8 Bit) wird in Hexadezimaldarstellung als

  $\mathbf{y} = xy_{hex}$

  geschrieben.
- Damit insgesamt 256 Werte

  $\mathbf{x} = uv_{hex}$,

  die ebenfalls hexadezimal interpretiert werden.

$\rightarrow$ **Inverse S-Box**, Substitutionswerte uv_{hex} für das Byte xy_{hex}

Substitutionswerte:

x \ y	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

Die `ShiftRow`-Transformation:

- Diese Transformation verschiebt die Zeilen 1 bis 3 der Zustands-matrix zyklisch nach **links**.
- Die Verschiebung hängt von der Blockgröße **b** ab.

- Zeile 0 wird nicht verändert.

- Zeile 1 wird im allgemeinen um c_1 Bytes, Zeile 2 und c_2 Bytes und Zeile 3 um c_3 Bytes verschoben.
- Zum Dechiffrieren erhält man die **inverse** Transformation durch Ausführen der zyklischen Verschiebung nach **rechts**.

Die `ShiftRow`-Transformation (Fortsetzung):

		Blocklänge		
Verschiebungen		**b** = 128	**b** = 192	**b** = 256
c_1		**1**	**1**	**1**
c_2		**2**	**2**	**3**
c_3		**3**	**3**	**4**

hier: Im Falle von **AES** gilt **b** = 128.

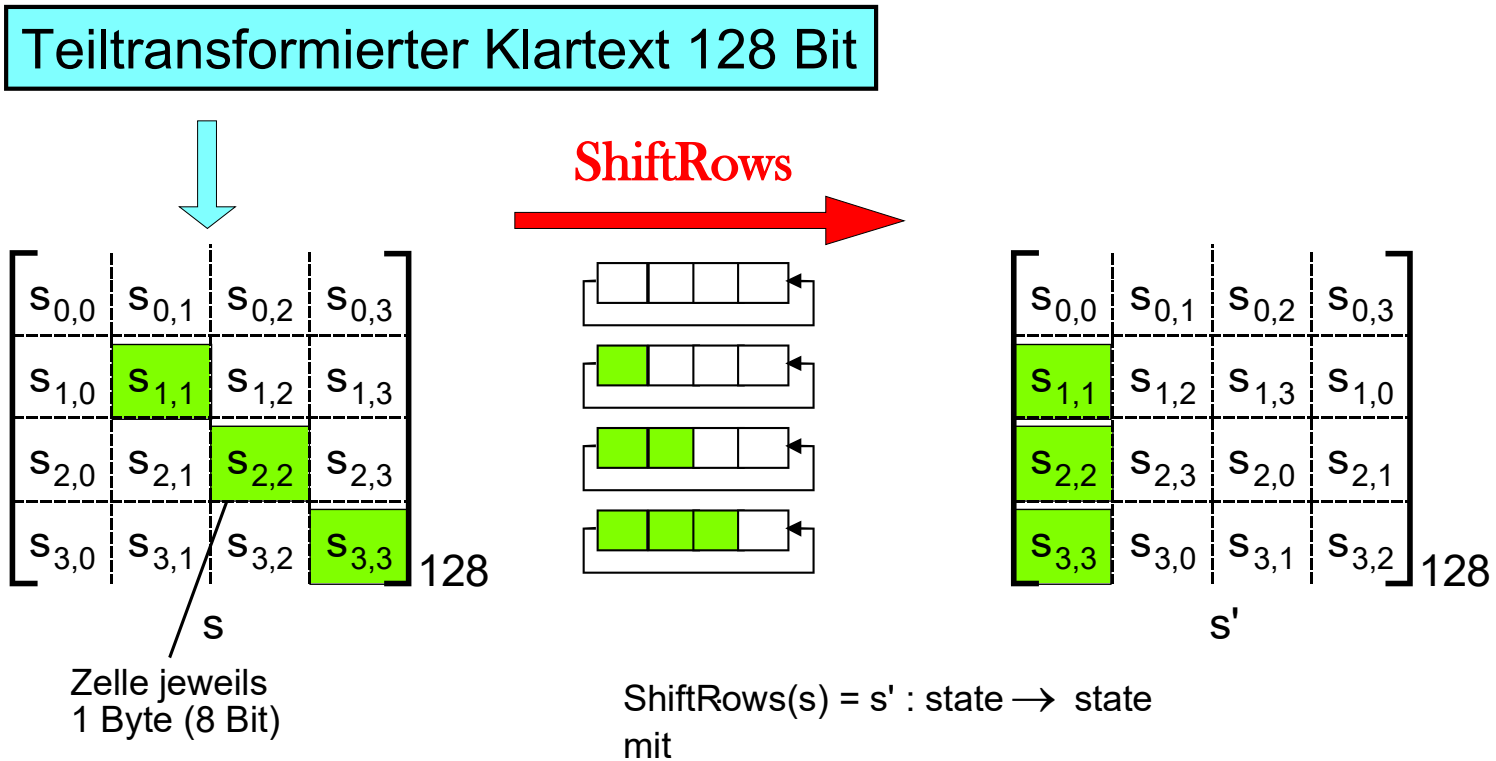
Die `ShiftRow`-Transformation (Fortsetzung):

- Sei **s** ein **state**, also nach vorangegangener Substitution ein **teiltransformierter Klartext**.
- Schreibe **s** als **Zustandsmatrix** mit **4 Zeilen** und **4 Spalten**. Die Matrixeinträge sind jeweils Bytes.
- Verschiebe die letzten drei Zeilen der Zustandsmatrix **s** zyklisch nach links.

$$\text{LeftShift(Zeile i)} = i \quad \text{für} \quad i \in \{0, 1, 2, 3\}$$

- Es ergibt sich eine Abbildung **state** $\rightarrow$ **state**, die bei Anwendung in mehreren Runden für eine **hohe Diffusion** sorgt.

Die Wirkung der `ShiftRow`-Transformation:



ShiftRows(s) = s' : state $\rightarrow$ state

mit

$$s'_{r,s} = s_{r,(s+r) \bmod 4} \quad \text{für } r,s \in \{0, 1, 2, 3\}$$

Die `MixColumn`-Transformation:

- Diese Transformation wirkt auf verschiedene **Spalten k** $\in \{0, 1, 2, 3\}$ der **Zustandmatrix s** und sorgt dort jeweils für eine **Vermischung**.
- Die Elemente der vier Spaltenvektoren $\mathbf{s}_k = (\mathbf{s}_{0,k}, \mathbf{s}_{1,k}, \mathbf{s}_{2,k}, \mathbf{s}_{3,k})$ sind **1 Byte** lang und werden als Hexadezimalzahl $\mathbf{(xy)}_{hex}$ interpretiert.
- Ferner werden die Elemente $\mathbf{s}_{0,k}, \mathbf{s}_{1,k}, \ldots, \mathbf{s}_{3,k}$ einer jeden Spalte $\mathbf{s}_k$ als Koeffizienten eines Polynoms in $GF(2^8)[x] / (x^4 + 1)$ aufgefasst:

  $$\mathbf{s}_k(\mathbf{x}) = \mathbf{s}_{3,k} \cdot x^3 + \mathbf{s}_{2,k} \cdot x^2 + \mathbf{s}_{1,k} \cdot x + \mathbf{s}_{0,k} \in GF(2^8)[x] / (x^4 + 1)$$

- Die Transformation **`MixColumn`** setzt nun

  $$\mathbf{s}_k(\mathbf{x}) \leftarrow (\mathbf{s}_k(\mathbf{x}) \bullet a(x)) \bmod (x^4 + 1),\ 0 \le k \le 3,$$

  wobei $a(x)$ das feste Polynom $(03) \cdot x^3 + (01) \cdot x^2 + (01) \cdot x + (02)$ ist.

Grundlagen:

- Speziell für den **AES** wählen wir einen **endlichen Körper** $GF(2^8)$ der Charakteristik 2 mit dem zugehörigen Polynom m(x) vom Grad 8.

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

- Es ist also:

$$GF(2^8) = \{a(x) \mid a_7 \cdot x^7 + \ldots + a_1 \cdot x + a_0$$
$$\text{für } a_i \in \mathbf{Z}_2 = \{0, 1\},\ i = 0, 1, \ldots, 7\}$$

- Die Elemente werden auch als Binärstrings $a_7\ a_6 \ldots a_0$ (bzw. als Bytes) oder in hexadezimaler Notation $xy_{hex} = (xy)$ geschrieben.

Beispiel: $x^7 + x^6 + 1 = 1100\ 0001 = c1_{hex} = (c1)$

Grundlagen (Fortsetzung):

- Die Addition $\oplus$ in **GF(2^8)** erfolgt komponentenweise (XOR) und die Multiplikation $\bullet$ wird in **GF(2^8) modulo m(x)** durchgeführt.

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

Beispiel: (57) $\bullet$ (83) = 0101 0111 $\bullet$ 1000 0011 = ?

$\rightarrow$

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

also

= **(c1)**

Die `MixColumn`-Transformation (Fortsetzung):

- Dies kann wiederum als **lineare** Transformation ($\rightarrow$ Matrix-multiplikation **s'** = A $\cdot$**s**) in $(GF(2^8))^4$ über dem Körper $GF(2^8)$ für $k \in \{0, 1, 2, 3\}$ beschrieben werden:

$$\begin{pmatrix} s_{0,k} \\ s_{1,k} \\ s_{2,k} \\ s_{3,k} \end{pmatrix} \leftarrow \begin{pmatrix} (02) & (03) & (01) & (01) \\ (01) & (02) & (03) & (01) \\ (01) & (01) & (02) & (03) \\ (03) & (01) & (01) & (02) \end{pmatrix} \bullet \begin{pmatrix} s_{0,k} \\ s_{1,k} \\ s_{2,k} \\ s_{3,k} \end{pmatrix}$$

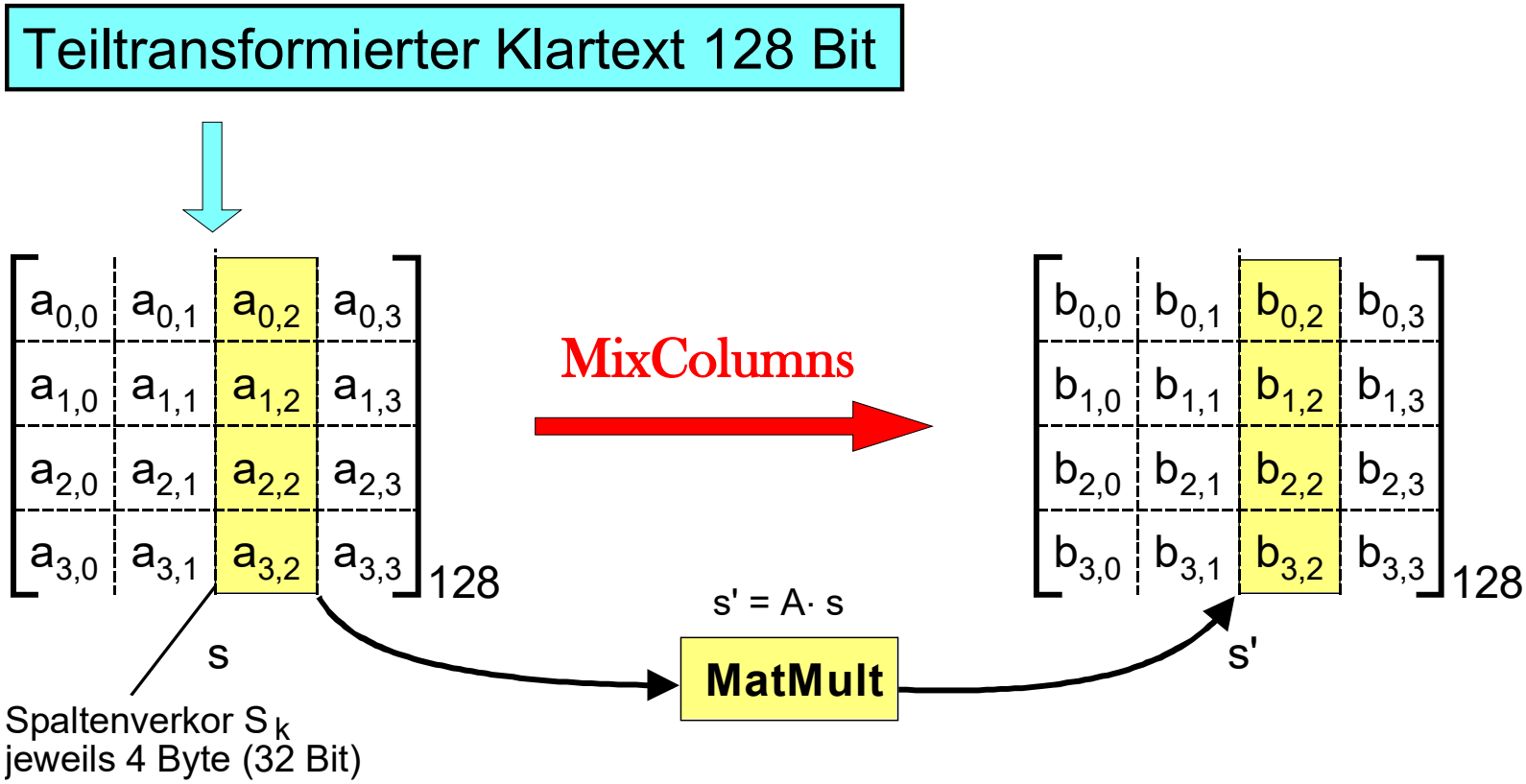
- Diese Transformation sorgt somit für eine **gute Diffusion** innerhalb der Spalten von **state**.
- Auch die **`MixColumn`**-Transformation ist **invertierbar**.

Anmerkung zur Invertierbarkeit:

- AES verwendet Polynome über dem Körper $GF(2^8)$, aber nur solche der Form $a_3 \cdot x^3 + a_2 \cdot x^2 + a_1 \cdot x + a_0$ für $a_i \in GF(2^8)$, i = 0, 1, 2, 3.
- Daher müssen Reduktionen **modulo** einem Polynom über $GF(2^8)$ vom Grad 4 durchgeführt werden. Es wird $x^4 + 1 \in GF(2^8)$ gewählt.
- Wir bilden also den Ring (und keinen Körper!!!) $GF(2^8)[x] / (x^4 + 1)$.

- Somit muss ein Element dieses so gebildeten Rings nicht unbedingt eine **Inverse** besitzen.
- Durch die spezielle Wahl $a(x) = (03) \cdot x^3 + (01) \cdot x^2 + (01) \cdot x + (02)$ existiert jedoch das **Inverse** $a^{-1}(x) = (0b) \cdot x^3 + (0d) \cdot x^2 + (09) \cdot x + (0e)$.

Die Wirkung der `MixColumn` -Transformation:



AES-Funktionen Cipher und KeyExpansion:

Der `KeyExpansion`-Algorithmus:

- Der Chiffrierschlüssel **K** wird beim AES durch Schlüsselexpansion so aufgeweitet, dass sich **r** + 1 Teilschlüssel mit je **b** Bits bilden.
- Bei einer Blocklänge von **b** = 128 Bit und **zwölf Runden** werden somit insgesamt $128 \cdot 13 = 1664$ Schlüsselbits generiert.

Der `Cipher`-Algorithmus:

- Eingabe ist der Klartextblock (128 Bit) und der expandierte Schlüssel (1664 Bit).
- Ausgabe ist nach **zwölf Runden** der Chiffretextblock (128 Bit).

AES-Dechiffrierfunktion:

Der `InvCipher`-Algorithmus:

- Die Entschlüsselung des AES wird von der Funktion **`InvCipher`** besorgt.
- Man erhält die Dechiffrierfunktion dadurch, dass wir die Reihenfolge der zuvor betrachteten Transformationen umdrehen und dabei – außen für Berechnung der Teilschlüssel – die jeweiligen inversen Transformationen betrachten.
- Des weiteren werden die Teilschlüssel in der umgekehrten Reihenfolge benutzt.
  - Ausgabe ist nach **zwölf Runden** der Klartextblock (128 Bit).

Grundlegende Konstruktionsprinzipien

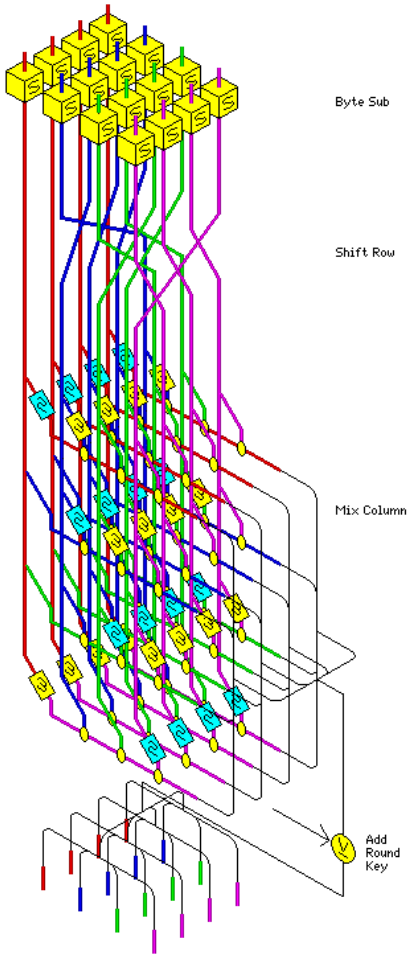
(**Claude Elwood Shannon**, Begründer der Informationstheorie)

Konfusion:

Auflösen von statistischen Strukturen (z. B. Buchstabenhäufigkeiten) eines Klartextes beim Verschlüsseln, d. h. jedes Ciphertextzeichen sollte von möglichst vielen Klartextzeichen abhängig sein.

Diffusion:

Verschleierung des Zusammenhangs zwischen Klartext und Geheimtext, d. h. bei einer Änderung von **einem** Schlüsselbit oder **einem** Klartextbit sollte sich 50 % des Geheimtextes ändern.



Kap. 4: Symmetrische Verfahren und moderne Blockchiffren

Teil 2: Betriebsmodi

- ECB
- CBC
- CFB
- OFB

ECB – Electronic Code Book

Wie bei einem Wörterbuch gibt es zu 2^N möglichen Klartextstrings 2^N Schlüsselstrings und umgekehrt.

Eigenschaften:

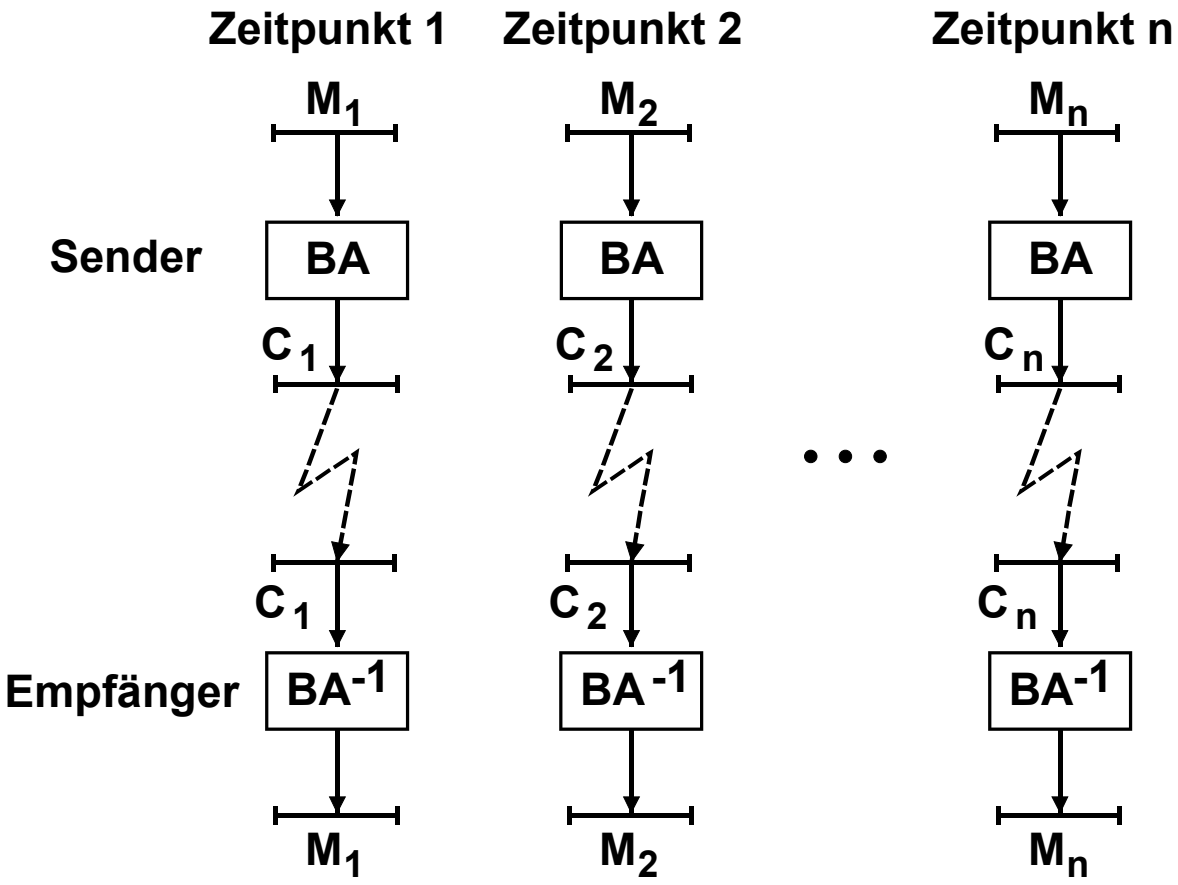
- Jeweils ein Block von N Bit wird **unabhängig** von anderen Blöcken verschlüsselt.
- Reihenfolge kann verändert werden, ohne daß die Entschlüsselung davon beeinflußt wird.
- Gleicher Klartext ergibt gleichen Schlüsseltext (sicherheitskritisch → keine identische Blöcke!).

Fehlerfortpflanzung:

- Bitfehler oder Bitgruppenfehler eines Schlüsseltextblockes verursachen einen fehlerhaften Klartextblock (mindestens 50 % aller Bits im Outputblock betroffen).

Synchronisation:

- Wenn Blockgrenzen während der Übertragung verlorengehen (z. B. Bitschlupf), geht die Synchronisation zwischen Ver- und Entschlüsselung verloren (d. h. alle Folgeblöcke werden nicht mehr korrekt entschlüsselt).



M_i = Klartextblock i

C_i = Ciphertextblock i

BA = Blockalgorithmus Verschlüsselung

BA^{-1} = Blockalgorithmus Entschlüsselung

CBC – Cipher Block Chaining

Im Gegensatz zum EBC-Modus erfolgt nun eine Verkettung der Blöcke.

Eigenschaften:

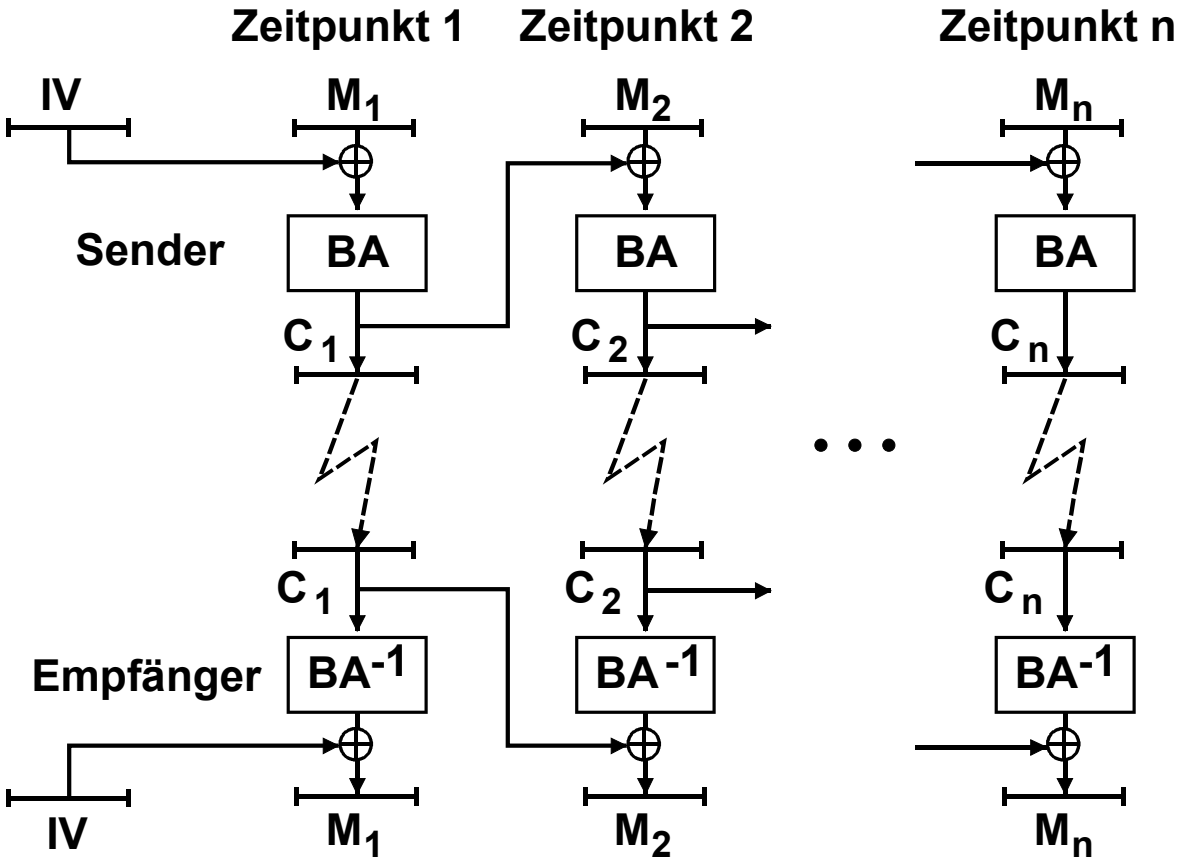
- Verkettung bewirkt, daß der Chiffretext von dem ganzen vorangegangenen Klartext und dem IV abhängt.
- Die Blöcke können daher nicht umgeordnet werden.
- Der IV verhindert, daß gleicher Klartext gleichen Chiffretext ergibt.

Fehlerfortpflanzung:

- Wenn in einem Block des Chiffretextes ein Bit- oder Bitgruppenfehler auftritt, wird die Entschlüsselung des betreffenden und des nachfolgenden Blockes gestört (→ Fehlerfortpflanzung).

Synchronisation:

- Wenn die Bitgrenzen z. B. durch Bitschlupf verlorengehen, geht auch die Synchronisation zwischen Ver- und Entschlüsselung verloren (→ Neuinitialisierung notwendig).



M_i = Klartextblock i

C_i = Ciphertextblock i

BA = Blockalgorithmus Verschlüsselung

BA^{-1} = Blockalgorithmus Entschlüsselung

IV = Initialisierungsvektor

CFB – Cipher FeedBack

Sowohl sender- als auch empfängerseitig arbeitet die Blockverschlüsselung im Verschlüsselungsmodus und erzeugt eine pseudozufällige Bitfolge E, die modulo 2 (XOR) zu den Klartextzeichen bzw. Schlüsseltextzeichen addiert wird (if j = 1 ⇒ Bitstromverschlüsselung; if j = N ⇒ verkettete Blockverschlüsselung).

Eigenschaften:

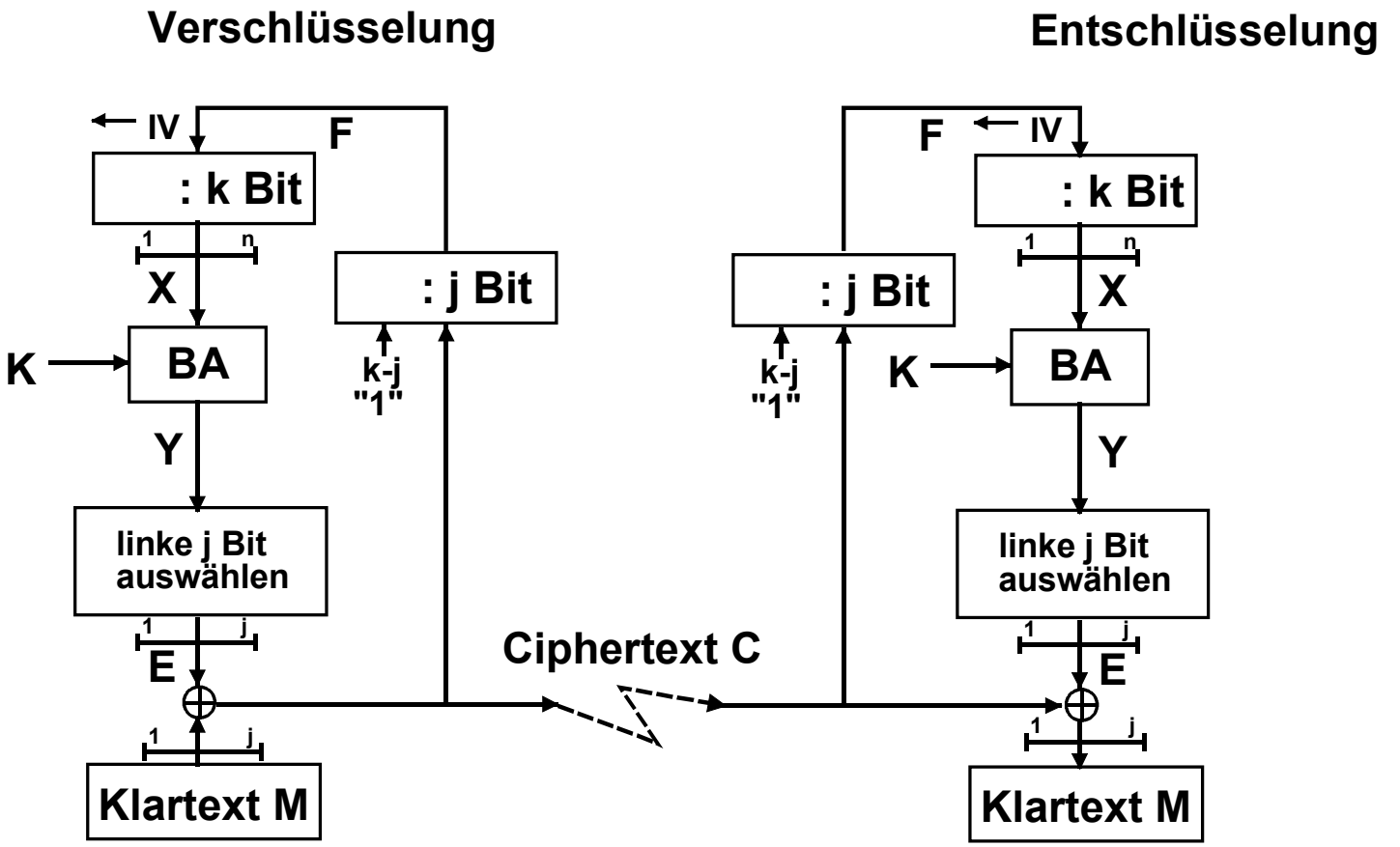
- Wenn implementierter Blockalgorithmus BA einen Durchsatz von T Bit/s bietet, so leistet der CFB-Modus effektiv nur noch T * j/N Bit/s (j = Länge der Klartextvariablen).
- Falls gleicher Schlüssel und IV verwendet wird, produziert CFB-Modus bei gleichem Klartext gleichen Chiffretext.

Fehlerfortpflanzung:

- Falls im CFB-Modus eine Chiffretextvariable C gestört wird, so werden solange falsche Klartextzeichen generiert, bis fehlerhafte Bits beim Empfänger herausgefiltert wurden.

Synchronisation:

- Wenn Variablengrenze verloren geht, sind Sender und Empfänger solange außer Synchronisation, bis Blockgrenzen wieder erreicht, d. h. CBF-Modus ist **selbst**synchronisierend.


Verschlüsselung
Entschlüsselung
IV
F
: k Bit
1
n
X
: j Bit
K
BA
k-j
"1"
Y
linke j Bit
auswählen
1
j
E
Ciphertext C
Klartext M

OFB – Output FeedBack

Im Gegensatz zum CFB-Modus werden beim OFB-Modus nicht die Chiffretext-variablen C, sondern der Output der Blockverschlüsselung BA als Input für die nächste Verschlüsselungsoperation zurückgeführt.

Eigenschaften:

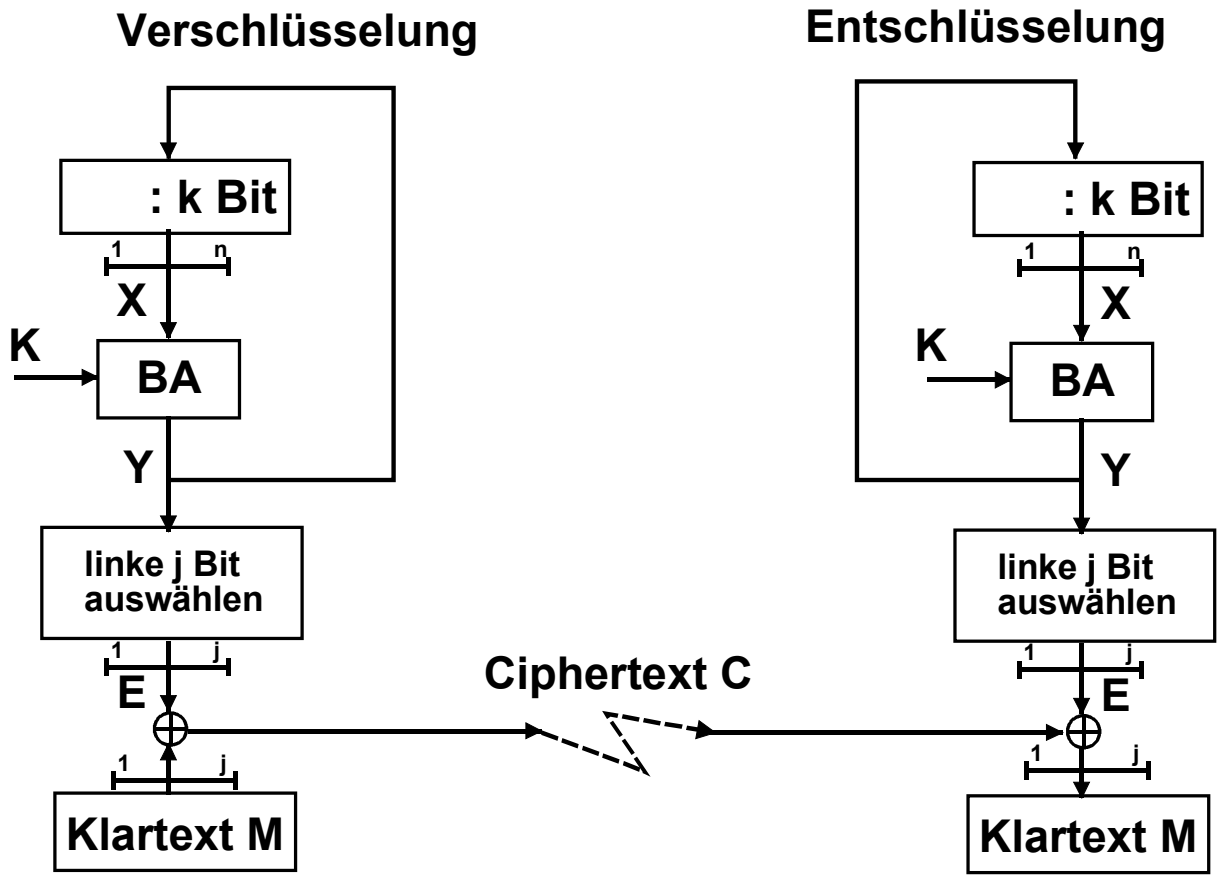
- Der erzeugte Schlüsselstrom hängt nicht vom Klartext ab.
- Da keine Verkettung erfolgt, ist der OFB durch spezifische Angriffe gefährdet.
- Gleicher Klartext ergibt gleichen Chiffretext, falls gleicher Schlüssel und IV verwendet wird.

Fehlerfortpflanzung:

- Es gibt keine Fehlerfortpflanzung, solange die Ver- und Entschlüsselung synchron erfolgt.
- Jedes fehlerhafte Bit im Schlüsseltext ergibt ein fehlerhaftes Bit im Klartext.

Synchronisation:

- Der OFB-Modus ist nicht selbstsynchronisierend.
- Tritt z. B. Bitschlupf auf, muß System neu initialisiert werden.


Verschlüsselung
: k Bit
1
n
X
K
BA
Y
linke j Bit auswählen
1
j
E
1
j
Klartext M
Ciphertext C
Entschlüsselung
: k Bit
1
n
X
K
BA
Y
linke j Bit auswählen
1
j
E
1
j
Klartext M

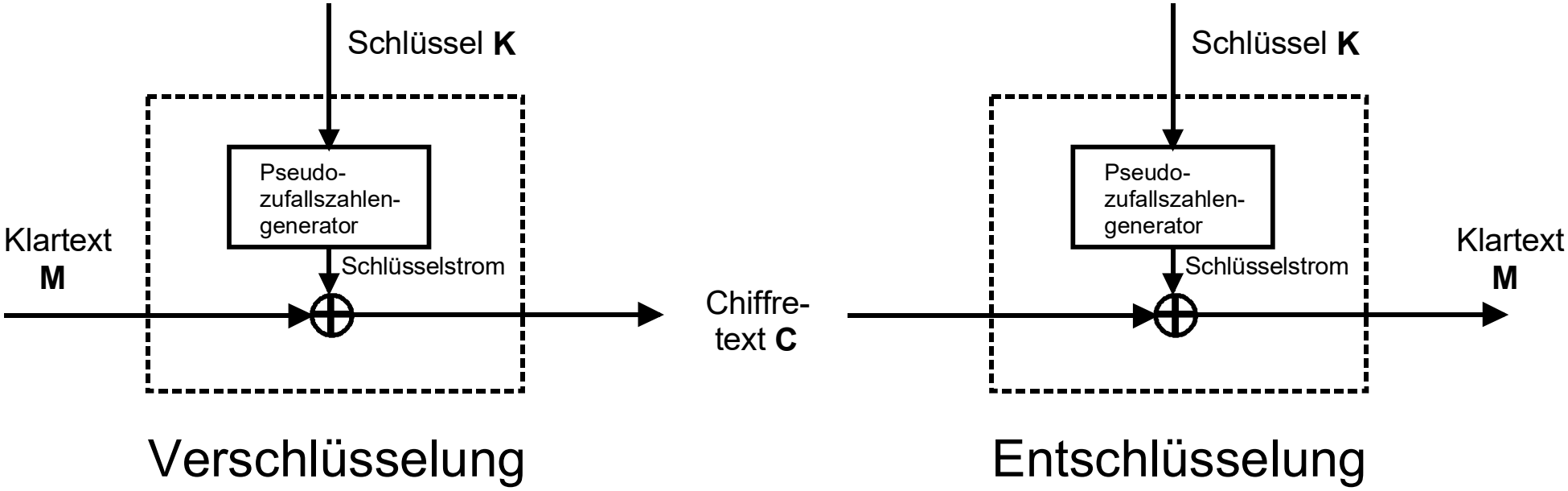
Kap. 4: Symmetrische Verfahren und moderne Blockchiffren

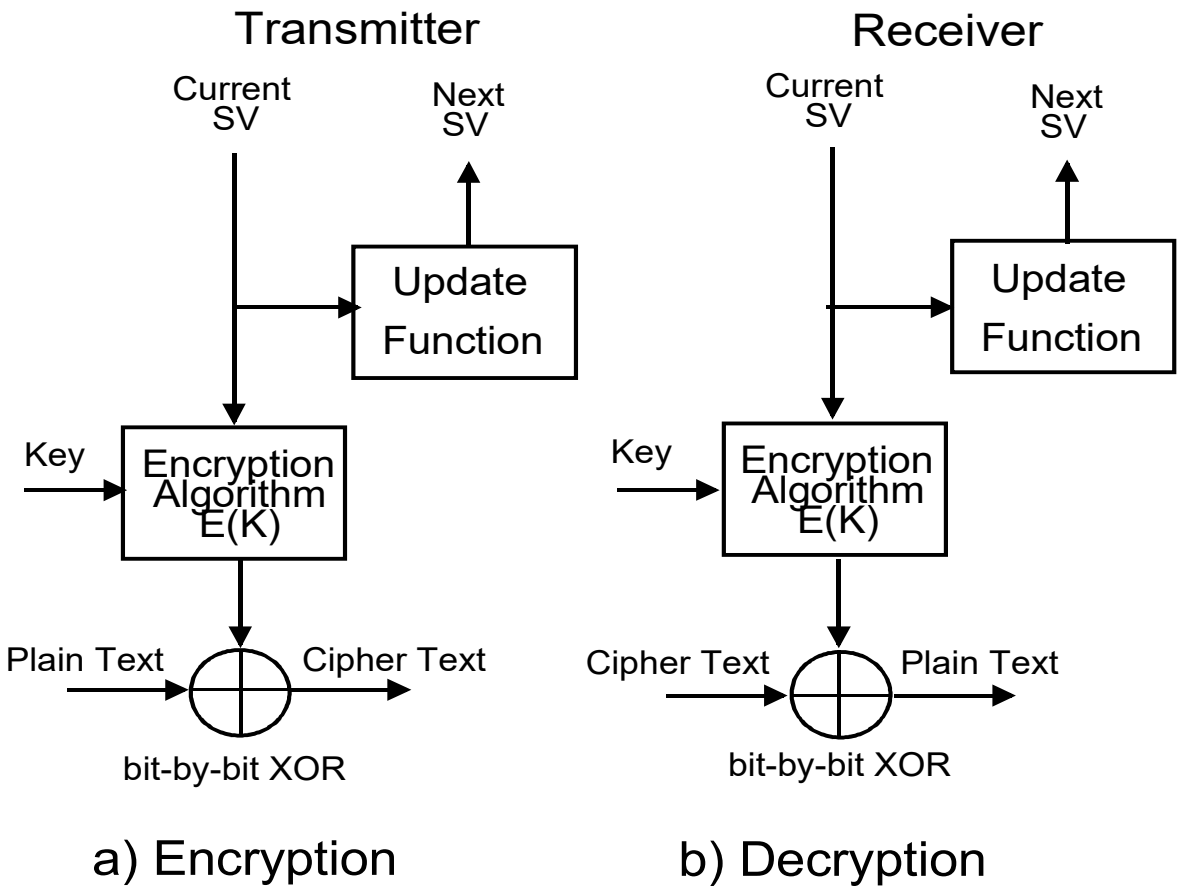
Teil 3: Symmetrische Bitstromverschlüsselung

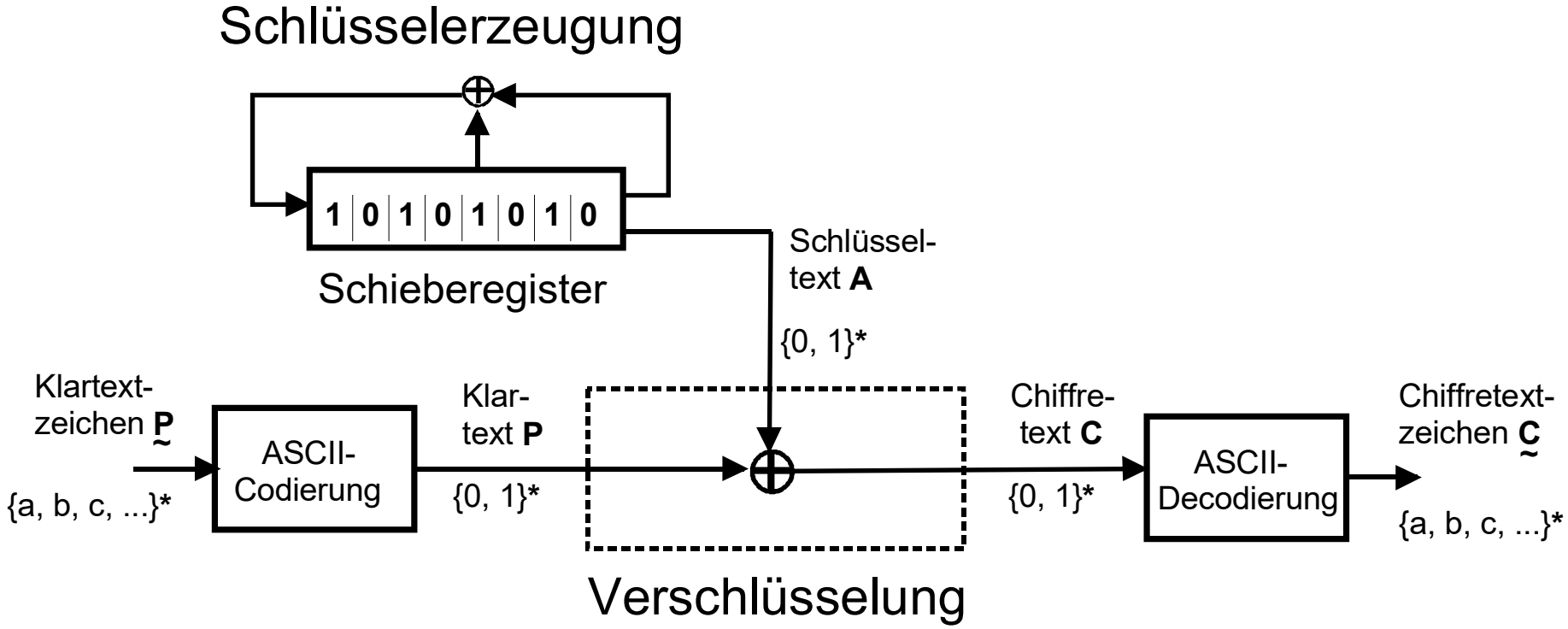
- Bitstromverschlüselung mittels XOR-Algorithmus
- One-Time-Pad und perfekte Sicherheit

Bitstromverschlüsselung

- Symmetrische Verschlüsselungsverfahren können in Blockalgorithmen und Bitstrom-verschlüsselung unterschieden werden.
- Bei der Bitstromverschlüsselung wird der Klartext Bit-für-Bit verschlüsselt.
- Der angewandte geheime Schlüssel(strom) wird in einem Pseudo-Zufallszahlen-generator eingegeben oder der Anfangszustand des Pseudo-Zufallszahlengenerators wird mit dem geheimen Schlüssel vorbelegt (Initialisierung).
- Die Periode der vom Pseudo-Zufallszahlengenerator erzeugten Folge muß größer als die zu verschlüsselnde Nachricht sein (sog. one time pat).
- Die Output-Bits werden auf der Verschlüsselungsseite mit dem Klartext durch XOR verknüpft.
- Auf der Entschlüsselungsseite wird der Pseudo-Zufallszahlengenerator mit demselben geheimen Schlüssel vorbelegt, und die Output-Folge mit dem Schlüsseltext wiederum durch XOR verknüpft, so daß man wieder den ursprünglichen Klartext erhält.
- Ein besonderes Problem stellt der Austausch des geheimen Schlüssels dar.


Schlüssel K
Pseudo-
zufallszahlen-
generator
Schlüsselstrom
Klartext
M
Chiffre-
text C
Schlüssel K
Pseudo-
zufallszahlen-
generator
Schlüsselstrom
Klartext
M
Verschlüsselung
Entschlüsselung


Transmitter
Current SV
Next SV
Update Function
Key
Encryption Algorithm E(K)
Plain Text
Cipher Text
bit-by-bit XOR
a) Encryption
Receiver
Current SV
Next SV
Update Function
Key
Encryption Algorithm E(K)
Cipher Text
Plain Text
bit-by-bit XOR
b) Decryption


Schlüsselerzeugung
1 0 1 0 1 0 1 0
Schieberegister
Schlüssel-
text **A**
{0, 1}*
Klartext-
zeichen **P**
{a, b, c, ...}*
ASCII-
Codierung
Klar-
text **P**
{0, 1}*
Chiffre-
text **C**
{0, 1}*
ASCII-
Decodierung
Chiffretext-
zeichen **C**
{a, b, c, ...}*
Verschlüsselung

Bitstromverschlüsselung

Klartext: $a_1, a_2, ..., a_n$

Schlüssel: $k_1, k_2, ..., k_n$

Geheimtext: $c_1, c_2, ..., c_n$

mit $a_i, k_i, c_i \in \{0, 1\}$

$i = 1, 2, ..., n$

Rechenvorschrift: **Binäre Addition modulo 2**

$1 \oplus 1 = 0$

$0 \oplus 0 = 0$

$1 \oplus 0 = 1$

$0 \oplus 1 = 1$

$a_i \oplus k_i := c_i$

Entschlüsselung:

$c_i \oplus k_i := a_i \oplus k_i \oplus k_i$

$:= a_i \oplus 0$

$:= a_i$

Eigenschaften:

- alle Folgen der Länge n mit derselben Wahrscheinlichkeit
- ohne Kenntnis von k_i läßt sich nicht auf a_i schließen
- mit Kenntnis von k_i läßt sich a_i aus c_i rekonstruieren
- gleicher Schlüssel auf beiden Seiten → (geheimer) Schlüsselaustausch notwendig!

	Sender	Übertragungsweg	Empfänger
	↓	↓	↓
Klartext:	0 110 110 011	(unsicher bzw.	
Schlüssel:	1 001 010 111	ungesichert)	
Geheimtext:	**1 111 100 100**		
	Geheimtext:	**1 111 100 100**	
		Geheimtext:	**1 111 100 100**
		Schlüssel:	1 001 010 111

Standardalgorithmen

- **56 bit DES-Schlüssel geknackt**
  - bis zu 7 Milliarden Schlüssel pro Sekunde ausprobiert
  - nach etwa 25 % der möglichen 72 Billiarden Schlüssel wurde der richtige gefunden
  - gerechnet wurde während der Wartezeiten gewöhnlicher Rechner von tausenden Leuten, die sich nie gesehen haben
- **RSA: "sicher" heißt nur "relativ sicher"**
  - 100 Pentium-Prozessoren mit 100 MHz Taktfrequenz brauchen rund 1 Jahr an Rechenzeit, um einen 428-Bit-RSA-Schlüssel zu knacken
  - generell kann kein bekanntes Chiffrierverfahren[1] als (mathe. beweisbar) vollkommen sicher erachtet werden
  - Sicherheit abhängig von Rechnerleistung, Wissen, Gelegenheit etc.

[1] Einzige Ausnahme: sog. One-time-pad

One-Time-Pad

- Das One-Time-Pad wurde 1917 von Major J. Mauborgne und G. Vernam von AT&T erfunden.
- Sei $\mathbf{A} = \{0, 1\}$ ein Alphabet und $z, r \in \mathbf{A}$.
  Ein Klartextbit **z** wird mit dem Zufallbit **r** des Schlüssels chiffriert durch die Vorschrift (Vigenere-Chiffre):

$$\mathbf{z \rightarrow (z + r) \bmod 2 = z \text{ XOR } r = z \oplus r}$$

- Das One-Time-Pad gehört damit eindeutig zur Klasse der **Stromchiffren** (bitweise XOR-Verknüpfung).
- Als eines der wenigen **perfekten** Chiffriersysteme ist es gleichzeitig gemäß der vorangestellten Definition **uneingeschränkt sicher**.

Definition:

Ein Chiffriersystem heißt **perfekt**, wenn bei beliebigem Klartext **M** und beliebigem Chiffretext **C** die a-priori-Wahrscheinlichkeit **P(M)** gleich der bedingten Wahrscheinlichkeit (a-posteriori-Wahrscheinlichkeit) **P(M | C)** ist, d. h.

$$\mathbf{P(M \mid C) = P(M)} \quad \Leftrightarrow \quad \text{perfektes Chiffriersystem} \qquad (1)$$

Mit der Definition der bedingten Wahrscheinlichkeit

$$\mathbf{P(M \mid C) = P(M \wedge C) / P(C)} \qquad (2)$$

folgt aus (1):

$$\mathbf{P(M \wedge C) = P(M) \cdot P(C)} \quad \Leftrightarrow \quad \text{statistisch unabhängig} \qquad (3)$$

sog. Produktregel

d. h. um **perfekte Sicherheit** zu garantieren, müssen ein beliebiger Klartext **M** und der zugehörige Chiffretext **C** **statistisch unabhängig** sein.

Sei $\mathcal{M}$ die Menge ∀ Klartexte **M** der Länge n ,

$\mathcal{C}$ die Menge ∀ Chiffretexte **C** der Länge n und

$\mathcal{K}$ die Menge ∀ Schlüsseltexte **K** der Länge n .

Bei einem One-Time-Pad gilt:

$$\mathbf{m} := |\mathcal{M}| = |\mathcal{C}| = |\mathcal{K}| = 2^{n} , \qquad (4)$$

denn alle drei Mengen bestehen aus Texten der Länge n über einem vorgegebenen Alphabet A = {0, 1}.

Für einen beliebig vorgegebenen Klartext **M** wird jeder Chiffretext **C** mit der gleichen Wahrscheinlichkeit erzeugt:

$$P(C) = 1 / m = 2^{-n} \qquad (5)$$

Da es genau m Schlüssel gibt, gibt es auch genau m unterschiedliche Chiffretexte **C** zu jedem Klartext **M**, also

$$\mathbf{P(C \mid M) = 1 / m = 2^{-n}} \qquad (6)$$

Wendet man (2) auch auf **P(C | M)** an, so erhält man die sog. **Bayes`sche Formel**:

$$\mathbf{P(M \mid C) \cdot P(C) = P(C \mid M) \cdot P(M)} \qquad (7)$$

Einsetzen von (5) und (6) in (7) ergibt:

$$\mathbf{P(M \mid C) \cdot 1 / m = 1 / m \cdot P(M)}$$

Hieraus folgt:

$$\mathbf{P(M \mid C) = P(M)} \quad \Leftrightarrow \quad \text{One-Time-Pad = perfektes Chiffriersystem,} \qquad (8)$$

gleichzeitig **uneingeschränkt sicher**, da es keine Möglichkeit gibt, aus einem beliebig langen Chiffretext **C** – ohne Kenntnis des Schlüssels – auf den Klartext **M** zu schließen!

Bitstromverschlüsselung

mittels Coset-Muster

```
/* Datum: 19.07.2002                                              */
/* Autor: Bernhard Geib                                           */
/* Funktion: Verschluesselung mit Coset-Muster 01010101           */

#include <stdio.h>

int main (void)
{  int c;

   c = getchar();
   while (c != EOF)          {
      if (c != '\n')
      {
         c = c ^ 85;    /* hier eine Zufallszahl verwenden */
      }
      printf ("%c", c);
      c = getchar();         }

   return 0;                 }
```

Bitstromverschlüsselung

mittels selbstinverser Chiffre (1)

```
/* Datum: 17.07.2002                                  */
/* Autor: Bernhard Geib                               */
/* Funktion: Bijektive, selbstinverse Chiffre         */

#include <stdio.h>
#include <string.h>


void verschluessler(char *text, const char *geheimnis)
{
   /* Annahme: text und geheimnis zeigen jeweils auf   */
   /* string mit Zeichen aus dem Bereich 32 .. 95      */

      int i ;
      int lt = strlen(text) ;
      int lg = strlen(geheimnis) ;
```

Bitstromverschlüsselung

mittels selbstinverser Chiffre (2)

```
// Fortsetzung Verschluessler

     for (i=0; i < lt; ++i)
     {
       char c = text[i] - ' ' ;
       char key = geheimnis[i % lg] - ' ' ;

       c = c ^ key ;
       text[i] = c + ' ' ;
     }

} // Ende Verschluessler
```

Bitstromverschlüsselung

mittels selbstinverser Chiffre (3)

```
// Zugehoeriges Hauptprogramm

int main( void )
{
        char text[] = "Dies ist der gegebene Klartext" ;
        char geheimnis[] = "GEHEIM" ;

        printf("Vor  der Verschlüsselung: %s\n", text) ;
        verschluessler(text, geheimnis) ;
        printf("Nach der Verschlüsselung: %s\n", text) ;
        verschluessler(text, geheimnis) ;
        printf("Nach der Entschlüsselung: %s\n", text) ;

        return 0 ;
}
```