

# **MEA**

Ein symmetrischer Blockverschlüsselungsalgorithmus

**Michael Engel**

Autor: Michael Engel  
Datum: 06.04.2022

# Abstract

Der Blockverschlüsselungsalgorithmus MEA ist ein dynamischer symmetrischer Verschlüsselungsalgorithmus. Er setzt auf ein dynamisches Netzwerk und eine höhere Blockgröße als AES [1], die die Sicherheit und den Algorithmus mit seiner größeren Blockgröße effizienter für 64-Bit CPUs macht. Der Algorithmus hat eine dynamische SPN-ähnliche[1] Struktur mit vergrößerten MDS-Matrizen und vier neuen S-Boxen. Zudem werden verschiedenste Transformationen angewendet, damit es zur einer besseren Obfuscation kommt. Die Schlüssellänge und die Blockgröße des Algorithmus sind äquivalent mit einer Größe von 512 Bits, was durch die Verminderung der Varietäten des Algorithmus zur einer Verminderung von Schwachstellen führt. Außerdem wird eine neue Schlüsselerzeugung benutzt, die sehr schnell und effizient im Vergleich zu andern Schlüsselerzeugungen ist. Zudem wird ein Permutationsalgorithmus angewandt, der schnell und sicher eine Permutation der Funktionen in Abhängigkeit vom Schlüssel erzeugt. Die Rundenanzahl vom MEA ist höher im Vergleich zu AES [1], was die Sicherheit verbessern sollte. Aktuell sind keine effizienten und effektiven Angriffe oder Schwachstellen vom MEA bekannt, weswegen MEA zur Zeit als sehr sicher einzustufen ist. Der Blockverschlüsselungsalgorithmus MEA ist auch für Hardware-basierte Aufgaben gedacht, da er leicht auf spezielle Hardware implementierbar ist. Die unten gegebene Implementierung ist auf Schnelligkeit ausgelegt, weswegen sie in der Programmiersprache C verfasst wurde.

# Inhaltsverzeichnis

<b>Abstract</b>	<b>i</b>
<b>1 Symbole und Definitionen</b>	<b>1</b>
<b>2 Generelles</b>	<b>2</b>
2.1 Input und Output . . . . .	2
<b>3 Verschlüsselung</b>	<b>3</b>
3.1 Algorithmus . . . . .	3
3.2 Die horizontale Permutation $\vartheta_l$ . . . . .	4
3.3 Das bijektive nicht lineare mapping $\Omega_l$ . . . . .	4
3.4 Die vertikale Permutation $\Gamma_l$ . . . . .	4
3.5 Die lineare Transformation $\pi_l$ . . . . .	5
3.6 Die dimensionale Permutation $\chi_l^{(h)}$ . . . . .	5
3.7 Die Modulo 2 Addition (XOR-Operation) $\kappa_l^{(K_z)}$ . . . . .	6
<b>4 Entschlüsselung</b>	<b>7</b>
4.1 Algorithmus . . . . .	7
4.2 Die Inverse der horizontalen Permutation $\hat{\vartheta}_l$ . . . . .	8
4.3 Das bijektive nicht lineare mapping $\hat{\Omega}_l$ . . . . .	8
4.4 Die Inverse der vertikalen Permutation $\hat{\Gamma}_l$ . . . . .	8
4.5 Die Inverse der linearen Transformation $\hat{\pi}_l$ . . . . .	9
4.6 Die Inverse der dimensionalen Permutation $\hat{\chi}_l^{(h)}$ . . . . .	9
<b>5 Rundenschlüssel Erzeugung</b>	<b>10</b>
<b>6 Sequenz Shuffle Funktion <math>\Psi^{(K_z)}</math></b>	<b>11</b>
<b>7 Weiteres</b>	<b>12</b>
7.1 S-Boxen $\beta_b$ und $-\beta_b$ . . . . .	12
<b>8 Implementierung in der Programmiersprache C</b>	<b>20</b>
8.1 Code . . . . .	20
8.1.1 mea.h . . . . .	20

## Inhaltsverzeichnis

# 1 Symbole und Definitionen

Die folgenden Symbole und Definitionen werden benutzt im MEA.

$0x$	- Prefix für Nummern im Hexadezimalsystem;
$\eta(x)$	- das irreduzible Polynom $x^8 + x^4 + x^3 + x^2 + 1$ ;
$GF(2^8)$	- ein endlicher Körper mit dem irreduziblen Polynom $\eta(x)$ ;
$l$	- die Blockgröße vom MEA, $l = 512$ ;
$k$	- die Schlüsselgröße vom MEA, $k = 512$ ;
$r$	- die Anzahl der Reihen in der State-Matrix, $r \in \{4, 8\}$ ;
$s$	- die Anzahl der Spalten in der State-Matrix, $s \in \{4, 8\}$ ;
$K_k$	- der Schlüssel mit Länge $k$ ;
$V_d$	- d-Dimensionaler Vektorraum im $GF(2)$ , $d \geq 1$ ;
$\oplus$	- die binäre exklusiv ODER Verknüpfung;
$\ggg$	- die rechts shift Operation mit einer konstanten Länge;
$\lll$	- die links shift Operation mit einer konstanten Länge;
$E_{l,k}^{(K_k)}$	- die symmetrische Verschlüsselungstransformation, dass mapping von $V_l \mapsto V_l$ , abhängig von $K_k$ ;
$D_{l,k}^{(K_k)}$	- die symmetrische Entschlüsselungstransformation, dass mapping von $V_l \mapsto V_l$ , abhängig von $K_k$ ;
$\tau \odot \nu$	- sequentieller Ablauf der Transformationen $\tau$ und $\nu$ , ( $\nu$ wird zuerst angewandt);
$\tau \nu$	- Ablauf der Transformationen $\tau$ oder $\nu$ (jede wird einmal ausgeführt), Permutation generiert durch die Sequenz-Shuffle Funktion $\Psi^{K_k}$ ;
$\mu  \lambda$	- Ablauf der Packete $\mu$ oder $\lambda$ , $\mu$ wird bei $i \bmod 2 = 0$ ausgeführt, abhängig von der Hauptrundenzahl $i$ , ansonsten wird $\lambda$ ausgeführt;
$\mu \doteq \lambda$	Substitution der Elemente $\mu \& \lambda$ , $\mu \mapsto \lambda$ und $\lambda \mapsto \mu$ ;
$n$	- die jeweilige Anzahl der Iterationen in den Transformationen $E_{l,k}^{(K_k)}$ und $D_{l,k}^{(K_k)}$ , $n = 36$ ;
$meal_{l,k}$	Applikation der Transformationen $E_{l,k}^{(K_k)}$ und $D_{l,k}^{(K_k)}$ ;
$\prod_{i=c}^n \tau^{(i)}$	- sequentieller Ablauf der Transformationen $\tau^{(c)}$ , $\tau^{(c+1)}, \tau^{(c+2)}, \dots, \tau^{(n)}$ , ( $\tau^{(c)}$ wird zuerst angewandt);

## 2 Generelles

Die Verschlüsselungstransformation ist das mapping von  $E_{l,k}^{(K_k)}: V_d \mapsto V_d$ , dass vom Schlüssel  $K \in V_k$  abhängig ist, wobei  $l = 512$  und  $k = 512$ , also  $l = k$ .  $E_{l,k}^{(K_k)}$  ist definiert als eine Reihenfolge von  $n$  Paketen, jeweils bestehend aus  $\sqrt{n}$  Funktionen, wobei bei  $\frac{n}{2}$  Paketen, falls  $i_i \in \{0, 1, 2, 3, \dots, 35\}$ ,  $i_i \bmod 2 = 0$  (die Rundenzahl), die Reihenfolge der Funktionen im Paket konstant ist. Ansonsten ist bei  $i_i \bmod 2 \neq 0$  die Reihenfolge eine nicht vorhersehbare Permutation der  $\sqrt{n}$  Funktionen in einem dynamischen Paket, die in Abhängigkeit von  $K_k$  generiert wird. Die jeweiligen Funktionen nehmen eine  $r * s$  Matrix im  $GF(2^8)$  als Input, wobei  $r = 8$  und  $s = 8$  und  $x \in V_l$ . Die  $r * s$  Matrix ist der Cihper State. Die Entschlüsselungstransformation  $D_{l,k}^{(K_k)}$  in Abhängigkeit von  $K$  ist das inversive mapping von  $E_{l,k}^{(K_k)}$  mit allen Inversiven der Funktionen.  
Alle Parameter, die  $meal_{l,k}$  definieren, sind in Tabelle 1 angegeben.

MEA				
Blockgröße	Rundenanzahl	Schlüssellänge	Anzahl der Reihen	Anzahl der Spalten
512	36	512	8 $\vee$ 4	8 $\vee$ 4

Tabelle 1

### 2.1 Input und Output

Die Transformationprozesse nehmen als Input einen Block der Länge  $l$ , egal ob bei der Verschlüsselung oder Entschlüsselung und geben am Ende der Transformationen einen Block mit der Länge  $l$  als Output. Die State-Matrix  $S$  wird repräsentiert durch  $(s_{a,d,h})$ , wobei  $(s_{a,d,h}) \in V_8$  (bei  $\vartheta_l$ ,  $\Gamma_l$  und  $\chi_l^{(h)}$ ),  $a = \overline{0, r - 1}$ ,  $d = \overline{0, s - 1}$  und falls  $r = 4$ ,  $s = 4$  ist, ist  $h = \overline{0, 3}$ . Ansonsten ist  $h = 1$ . Die State-Matrix wird befüllt mit den Input Bytes  $B_1, B_2, B_4, \dots, B_{l/8}$  in der Row-Major Order, dass heißt, dass als erstes die erste Reihe sequentiell von links nach rechts befüllt wird und danach die darunterliegende Reihe, bis alle Reihen der Matrix voll sind. Falls die Input Nachricht  $P \bmod 64 \neq 0$  (in Bytes) ist, muss ein Padding Algorithmus<sup>1</sup> angewendet werden, damit die Nachricht in Bytes  $P \bmod 64 = 0$  erfüllt.

---

<sup>1</sup>Ein Padding Algorithmus ist ein Algorithmus, der einen vorhandenen Datenbestand mit Fülldaten füllt.

# 3 Verschlüsselung

## 3.1 Algorithmus

Der Verschlüsselungsalgorithmus  $E_{l,k}^{(K_k)}$  ist wie folgt definiert:

$$E_{l,k}^{(K_k)} = \prod_{i=0}^{\sqrt{n}-1} \prod_{t=0}^{\sqrt{n}-1} (\kappa_l^{(K_z)} \odot \chi_l^{(\frac{t}{2})} \odot \pi_l \odot \Gamma_l \odot \Omega_l \odot \vartheta_l) || (\kappa_l^{(K_z)} \odot (\vartheta_l | \Omega_l | \Gamma_l | \pi_l | \chi_l^{(\frac{t+1}{2})} | \kappa_l^{(K_z)}) \Psi^{(K_z)}),$$

wo  $K_k$  der Schlüssel mit der Länge  $k$  ist,

- $\vartheta_l$  - die horizontale Permutation der Elemente  $(s_{a,d,h})$ , wo  $(s_{a,d,h}) \in GF(2^8)$ ,  $a = \overline{0,3}$ ,  $d = \overline{0,3}$  und  $h = \overline{0,3}$ , mit dem Cipher-State  $S$ ,
- $\Omega_l$  - das bijektive nicht lineare mapping der S-Boxen  $\beta_b$ ,  $b \in \{0, 1, 2, 3\}$  mit den State-Matrix Vektoren,
- $\Gamma_l$  - die vertikale Permutation der Elemente  $(s_{a,d,h})$ , wo  $(s_{a,d,h}) \in GF(2^8)$ ,  $a = \overline{0,3}$ ,  $d = \overline{0,3}$  und  $h = \overline{0,3}$ , mit dem Cipher-State  $S$ ,
- $\pi_l$  - die lineare Transformation des Cipher-State über das endliche Feld  $GF(2^8)$ ,
- $\chi_l^{(\frac{t+1}{2})}$  - die dimensionale Permutation der Elemente  $(s_{a,d,h})$ , wo  $(s_{a,d,h}) \in GF(2^8)$ ,  $a = \overline{0,3}$ ,  $d = \overline{0,3}$  und  $h = \overline{0,3}$ , mit dem Cipher-State  $S$ , bei  $h = \frac{t+1}{2}$ ,
- $\chi_l^{(\frac{t}{2})}$  - die dimensionale Permutation der Elemente  $(s_{a,d,h})$ , wo  $(s_{a,d,h}) \in GF(2^8)$ ,  $a = \overline{0,3}$ ,  $d = \overline{0,3}$  und  $h = \overline{0,3}$ , mit dem Cipher-State  $S$ , bei  $h = \frac{t}{2}$ ,
- $\kappa_l^{(K_z)}$  - eine Modulo 2 Addition (XOR-Operation) mit den Rundenschlüssel  $K_l^{(K_z)}$ ,  $z = i \times 6 + t$  und mit der State-Matrix ist.

In den Funktionen  $\vartheta_l$ ,  $\Gamma_l$ ,  $\chi_l^{((t+1)/2)}$  und  $\chi_l^{(t/2)}$ , mit dem Input  $x \in V_l$ , werden die Permutationen in einer dreidimensionalen  $4 \times 4 \times 4$  State-Matrix ausgeführt ( $r = 4$ ,  $s = 4$ ,  $h = \overline{0,3}$ ), um eine bessere Obfuscation zu erzielen. Ansonsten wird immer eine zweidimensionale  $8 \times 8$  State-Matrix benutzt ( $r = 8$ ,  $s = 8$ ,  $h = 1$ ). Als Rückgabe aller Funktionen wird eine zweidimensionale  $8 \times 8$  State-Matrix ausgegeben.

## 3.2 Die horizontale Permutation $\vartheta_l$

Die horizontale Permutation  $\vartheta_l$  ist eine horizontale rechts shift Operation, die jede Reihe der drei dimensionalen State-Matrix  $S = (s_{a,d,h})$ ,  $r = 4, s = 4, h = \overline{0,3}$ , um  $\zeta_r$  Positionen in einer Reihe  $r_h$  nach rechts bewegt.  $\zeta_r$  ist abhängig von der Reihennummer  $r_h \in \{0, 1, 2, 3\}$ , die Blockgröße  $l$  und kann mit der Formel  $\zeta_r = \frac{r \times l}{512}$  beschrieben werden. So wird jede Reihe in jeder der vier Dimensionen um die Anzahl der Reihenzahl der Reihe nach rechts verschoben. So wird zum Beispiel jedes Element der Reihe  $r_h = 2$  (in der dritten Reihe) um 2 Position nach rechts verschoben. Die Elemente, die rechts aus der Reihe gehen, werden wieder links angehängt. Dieser Prozess wird für jeder der  $h = \overline{0,3}$  Dimensionen durchgeführt.

Ein Beispiel für die Dimension  $h = 0$ :

$$(S_{a,d,h}) = \begin{vmatrix} x_{00} & x_{01} & x_{02} & x_{03} \\ x_{10} & x_{11} & x_{12} & x_{13} \\ x_{20} & x_{21} & x_{22} & x_{23} \\ x_{30} & x_{31} & x_{32} & x_{33} \end{vmatrix} \Rightarrow \begin{vmatrix} x_{00} & x_{01} & x_{02} & x_{03} \\ x_{13} & x_{10} & x_{11} & x_{12} \\ x_{22} & x_{23} & x_{20} & x_{21} \\ x_{31} & x_{32} & x_{33} & x_{30} \end{vmatrix}$$

## 3.3 Das bijektive nicht lineare mapping $\Omega_l$

Die bijektive nicht lineare mapping Funktion  $\Omega_l$  implementiert die S-Box Layer. Hier wird jedes Element  $s_{a,d,h} \in V_8$ , wobei  $r = 8, s = 8, h = 1$ , der State-Matrix mit  $\beta_r \bmod 4(s_{a,d,h})$ , wo  $\beta_b : V_8 \mapsto V_8, b \in \{0, 1, 2, 3\}$  substituiert.  $\beta_b$  sind Substitutionsboxen, die unten angegeben sind. Zum Beispiel wird  $s_{a,d,h} = 0x33$  zu  $\beta_0(0x33) = 0xf7$  bei  $\beta_0$ . Es können auch andere S-Boxen benutzt werden, solange sie sicher sind und in der beschriebenen Fuktionsweise funktionieren. Die angegebenen S-Boxen  $\beta_b$  wurden mit Hilfe des Papers [2] generiert.

## 3.4 Die vertikale Permutation $\Gamma_l$

Die vertikale Permutation  $\Gamma_l$  ist eine vertikale down shift Operation, die jede Spalte der drei dimensionalen State-Matrix  $S = (s_{a,d,h})$ ,  $r = 4, s = 4, h = \overline{0,3}$ , um  $\zeta_s$  Positionen in einer Spalte  $s_h$  nach unten bewegt.  $\zeta_s$  ist abhängig von der Spaltennummer  $s_h \in \{0, 1, 2, 3\}$ , die Blockgröße  $l$  und kann mit der Formel  $\zeta_s = \frac{s \times l}{512}$  beschrieben werden. Jede Spalte in jeder der vier Dimensionen wird um die Anzahl der der Spalten nach unten verschoben. So wird zum Beispiel jedes Element der Spalte  $s_h = 3$  (in der vierten Spalte) um 3 Positionen nach unten verschoben. Die Elemente, die unten aus der Spalte gehen, werden wieder oben angehängt. Dieser Prozess wird für jeder der  $h = \overline{0,3}$  Dimensionen durchgeführt.

Ein Beispiel für die Dimension  $h = 0$ :

$$(S_{a,d,h}) = \begin{vmatrix} x_{00} & x_{01} & x_{02} & x_{03} \\ x_{10} & x_{11} & x_{12} & x_{13} \\ x_{20} & x_{21} & x_{22} & x_{23} \\ x_{30} & x_{31} & x_{32} & x_{33} \end{vmatrix} \Rightarrow \begin{vmatrix} x_{00} & x_{31} & x_{22} & x_{13} \\ x_{10} & x_{01} & x_{32} & x_{23} \\ x_{20} & x_{11} & x_{02} & x_{33} \\ x_{30} & x_{21} & x_{12} & x_{03} \end{vmatrix}$$

### 3.5 Die lineare Transformation $\pi_l$

In der linearen Transformation  $\pi_l$  wird jedes Element  $s_{a,d,h} \in V_8$  der State-Matrix  $S$ , wobei  $r = 8, s = 8; h = 1$  ist, als ein Element des endlichen Feldes  $GF(2^8)$  mit dem irreduziblen Polynom  $\eta(x) = x^8 + x^4 + x^3 + x^2 + 1$  dargestellt. Jedes neue Element der neuen resultierenden Matrix  $T = (t_{a,d})$  wird in dem  $GF(2^8)$  mit der folgenden Gleichung berechnet:

$$(t_{a,d}) = (q \ggg a) \otimes S_d$$

$Q$  ist hier die MDS-Matrix (maximum distance separable)<sup>1</sup>  $q = (0x08, 0x06, 0x07, 0x04, 0x01, 0x01, 0x05, 0x01)$ , die eine Matrix ist, mit bestimmten MDS Eigenschaften, die die Diffusion des Algorithmus stärkt.  $S_d$  ist die d. Spalte der  $8 \times 8$  State-Matrix  $S = (s_{a,d,h}), r = 8, s = 8, h = 1$ . Der Vektor  $q$  besteht aus Elementen des endlichen Feldes  $GF(2^8)$ , die in jeder Reihe um  $a$  Einheiten, die Reihenzahl nach rechts verschoben werden. Am Ende der Transformation  $\pi_l$  resultiert eine neue  $8 \times 8$  State-Matrix.

### 3.6 Die dimensionale Permutation $\chi_l^{(h)}$

In der dimensionalen Permutation  $\chi_l^{(h)}$ , abhängig vom Parameter  $h \in \{0, 1, 2, 3\}$ , wird die Dimension  $h$  in der  $4 \times 4 \times 4$  State-Matrix  $S = (s_{a,d,h}), r = 4, s = 4, h = \overline{0,3}$ , einmal um 90 Grad nach links gedreht. So wird die Reihe  $S_{a,h}$  zur Spalte  $S_{d,h}$ , wobei  $S_{a_0,h}$  nach  $S_{d_3,h}$  verschoben wird. Bei dem konstanten Packet  $\mu$  wird die Dimension  $h$  mit  $h = \frac{t}{2}$  berechnet, bei dem dann die dimensionale Permutation angewendet wird. Bei dem variablen Packet  $\lambda$  wird die Dimension  $h$  mit  $h = \frac{t+1}{2}$  berechnet. Durch diese Gleichungen werden nicht immer die gleichen Dimensionen im Cipher-State  $S$  permutiert. Ein Beispiel für die Dimension  $h = 0$ :

$$(S_{a,d,h}) = \begin{vmatrix} x_{00} & x_{01} & x_{02} & x_{03} \\ x_{10} & x_{11} & x_{12} & x_{13} \\ x_{20} & x_{21} & x_{22} & x_{23} \\ x_{30} & x_{31} & x_{32} & x_{33} \end{vmatrix} \Rightarrow \begin{vmatrix} x_{03} & x_{13} & x_{23} & x_{33} \\ x_{02} & x_{12} & x_{22} & x_{32} \\ x_{01} & x_{11} & x_{21} & x_{31} \\ x_{00} & x_{10} & x_{20} & x_{30} \end{vmatrix}$$

---

<sup>1</sup>Eine MDS-Matrix ist eine Matrix, die spezielle Eigenschaften der Diffusion besitzt.

### 3.7 Die Modulo 2 Addition (XOR-Operation) $\kappa_l^{(K_z)}$

Die Funktion  $\kappa_l^{(K_z)}$ , welche abhänngig vom Parameter  $K_z \in V_l$  ist, hat als Argument die State-Matrix  $S$ ,  $x \in V_l$ . Der Schlüssel  $K_z$ , wo  $z \in \{0, 1, 2, \dots, 35\}$  die aktuelle Runde ist, wird wie die State-Matrix  $S$ , in einer Matrix der Größe  $8 \times 8$  dargestellt. Dann wird der Schlüssel mit der State-Matrix mit Hilfe der XOR-Operation addiert. Das Ergebnis ist eine State-Matrix der Größe  $8 \times 8$ , mit der dann weitere Funktionen ausgeführt werden.

# 4 Entschlüsselung

## 4.1 Algorithmus

Der Entschlüsselungsalgorithmus  $D_{l,k}^{(K_k)}$  ist wie folgt definiert:

$$D_{l,k}^{(K_k)} = \prod_{i=\sqrt{n}-1}^0 \prod_{t=\sqrt{n}-1}^0 (\hat{\vartheta}_l \odot \hat{\Omega}_l \odot \hat{\Gamma}_l \odot \hat{\pi}_l \odot \hat{\chi}_l^{(\frac{t}{2})} \odot \kappa_l^{(K_z)}) || (\hat{\vartheta}_l | \hat{\Omega}_l | \hat{\Gamma}_l | \hat{\pi}_l | \hat{\chi}_l^{(\frac{t+1}{2})} | \kappa_l^{(K_z)}) \Psi^{(K_z)} \odot \kappa_l^{(K_z)}),$$

wo  $K_k$  der Schlüssel mit der Länge  $k$  ist,

- $\hat{\vartheta}_l$  - die Inverse der horizontalen Permutation der Elemente  $(s_{a,d,h})$ , wo  $(s_{a,d,h}) \in GF(2^8)$ ,  $a = \overline{0,3}$ ,  $d = \overline{0,3}$  und  $h = \overline{0,3}$ , mit dem Cipher-State  $S$ ,
- $\hat{\Omega}_l$  - das bijektive nicht lineare mapping der S-Boxen  $-\beta_b$ ,  $b \in \{0, 1, 2, 3\}$  mit den State-Matrix Vektoren,
- $\hat{\Gamma}_l$  - die Inverse der vertikalen Permutation der Elemente  $(s_{a,d,h})$ , wo  $(s_{a,d,h}) \in GF(2^8)$ ,  $a = \overline{0,3}$ ,  $d = \overline{0,3}$  und  $h = \overline{0,3}$ , mit dem dem Cipher-State  $S$ ,
- $\hat{\pi}_l$  - die Inverse der linearen Transformation des Cipher-State über das endliche Feld  $GF(2^8)$ ,
- $\hat{\chi}_l^{(\frac{t+1}{2})}$  - die Inverse der dimensionalen Permutation der Elemente  $(s_{a,d,h})$ , wo  $(s_{a,d,h}) \in GF(2^8)$ ,  $a = \overline{0,3}$ ,  $d = \overline{0,3}$  und  $h = \overline{0,3}$ , mit dem Cipher-State  $S$ , bei  $h = \frac{t+1}{2}$ ,
- $\hat{\chi}_l^{(\frac{t}{2})}$  - die Inverse der dimensionalen Permutation der Elemente  $(s_{a,d,h})$ , wo  $(s_{a,d,h}) \in GF(2^8)$ ,  $a = \overline{0,3}$ ,  $d = \overline{0,3}$  und  $h = \overline{0,3}$ , mit dem Cipher-State  $S$ , bei  $h = \frac{h}{2}$ ,
- $\kappa_l^{(K_z)}$  - eine Modulo 2 Addition (XOR-Operation) mit den Rundenschlüssel  $K_l^{(K_z)}$ ,  $z = i \times 6 + t$  und mit der State-Matrix ist .

Wie bei der Verschlüsselung, wird bei den Funktionen  $\hat{\vartheta}_l$ ,  $\hat{\Gamma}_l$ ,  $\hat{\chi}_l^{((t+1)/2)}$  und  $\hat{\chi}_l^{(t/2)}$ , mit dem Input  $x \in V_l$ , die Inverse der Permutationen in einer dreidimensionalen  $4 \times 4 \times 4$  State-Matrix ausgeführt ( $r = 4, s = 4, h = \overline{0,3}$ ), um eine bessere Obfuscation zu erzielen. Ansonsten wird immer eine zweidimensionale  $8 \times 8$  State-Matrix benutzt ( $r = 8, s =$

$8, h = 1$ ). Als Rückgabe aller Funktionen wird eine zweidimensionale  $8 \times 8$  State-Matirx ausgegeben.

## 4.2 Die Inverse der horizontalen Permutation $\hat{\vartheta}_l$

Die Inverse der horizontalen Permutation  $\hat{\vartheta}_l$  ist eine horizontale links shift Operation, die jede Reihe der drei dimensionalen State-Matrix  $S = (s_{a,d,h})$ ,  $r = 4, s = 4, h = \overline{0,3}$ , um  $\zeta_r$  Positionen in einer Reihe  $r_h$  nach links bewegt.  $\zeta_r$  ist abhängig von der Reihennummer  $r_h \in \{0, 1, 2, 3\}$ , die Blockgröße  $l$  und kann mit der Formel  $\zeta_r = \frac{r \times l}{512}$  berechnet werden. So wird jede Reihe in jeder der vier Dimensionen um die Anzahl der Reihenzahl der Reihe nach links verschoben. Zum Beispiel wird jedes Element der Reihe  $r_h = 2$  (in der dritten Reihe) um 2 Positionen nach links verschoben. Die Elemente, die links aus der Reihe gehen, werden wieder rechts angehängt. Dieser Prozess wird für jeder der  $h = \overline{0,3}$  Dimensionen durchgeführt.

Ein Beispiel für die Dimension  $h = 0$ :

$$(S_{a,d,h}) = \begin{vmatrix} x_{00} & x_{01} & x_{02} & x_{03} \\ x_{10} & x_{11} & x_{12} & x_{13} \\ x_{20} & x_{21} & x_{22} & x_{23} \\ x_{30} & x_{31} & x_{32} & x_{33} \end{vmatrix} \Rightarrow \begin{vmatrix} x_{00} & x_{01} & x_{02} & x_{03} \\ x_{11} & x_{12} & x_{13} & x_{10} \\ x_{22} & x_{23} & x_{20} & x_{21} \\ x_{33} & x_{30} & x_{31} & x_{32} \end{vmatrix}$$

## 4.3 Das bijektive nicht lineare mapping $\hat{\Omega}_l$

Die bijektive nicht lineare mapping Funktion  $\hat{\Omega}_l$  ist die Inverse der S-Box Layer. Hier wird jedes Element  $s_{a,d,h} \in V_8$ , wobei  $r = 8, s = 8, h = 1$ , der State-Matrix mit  $-\beta_r \bmod 4(s_{a,d,h})$ , wo  $-\beta_b : V_8 \mapsto V_8, b \in \{0, 1, 2, 3\}$  substituiert.  $-\beta_b$  sind die inversen Substitutionsboxen, die unten angegeben sind. Zum Beispiel wird  $s_{a,d,h} = 0xf7$  zu  $-\beta_0(0xf7) = 0x33$  bei  $-\beta_0$ . Es können auch andere S-Boxen benutzt werden, solange sie die korrekten Inversen der S-Boxen sind.

## 4.4 Die Inverse der vertikalen Permutation $\hat{\Gamma}_l$

Die Inverse der vertikalen Permutation  $\hat{\Gamma}_l$  ist eine vertikale up shift Operation, die jede Spalte der drei dimensionalen State-Matrix  $S = (s_{a,d,h})$ ,  $r = 4, s = 4, h = \overline{0,3}$ , um  $\zeta_s$  Positionen in einer Spalte  $s_h$  nach oben verschiebt.  $\zeta_s$  ist abhängig von der Spaltennummer  $s_h \in \{0, 1, 2, 3\}$ , die Blockgröße  $l$  und kann mit der Formel  $\zeta_s = \frac{s \times l}{512}$  beschrieben werden. So wird jede Spalte in jeder der drei Dimensionen um die Anzahl der Spaltenzahl der Spalten nach oben verschoben. Zum Beispiel wird jedes Element der Spalte  $s_h = 3$  (in der vierten Spalte) um 3 Positionen nach oben verschoben. Die Elemente, die oben aus

der Spalte gehen, werden wieder unten angehangen. Dieser Prozess wird für jede der  $h = \overline{0, 3}$  Dimensionen durchgeführt.

Ein Beispiel für die Dimension  $h = 0$ :

$$(S_{a,d,h}) = \begin{vmatrix} x_{00} & x_{01} & x_{02} & x_{03} \\ x_{10} & x_{11} & x_{12} & x_{13} \\ x_{20} & x_{21} & x_{22} & x_{23} \\ x_{30} & x_{31} & x_{32} & x_{33} \end{vmatrix} \Rightarrow \begin{vmatrix} x_{00} & x_{11} & x_{22} & x_{33} \\ x_{10} & x_{21} & x_{32} & x_{03} \\ x_{20} & x_{31} & x_{02} & x_{13} \\ x_{30} & x_{01} & x_{12} & x_{23} \end{vmatrix}$$

## 4.5 Die Inverse der linearen Transformation $\hat{\pi}_l$

In der Inverse der linearen Transformation  $\hat{\pi}_l$  wird jedes Element  $s_{a,d,h} \in V_8$  der State-Matrix  $S$ , wobei  $r = 8, s = 8, h = 1$  ist, als ein Element des endlichen Feldes  $GF(2^8)$  mit dem irreduziblen Polynom  $\eta(x) = x^8 + x^4 + x^3 + x^2 + 1$  dargestellt. Jedes neue Element der neuen resultierenden Matrix  $\hat{T} = (\hat{t}_{a,d})$  wird in dem  $GF(2^8)$  mit der folgenden Gleichung berechnet:

$$(\hat{t}_{a,d}) = (\hat{q} \lll a) \otimes S_d$$

$\hat{Q}$  ist hier die Inverse MDS-Matrix  $\hat{q} = (0x2f, 0x49, 0xd7, 0xca, 0xad, 0x95, 0x76, 0xa8)$ , die auch die MDS Eigenschaften besitzt.  $S_d$  ist die d. Spalte der  $8 \times 8$  State-Matrix  $S = (s_{a,d,h}), r = 8, s = 8, h = 1$ . Der Vektor  $\hat{q}$  besteht nur aus Elementen des endlichen Feldes  $GF(2^8)$ , die in jeder Reihe um  $a$  Einheiten, die Reihenzahl nach rechts verschoben werden. Am Ende der Inverse resultiert eine neue  $8 \times 8$  State-Matrix.

## 4.6 Die Inverse der dimensionalen Permutation $\hat{\chi}_l^{(h)}$

In der Inverse der dimensionalen Permutation  $\hat{\chi}_l^{(h)}$ , abhängig vom Parameter  $h \in \{0, 1, 2, 3\}$ , wird die Dimension  $h$  in der  $4 \times 4 \times 4$  State-Matirx  $S = (s_{a,d,h}), r = 4, s = 4, h = \overline{0, 3}$ , zurück 90 Grad nach rechts gedreht. So wird wird die Spalte  $S_{d,h}$  zur Reihe  $S_{a,h}$ , wobei  $S_{d_0,h}$  nach  $S_{a_3,h}$  verschoben wird. Bei dem konstanten Packet  $\mu$ , wird die Dimension  $h$  mit  $h = \frac{t}{2}$  berechnet, bei der dann die Inverse der dimensionalen Permutation angewendet wird. Bei den variablen Packeten  $\lambda$  wird die Dimension  $h$  mit  $h = \frac{t+1}{2}$  berechnet. Durch diese Gleichungen wird die korrekte Inverse der dimensionalen Permutation berechnet. Ein Beispiel für die Dimension  $h = 0$ :

$$(S_{a,d,h}) = \begin{vmatrix} x_{00} & x_{01} & x_{02} & x_{03} \\ x_{10} & x_{11} & x_{12} & x_{13} \\ x_{20} & x_{21} & x_{22} & x_{23} \\ x_{30} & x_{31} & x_{32} & x_{33} \end{vmatrix} \Rightarrow \begin{vmatrix} x_{30} & x_{20} & x_{10} & x_{00} \\ x_{31} & x_{21} & x_{11} & x_{01} \\ x_{32} & x_{22} & x_{12} & x_{02} \\ x_{33} & x_{23} & x_{13} & x_{03} \end{vmatrix}$$

## 5 Rundenschlüssel Erzeugung

Die Rundenschlüssel  $K_z, z \in \{0, 1, 2, \dots, 35\}$  haben die gleiche Größe wie die Blocklänge  $l, l = k$ . In der Funktion  $\kappa_l^{(K_z)}$  werden die Rundenschlüssel in der  $8 \times 8$  State-Matrix  $S$  mit der XOR-Operation zusammenaddiert. Da es aber nicht sicher wäre, für jede Runde der  $n = 36$  Runden den gleichen Schlüssel zu benutzen, werden 36 Rundenschlüssel mit der folgenden Gleichung  $G^{(K_k)}$  erzeugt.  $G^{(K_k)}$  ist abhängig von dem Master-Schlüssel  $K_k$  und von dem temporären Schlüssel  $\alpha$ , der bei der ersten Runde  $\alpha = K_k$  ist, und bei den restlichen 35 Schlüssel,  $\alpha = K_{-z}$  ist, wobei  $K_{-z}$  der vorherige Rundenschlüssel ist. Somit ist  $G^{(K_k)}$ :

$$G^{(K_k)} = \hat{\kappa}_k^{(\alpha)} \odot F_k \odot \Gamma_k \odot \Omega_k^{(rh)} \odot \kappa_k^{(\alpha)},$$

wo  $K_k$  der Master-Schlüssel mit der Länge  $k$  ist,

$\kappa_k^{(K_z)}$  - eine Modulo 2 Addition (XOR-Operaton) mit dem Schlüssel  $\alpha$  und der RCON Konstanten  $m = 0xc6e8e5ed7b352d4$  ist,

$\kappa_k^{(\hat{K}_z)}$  - eine Modulo 2 Addition (XOR-Operaton) mit dem Schlüssel  $\alpha$  und den Rundenschlüssel  $K_z$  ist,

$\Omega_k^{(rh)}$  - das bijektive nicht lineare mapping der S-Boxen  $\beta_b, b \in \{0, 1, 2, 3\}$ , mit der konstanten Reihenfolge  $rh = \{1, 0, 3, 2, 3, 0, 1, 2\}$ , statt einer sequenziellen Reihenfolge wie bei der Verschlüsselung, mit den Rundenschlüssel  $K_z$  ist,

$\Gamma_k$  - die vertikale Permutation der Elemente  $(k_{a,d,h})$ , wo  $(k_{a,d,h}) \in GF(2^8)$ ,  $a = \overline{0,3}$ ,  $d = \overline{0,3}$  und  $h = \overline{0,3}$ , mit dem Rundenschlüssel  $K_z$  ist,

$F_k$  - die dimensionale Permutation der Elemente  $(k_{a,d,h})$ , wo  $(k_{a,d,h}) \in GF(2^8)$ ,  $a = \overline{0,3}$ ,  $d = \overline{0,3}$  und  $h = \overline{0,3}$ , mit dem Rundenschlüssel  $K_z$ , hier aber die Dimensionen h in der Reihenfolge,  $h = 2 \mapsto h = 3$  und  $h = 3 \mapsto h = 2$  gewechselt wird.

Die Funktion  $G^{(K_k)}$  muss dann  $n$  mal ausgeführt werden, damit man die benötigte Anzahl von Rundenschlüssel erzeugt. Es wurde bewusst ein recht leicht zu berechnener Algorithmus für die Rundenschlüssel-Erzeugung entwickelt, da dieser Algorithmus auf die nicht Vorhersehbarkeit des Master-Schlüssel  $K_k$  setzt und es somit nicht nötig und effizient wäre, einen komplexen Algorithmus für die Rundenschlüssel-Erzeugung zu entwickeln und anzuwenden.

## 6 Sequenz Shuffle Funktion $\Psi^{(K_z)}$

Da MEA nicht wie AES [1] ein konstantes kryptographisches Netzwerk wie das SPN [1] benutzt, muss eine dynamische Reihenfolge für jedes dynamische Packet generiert werden. Diese Reihenfolge ist abhängig vom Schlüssel  $K_k$ , doch sollte sie nicht vorhersehbar ohne den Schlüssel  $K_k$  sein. Aus diesem Grund wird der Permutations Algorithmus  $\Psi^{(K_k)}$  angewandt, der abhängig von den Rundenschlüsseln  $K_z$  ist, die vom Masterschlüssel  $K_k$  mit der vorher beschriebenen Funktion  $G^{(K_k)}$  generiert wurden. Zuerst werden  $\frac{n}{2}$  Arrays mit jeweils 6 Elementen generiert, die jeweils sequentiell mit 0 bis 5 aufgefüllt werden. Dies ist die Startreihenfolge  $Rt_{\sqrt{n}} \in R_{\frac{n}{2}}$ , wobei  $Rt_{\sqrt{n}}$  das t. Element von  $R_{\frac{n}{2}}$  ist.  $t \in \{0, 1, 2, \dots, \frac{n}{2} - 1\}$ .  $\Psi^{(K_z)}$  ist wie folgt definiert:

$$\Psi^{(K_z)} = \prod_{i=0}^{\sqrt{n}-1} \prod_{t=0}^{\frac{n}{2}} \prod_{z=0}^{\sqrt{n}-2} Rt_{(\iota_{K_{(i+t)}}^{(z)})} \doteq Rt_{(\iota_{K_{(i+t)}}^{(z+1)})},$$

wobei  $\iota_{K_{(i+t)}}^{(-z)}$  das x. Element von  $Rt_{\sqrt{n}}$  ist und mit Hilfe der Tabelle 2 in Abhängigkeit vom Rundenschlüssel  $K_{(i+t)}$  bestimmt wird. Bei der Funktion  $\iota_{K_{(i+t)}}^{(-z)}$  wird geschaut, ob die Nummer  $-y \in K_{[-z]}$ , im Rundenschlüssel  $K_z \in K_{z=i+t}$ , kleiner als ein Wert ist und dann mit Hilfe der Tabelle 2 den größtmöglichen Wert zugewiesen wird. Das Resultat wird dann als Rückgabe gegeben.

Wert	Resultat	Wert	Funktion
$-y \leq 0x2A$	0x00	0x00	$\vartheta_l$ oder $\hat{\vartheta}_l$ (bei $D_{l,k}^{(K_k)}$ )
$-y \leq 0x54$	0x01	0x01	$\Omega_l$ oder $\hat{\Omega}_l$ (bei $D_{l,k}^{(K_k)}$ )
$-y \leq 0x7E$	0x02	0x02	$\Gamma_l$ oder $\hat{\Gamma}_l$ (bei $D_{l,k}^{(K_k)}$ )
$-y \leq 0xA8$	0x03	0x03	$\pi_l$ oder $\hat{\pi}_l$ (bei $D_{l,k}^{(K_k)}$ )
$-y \leq 0xD2$	0x04	0x04	$\chi_l^{(h)}$ oder $\hat{\chi}_l^{(h)}$ (bei $D_{l,k}^{(K_k)}$ )
$-y \leq 0xFF$	0x05	0x05	$\kappa_l^{(K_z)}$

Tabelle 2 und Tabelle 3

Dieser Prozess wird dann  $\sqrt{n}$ -mal wiederholt, damit eine sichere Permutation entsteht. Diese Permutation wird dann als dynamische Reihenfolge  $\Psi^{(K_z)}$  benutzt, die mit Hilfe der Tabelle 3 die jeweiligen Funktionen in  $E_{l,k}^{(K_k)}$  und  $D_{l,k}^{(K_k)}$  ausführt.

# 7 Weiteres

## 7.1 S-Boxen $\beta_b$ und $-\beta_b$

Die 8 Substitutionsboxen sind wie folgt definiert:

$\beta_0 =$

{

0xce, 0xbb, 0xeb, 0x92, 0xea, 0xcb, 0x13, 0xc1,  
0xe9, 0x3a, 0xd6, 0xb2, 0xd2, 0x90, 0x17, 0xf8,  
0x42, 0x15, 0x56, 0xb4, 0x65, 0x1c, 0x88, 0x43,  
0xc5, 0x5c, 0x36, 0xba, 0xf5, 0x57, 0x67, 0x8d,  
0x31, 0xf6, 0x64, 0x58, 0x9e, 0xf4, 0x22, 0xaa,  
0x75, 0x0f, 0x02, 0xb1, 0xdf, 0x6d, 0x73, 0x4d,  
0x7c, 0x26, 0x2e, 0xf7, 0x08, 0x5d, 0x44, 0x3e,  
0x9f, 0x14, 0xc8, 0xae, 0x54, 0x10, 0xd8, 0xbc,  
0x1a, 0x6b, 0x69, 0xf3, 0xbd, 0x33, 0xab, 0xfa,  
0xd1, 0x9b, 0x68, 0x4e, 0x16, 0x95, 0x91, 0xee,  
0x4c, 0x63, 0x8e, 0x5b, 0xcc, 0x3c, 0x19, 0xa1,  
0x81, 0x49, 0x7b, 0xd9, 0x6f, 0x37, 0x60, 0xca,  
0xe7, 0x2b, 0x48, 0xfd, 0x96, 0x45, 0xfc, 0x41,  
0x12, 0x0d, 0x79, 0xe5, 0x89, 0x8c, 0xe3, 0x20,  
0x30, 0xdc, 0xb7, 0x6c, 0x4a, 0xb5, 0x3f, 0x97,  
0xd4, 0x62, 0x2d, 0x06, 0xa4, 0xa5, 0x83, 0x5f,  
0x2a, 0xda, 0xc9, 0x00, 0x7e, 0xa2, 0x55, 0xbf,  
0x11, 0xd5, 0x9c, 0xcf, 0x0e, 0x0a, 0x3d, 0x51,  
0x7d, 0x93, 0x1b, 0xfe, 0xc4, 0x47, 0x09, 0x86,  
0x0b, 0x8f, 0x9d, 0x6a, 0x07, 0xb9, 0xb0, 0x98,  
0x18, 0x32, 0x71, 0x4b, 0xef, 0x3b, 0x70, 0xa0,  
0xe4, 0x40, 0xff, 0xc3, 0xa9, 0xe6, 0x78, 0xf9,  
0x8b, 0x46, 0x80, 0x1e, 0x38, 0xe1, 0xb8, 0xa8,  
0xe0, 0x0c, 0x23, 0x76, 0x1d, 0x25, 0x24, 0x05,  
0xf1, 0x6e, 0x94, 0x28, 0x9a, 0x84, 0xe8, 0xa3,  
0x4f, 0x77, 0xd3, 0x85, 0xe2, 0x52, 0xf2, 0x82,  
0x50, 0x7a, 0x2f, 0x74, 0x53, 0xb3, 0x61, 0xaf,  
0x39, 0x35, 0xde, 0xcd, 0x1f, 0x99, 0xac, 0xad,

```
0x72, 0x2c, 0xdd, 0xd0, 0x87, 0xbe, 0x5e, 0xa6,
0xec, 0x04, 0xc6, 0x03, 0x34, 0xfb, 0xdb, 0x59,
0xb6, 0xc2, 0x01, 0xf0, 0x5a, 0xed, 0xa7, 0x66,
0x21, 0x7f, 0x8a, 0x27, 0xc7, 0xc0, 0x29, 0xd7
```

}

$\beta_1 =$

{

```
0x14, 0x9d, 0xb9, 0xe7, 0x67, 0x4c, 0x50, 0x82,
0xca, 0xe5, 0x1d, 0x31, 0x0a, 0xc6, 0xb2, 0x51,
0xa2, 0xd8, 0x54, 0x90, 0xd0, 0xce, 0x2d, 0x7d,
0xc7, 0x7e, 0xd7, 0x94, 0xdf, 0x83, 0x8e, 0x6c,
0x66, 0xd2, 0x6f, 0x16, 0x1e, 0x76, 0xfe, 0xcc,
0xaa, 0x5a, 0x8f, 0x17, 0xbd, 0x2c, 0xac, 0xea,
0x7b, 0x65, 0xa9, 0x10, 0xc0, 0x92, 0xee, 0xbe,
0x6a, 0x6e, 0x48, 0x96, 0x95, 0xe9, 0x32, 0xbc,
0xa1, 0x42, 0xd5, 0xa7, 0x81, 0xb4, 0x5f, 0xe6,
0xc2, 0x5d, 0xad, 0x3a, 0xb7, 0x0c, 0x8d, 0x01,
0x98, 0xfd, 0x12, 0x02, 0x75, 0x13, 0x0f, 0x6b,
0x22, 0xe2, 0xab, 0xf7, 0x7f, 0xba, 0x97, 0xd1,
0x64, 0xd9, 0xc4, 0x59, 0xaf, 0x23, 0x33, 0x37,
0xde, 0xae, 0x60, 0x05, 0x63, 0xa8, 0x52, 0xa5,
0x4e, 0xe0, 0xdd, 0x71, 0xf2, 0x24, 0x34, 0x57,
0x47, 0xa4, 0xb3, 0x9e, 0x2f, 0xc1, 0xb8, 0xcb,
0x2b, 0xd4, 0x0d, 0x36, 0x91, 0x8b, 0x9c, 0x26,
0x25, 0x61, 0xa3, 0xd6, 0xeb, 0x35, 0x53, 0xf4,
0x2e, 0x88, 0x80, 0xe4, 0x30, 0xdb, 0xfc, 0x0e,
0x77, 0x8c, 0x93, 0xa6, 0x78, 0x06, 0xe1, 0xec,
0xf9, 0x03, 0xa0, 0x27, 0xda, 0xef, 0x5c, 0x00,
0x7a, 0x45, 0xe8, 0x40, 0x1a, 0x4b, 0x5e, 0x73,
0xc3, 0xff, 0xf5, 0xf3, 0xb0, 0xc5, 0x49, 0x21,
0xfa, 0x11, 0x39, 0x84, 0x43, 0x38, 0x85, 0x07,
0xf0, 0x79, 0x46, 0xf8, 0xe3, 0x1f, 0x09, 0xb6,
0xcd, 0x55, 0x1c, 0x1b, 0xfb, 0x7c, 0xed, 0x6d,
0x15, 0x56, 0x86, 0x20, 0x68, 0x4a, 0x41, 0x4f,
0xd3, 0x99, 0x08, 0xf6, 0x3f, 0x89, 0x62, 0x04,
0xcf, 0xc8, 0x69, 0x9f, 0x19, 0x5b, 0x44, 0x9b,
0x87, 0xb1, 0x3d, 0xbb, 0xdc, 0x2a, 0xbf, 0x58,
0x3c, 0x8a, 0x18, 0x3e, 0x72, 0x0b, 0x28, 0x4d,
0xb5, 0x9a, 0xc9, 0x74, 0x29, 0xf1, 0x3b, 0x70
```

}

$\beta_2 =$

```
{
    0x68, 0x8d, 0xca, 0x4d, 0x73, 0x4b, 0x4e, 0x2a,
    0xd4, 0x52, 0x26, 0xb3, 0x54, 0x1e, 0x19, 0x1f,
    0x22, 0x03, 0x46, 0x3d, 0x2d, 0x4a, 0x53, 0x83,
    0x13, 0x8a, 0xb7, 0xd5, 0x25, 0x79, 0xf5, 0xbd,
    0x58, 0x2f, 0x0d, 0x02, 0xed, 0x51, 0x9e, 0x11,
    0xf2, 0x3e, 0x55, 0x5e, 0xd1, 0x16, 0x3c, 0x66,
    0x70, 0x5d, 0xf3, 0x45, 0x40, 0xcc, 0xe8, 0x94,
    0x56, 0x08, 0xce, 0x1a, 0x3a, 0xd2, 0xe1, 0xdf,
    0xb5, 0x38, 0x6e, 0x0e, 0xe5, 0xf4, 0xf9, 0x86,
    0xe9, 0x4f, 0xd6, 0x85, 0x23, 0xcf, 0x32, 0x99,
    0x31, 0x14, 0xae, 0xee, 0xc8, 0x48, 0xd3, 0x30,
    0xa1, 0x92, 0x41, 0xb1, 0x18, 0xc4, 0x2c, 0x71,
    0x72, 0x44, 0x15, 0xfd, 0x37, 0xbe, 0x5f, 0xaa,
    0x9b, 0x88, 0xd8, 0xab, 0x89, 0x9c, 0xfa, 0x60,
    0xea, 0xbc, 0x62, 0x0c, 0x24, 0xa6, 0xa8, 0xec,
    0x67, 0x20, 0xdb, 0x7c, 0x28, 0xdd, 0xac, 0x5b,
    0x34, 0x7e, 0x10, 0xf1, 0x7b, 0x8f, 0x63, 0xa0,
    0x05, 0x9a, 0x43, 0x77, 0x21, 0xbf, 0x27, 0x09,
    0xc3, 0x9f, 0xb6, 0xd7, 0x29, 0xc2, 0xeb, 0xc0,
    0xa4, 0x8b, 0x8c, 0x1d, 0xfb, 0xff, 0xc1, 0xb2,
    0x97, 0x2e, 0xf8, 0x65, 0xf6, 0x75, 0x07, 0x04,
    0x49, 0x33, 0xe4, 0xd9, 0xb9, 0xd0, 0x42, 0xc7,
    0x6c, 0x90, 0x00, 0x8e, 0x6f, 0x50, 0x01, 0xc5,
    0xda, 0x47, 0x3f, 0xcd, 0x69, 0xa2, 0xe2, 0x7a,
    0xa7, 0xc6, 0x93, 0x0f, 0xa, 0x06, 0xe6, 0x2b,
    0x96, 0xa3, 0x1c, 0xaf, 0x6a, 0x12, 0x84, 0x39,
    0xe7, 0xb0, 0x82, 0xf7, 0xfe, 0x9d, 0x87, 0x5c,
    0x81, 0x35, 0xde, 0xb4, 0xa5, 0xfc, 0x80, 0xef,
    0xcb, 0xbb, 0x6b, 0x76, 0xba, 0x5a, 0x7d, 0x78,
    0x0b, 0x95, 0xe3, 0xad, 0x74, 0x98, 0x3b, 0x36,
    0x64, 0x6d, 0xdc, 0xf0, 0x59, 0xa9, 0x4c, 0x17,
    0x7f, 0x91, 0xb8, 0xc9, 0x57, 0x1b, 0xe0, 0x61
}
```

$\beta_3 =$

```
{
    0xa8, 0x43, 0x5f, 0x06, 0x6b, 0x75, 0x6c, 0x59,
    0x71, 0xdf, 0x87, 0x95, 0x17, 0xf0, 0xd8, 0x09,
    0x6d, 0xf3, 0x1d, 0xcb, 0xc9, 0x4d, 0x2c, 0xaf,
    0x79, 0xe0, 0x97, 0xfd, 0x6f, 0x4b, 0x45, 0x39,
    0x3e, 0xdd, 0xa3, 0x4f, 0xb4, 0xb6, 0x9a, 0x0e,
```

```
0x1f, 0xbff, 0x15, 0xe1, 0x49, 0xd2, 0x93, 0xc6,
0x92, 0x72, 0x9e, 0x61, 0xd1, 0x63, 0xfa, 0xee,
0xf4, 0x19, 0xd5, 0xad, 0x58, 0xa4, 0xbb, 0xa1,
0xdc, 0xf2, 0x83, 0x37, 0x42, 0xe4, 0x7a, 0x32,
0x9c, 0xcc, 0xab, 0x4a, 0x8f, 0x6e, 0x04, 0x27,
0x2e, 0xe7, 0xe2, 0x5a, 0x96, 0x16, 0x23, 0x2b,
0xc2, 0x65, 0x66, 0x0f, 0xbc, 0xa9, 0x47, 0x41,
0x34, 0x48, 0xfc, 0xb7, 0x6a, 0x88, 0xa5, 0x53,
0x86, 0xf9, 0x5b, 0xdb, 0x38, 0x7b, 0xc3, 0x1e,
0x22, 0x33, 0x24, 0x28, 0x36, 0xc7, 0xb2, 0x3b,
0x8e, 0x77, 0xba, 0xf5, 0x14, 0x9f, 0x08, 0x55,
0x9b, 0x4c, 0xfe, 0x60, 0x5c, 0xda, 0x18, 0x46,
0xcd, 0x7d, 0x21, 0xb0, 0x3f, 0x1b, 0x89, 0xff,
0xeb, 0x84, 0x69, 0x3a, 0x9d, 0xd7, 0xd3, 0x70,
0x67, 0x40, 0xb5, 0xde, 0x5d, 0x30, 0x91, 0xb1,
0x78, 0x11, 0x01, 0xe5, 0x00, 0x68, 0x98, 0xa0,
0xc5, 0x02, 0xa6, 0x74, 0x2d, 0x0b, 0xa2, 0x76,
0xb3, 0xbe, 0xce, 0xbd, 0xae, 0xe9, 0x8a, 0x31,
0x1c, 0xec, 0xf1, 0x99, 0x94, 0xaa, 0xf6, 0x26,
0x2f, 0xef, 0xe8, 0x8c, 0x35, 0x03, 0xd4, 0x7f,
0xfb, 0x05, 0xc1, 0x5e, 0x90, 0x20, 0x3d, 0x82,
0xf7, 0xea, 0xa0, 0x0d, 0x7e, 0xf8, 0x50, 0x1a,
0xc4, 0x07, 0x57, 0xb8, 0x3c, 0x62, 0xe3, 0xc8,
0xac, 0x52, 0x64, 0x10, 0xd0, 0xd9, 0x13, 0x0c,
0x12, 0x29, 0x51, 0xb9, 0xcf, 0xd6, 0x73, 0x8d,
0x81, 0x54, 0xc0, 0xed, 0x4e, 0x44, 0xa7, 0x2a,
0x85, 0x25, 0xe6, 0xca, 0x7c, 0x8b, 0x56, 0x80
}


$$-\beta_0 =$$


{

  0x83, 0xf2, 0x2a, 0xeb, 0xe9, 0xbff, 0x7b, 0x9c,
  0x34, 0x96, 0x8d, 0x98, 0xb9, 0x69, 0x8c, 0x29,
  0x3d, 0x88, 0x68, 0x06, 0x39, 0x11, 0x4c, 0x0e,
  0xa0, 0x56, 0x40, 0x92, 0x15, 0xbc, 0xb3, 0xdc,
  0x6f, 0xf8, 0x26, 0xba, 0xbe, 0xbd, 0x31, 0xfb,
  0xc3, 0xfe, 0x80, 0x61, 0xe1, 0x7a, 0x32, 0xd2,
  0x70, 0x20, 0xa1, 0x45, 0xec, 0xd9, 0x1a, 0x5d,
  0xb4, 0xd8, 0x09, 0xa5, 0x55, 0x8e, 0x37, 0x76,
  0xa9, 0x67, 0x10, 0x17, 0x36, 0x65, 0xb1, 0x95,
  0x62, 0x59, 0x74, 0xa3, 0x50, 0x2f, 0x4b, 0xc8,
  0xd0, 0x8f, 0xcd, 0xd4, 0x3c, 0x86, 0x12, 0x1d,
```

```
0x23, 0xef, 0xf4, 0x53, 0x19, 0x35, 0xe6, 0x7f,  
0x5e, 0xd6, 0x79, 0x51, 0x22, 0x14, 0xf7, 0x1e,  
0x4a, 0x42, 0x9b, 0x41, 0x73, 0x2d, 0xc1, 0x5c,  
0xa6, 0xa2, 0xe0, 0x2e, 0xd3, 0x28, 0xbb, 0xc9,  
0xae, 0x6a, 0xd1, 0x5a, 0x30, 0x90, 0x84, 0xf9,  
0xb2, 0x58, 0xcf, 0x7e, 0xc5, 0xcb, 0x97, 0xe4,  
0x16, 0x6c, 0xfa, 0xb0, 0x6d, 0x1f, 0x52, 0x99,  
0x0d, 0x4e, 0x03, 0x91, 0xc2, 0x4d, 0x64, 0x77,  
0x9f, 0xdd, 0xc4, 0x49, 0x8a, 0x9a, 0x24, 0x38,  
0xa7, 0x57, 0x85, 0xc7, 0x7c, 0x7d, 0xe7, 0xf6,  
0xb7, 0xac, 0x27, 0x46, 0xde, 0xdf, 0x3b, 0xd7,  
0x9e, 0x2b, 0x0b, 0xd5, 0x13, 0x75, 0xf0, 0x72,  
0xb6, 0x9d, 0x1b, 0x01, 0x3f, 0x44, 0xe5, 0x87,  
0xfd, 0x07, 0xf1, 0xab, 0x94, 0x18, 0xea, 0xfc,  
0x3a, 0x82, 0x5f, 0x05, 0x54, 0xdb, 0x00, 0x8b,  
0xe3, 0x48, 0x0c, 0xca, 0x78, 0x89, 0xa, 0xff,  
0x3e, 0x5b, 0x81, 0xee, 0x71, 0xe2, 0xda, 0x2c,  
0xb8, 0xb5, 0xcc, 0x6e, 0xa8, 0x6b, 0xad, 0x60,  
0xc6, 0x08, 0x04, 0x02, 0xe8, 0xf5, 0x4f, 0xa4,  
0xf3, 0xc0, 0xce, 0x43, 0x25, 0x1c, 0x21, 0x33,  
0x0f, 0xaf, 0x47, 0xed, 0x66, 0x63, 0x93, 0xaa
```

```
}
```

$-\beta_1 =$

```
{
```

```
0xa7, 0x4f, 0x53, 0xa1, 0xdf, 0x6b, 0x9d, 0xbff,  
0xda, 0xc6, 0x0c, 0xf5, 0x4d, 0x82, 0x97, 0x56,  
0x33, 0xb9, 0x52, 0x55, 0x00, 0xd0, 0x23, 0x2b,  
0xf2, 0xe4, 0xac, 0xcb, 0xca, 0xa, 0x24, 0xc5,  
0xd3, 0xb7, 0x58, 0x65, 0x75, 0x88, 0x87, 0xa3,  
0xf6, 0xfc, 0xed, 0x80, 0x2d, 0x16, 0x90, 0x7c,  
0x94, 0x0b, 0x3e, 0x66, 0x76, 0x8d, 0x83, 0x67,  
0xbd, 0xba, 0x4b, 0xfe, 0xf0, 0xea, 0xf3, 0xdc,  
0xab, 0xd6, 0x41, 0xbc, 0xe6, 0xa9, 0xc2, 0x78,  
0x3a, 0xb6, 0xd5, 0xad, 0x05, 0xf7, 0x70, 0xd7,  
0x06, 0x0f, 0x6e, 0x8e, 0x12, 0xc9, 0xd1, 0x77,  
0xef, 0x63, 0x29, 0xe5, 0xa6, 0x49, 0xae, 0x46,  
0x6a, 0x89, 0xde, 0x6c, 0x60, 0x31, 0x20, 0x04,  
0xd4, 0xe2, 0x38, 0x57, 0x1f, 0xcf, 0x39, 0x22,  
0xff, 0x73, 0xf4, 0xaf, 0xfb, 0x54, 0x25, 0x98,  
0x9c, 0xc1, 0xa8, 0x30, 0xcd, 0x17, 0x19, 0x5c,  
0x92, 0x44, 0x07, 0x1d, 0xbb, 0xbe, 0xd2, 0xe8,
```

```
0x91, 0xdd, 0xf1, 0x85, 0x99, 0x4e, 0x1e, 0x2a,  
0x13, 0x84, 0x35, 0x9a, 0x1b, 0x3c, 0x3b, 0x5e,  
0x50, 0xd9, 0xf9, 0xe7, 0x86, 0x01, 0x7b, 0xe3,  
0xa2, 0x40, 0x10, 0x8a, 0x79, 0x6f, 0x9b, 0x43,  
0x6d, 0x32, 0x28, 0x5a, 0x2e, 0x4a, 0x69, 0x64,  
0xb4, 0xe9, 0x0e, 0x7a, 0x45, 0xf8, 0xc7, 0x4c,  
0x7e, 0x02, 0x5d, 0xeb, 0x3f, 0x2c, 0x37, 0xee,  
0x34, 0x7d, 0x48, 0xb0, 0x62, 0xb5, 0x0d, 0x18,  
0xe1, 0xfa, 0x08, 0x7f, 0x27, 0xc8, 0x15, 0xe0,  
0x14, 0x5f, 0x21, 0xd8, 0x81, 0x42, 0x8b, 0x1a,  
0x11, 0x61, 0xa4, 0x95, 0xec, 0x72, 0x68, 0x1c,  
0x71, 0x9e, 0x59, 0xc4, 0x93, 0x09, 0x47, 0x03,  
0xaa, 0x3d, 0x2f, 0x8c, 0x9f, 0xce, 0x36, 0xa5,  
0xc0, 0xfd, 0x74, 0xb3, 0x8f, 0xb2, 0xdb, 0x5b,  
0xc3, 0xa0, 0xb8, 0xcc, 0x96, 0x51, 0x26, 0xb1
```

```
}
```

$-\beta_2 =$

```
{
```

```
0xb2, 0xb6, 0x23, 0x11, 0xa7, 0x88, 0xc5, 0xa6,  
0x39, 0x8f, 0xc4, 0xe8, 0x73, 0x22, 0x43, 0xc3,  
0x82, 0x27, 0xcd, 0x18, 0x51, 0x62, 0x2d, 0xf7,  
0x5c, 0x0e, 0x3b, 0xfd, 0xca, 0x9b, 0x0d, 0x0f,  
0x79, 0x8c, 0x10, 0x4c, 0x74, 0x1c, 0x0a, 0x8e,  
0x7c, 0x94, 0x07, 0xc7, 0x5e, 0x14, 0xa1, 0x21,  
0x57, 0x50, 0x4e, 0xa9, 0x80, 0xd9, 0xef, 0x64,  
0x41, 0xcf, 0x3c, 0xee, 0x2e, 0x13, 0x29, 0xba,  
0x34, 0x5a, 0xae, 0x8a, 0x61, 0x33, 0x12, 0xb9,  
0x55, 0xa8, 0x15, 0x05, 0xf6, 0x03, 0x06, 0x49,  
0xb5, 0x25, 0x09, 0x16, 0x0c, 0x2a, 0x38, 0xfc,  
0x20, 0xf4, 0xe5, 0x7f, 0xd7, 0x31, 0x2b, 0x66,  
0x6f, 0xff, 0x72, 0x86, 0xf0, 0xa3, 0x2f, 0x78,  
0x00, 0xbc, 0xcc, 0xe2, 0xb0, 0xf1, 0x42, 0xb4,  
0x30, 0x5f, 0x60, 0x04, 0xec, 0xa5, 0xe3, 0x8b,  
0xe7, 0x1d, 0xbf, 0x84, 0x7b, 0xe6, 0x81, 0xf8,  
0xde, 0xd8, 0xd2, 0x17, 0xce, 0x4b, 0x47, 0xd6,  
0x69, 0x6c, 0x19, 0x99, 0x9a, 0x01, 0xb3, 0x85,  
0xb1, 0xf9, 0x59, 0xc2, 0x37, 0xe9, 0xc8, 0xa0,  
0xed, 0x4f, 0x89, 0x68, 0x6d, 0xd5, 0x26, 0x91,  
0x87, 0x58, 0xbd, 0xc9, 0x98, 0xdc, 0x75, 0xc0,  
0x76, 0xf5, 0x67, 0x6b, 0x7e, 0xeb, 0x52, 0xcb,  
0xd1, 0x5b, 0x9f, 0x0b, 0xdb, 0x40, 0x92, 0x1a,
```

```

0xfa, 0xac, 0xe4, 0xe1, 0x71, 0x1f, 0x65, 0x8d,
0x97, 0x9e, 0x95, 0x90, 0x5d, 0xb7, 0xc1, 0xaf,
0x54, 0xfb, 0x02, 0xe0, 0x35, 0xbb, 0x3a, 0x4d,
0xad, 0x2c, 0x3d, 0x56, 0x08, 0x1b, 0x4a, 0x93,
0x6a, 0xab, 0xb8, 0x7a, 0xf2, 0x7d, 0xda, 0x3f,
0xfe, 0x3e, 0xbe, 0xea, 0xaa, 0x44, 0xc6, 0xd0,
0x36, 0x48, 0x70, 0x96, 0x77, 0x24, 0x53, 0xdf,
0xf3, 0x83, 0x28, 0x32, 0x45, 0x1e, 0xa4, 0xd3,
0xa2, 0x46, 0x6e, 0x9c, 0xdd, 0x63, 0xd4, 0x9d
}
```

$-\beta_3 =$

{

```

0xa4, 0xa2, 0xa9, 0xc5, 0x4e, 0xc9, 0x03, 0xd9,
0x7e, 0x0f, 0xd2, 0xad, 0xe7, 0xd3, 0x27, 0x5b,
0xe3, 0xa1, 0xe8, 0xe6, 0x7c, 0x2a, 0x55, 0x0c,
0x86, 0x39, 0xd7, 0x8d, 0xb8, 0x12, 0x6f, 0x28,
0xcd, 0x8a, 0x70, 0x56, 0x72, 0xf9, 0xbf, 0x4f,
0x73, 0xe9, 0xf7, 0x57, 0x16, 0xac, 0x50, 0xc0,
0x9d, 0xb7, 0x47, 0x71, 0x60, 0xc4, 0x74, 0x43,
0x6c, 0x1f, 0x93, 0x77, 0xdc, 0xce, 0x20, 0x8c,
0x99, 0x5f, 0x44, 0x01, 0xf5, 0x1e, 0x87, 0x5e,
0x61, 0x2c, 0x4b, 0x1d, 0x81, 0x15, 0xf4, 0x23,
0xd6, 0xea, 0xe1, 0x67, 0xf1, 0x7f, 0xfe, 0xda,
0x3c, 0x07, 0x53, 0x6a, 0x84, 0x9c, 0xcb, 0x02,
0x83, 0x33, 0xdd, 0x35, 0xe2, 0x59, 0x5a, 0x98,
0xa5, 0x92, 0x64, 0x04, 0x06, 0x10, 0x4d, 0x1c,
0x97, 0x08, 0x31, 0xee, 0xab, 0x05, 0xaf, 0x79,
0xa0, 0x18, 0x46, 0x6d, 0xfc, 0x89, 0xd4, 0xc7,
0xff, 0xf0, 0xcf, 0x42, 0x91, 0xf8, 0x68, 0xa,
0x65, 0x8e, 0xb6, 0xfd, 0xc3, 0xef, 0x78, 0x4c,
0xcc, 0x9e, 0x30, 0x2e, 0xbc, 0x0b, 0x54, 0x1a,
0xa6, 0xbb, 0x26, 0x80, 0x48, 0x94, 0x32, 0x7d,
0xa7, 0x3f, 0xae, 0x22, 0x3d, 0x66, 0xaa, 0xf6,
0x00, 0x5d, 0xbd, 0x4a, 0xe0, 0x3b, 0xb4, 0x17,
0x8b, 0x9f, 0x76, 0xb0, 0x24, 0x9a, 0x25, 0x63,
0xdb, 0xeb, 0x7a, 0x3e, 0x5c, 0xb3, 0xb1, 0x29,
0xf2, 0xca, 0x58, 0x6e, 0xd8, 0xa8, 0x2f, 0x75,
0xdf, 0x14, 0xfb, 0x13, 0x49, 0x88, 0xb2, 0xec,
0xe4, 0x34, 0x2d, 0x96, 0xc6, 0x3a, 0xed, 0x95,
0x0e, 0xe5, 0x85, 0x6b, 0x40, 0x21, 0x9b, 0x09,
0x19, 0x2b, 0x52, 0xde, 0x45, 0xa3, 0xfa, 0x51,
```

```
0xc2, 0xb5, 0xd1, 0x90, 0xb9, 0xf3, 0x37, 0xc1,  
0x0d, 0xba, 0x41, 0x11, 0x38, 0x7b, 0xbe, 0xd0,  
0xd5, 0x69, 0x36, 0xc8, 0x62, 0x1b, 0x82, 0x8f  
}
```

# 8 Implementierung in der Programmiersprache C

## 8.1 Code

### 8.1.1 mea.h

```
1 /*  
2  * Projekt : MEA  
3  * Autor : Michael Engel  
4  * Datei : mea.h  
5 */  
6  
7 #ifndef MEA_H  
8 #define MEA_H  
9  
10 #include <stdint.h>  
11 #include <stddef.h>  
12  
13 #define MEA_SUB_ROUNDS 0x06  
14 #define MEA_M_ROUNDS 0x06  
15  
16 #define MEA_NW_STATE 0x08  
17 #define MEA_NW_KEY 0x08  
18  
19 #define MEA_MS_IN_DIM 0x10  
20 #define MEA_MS_DIM 0x03  
21 #define MEA_MS_ROW 0x04  
22  
23 #define MEA_FNC_HRSR 0x00  
24 #define MEA_FNC_SBB 0x01  
25 #define MEA_FNC_VRSC 0x02  
26 #define MEA_FNC_MXCL 0x03  
27 #define MEA_FNC_DRT 0x04  
28 #define MEA_FNC_XRK 0x05  
29  
30  
31 #define BYTE_TO_M_STATE(table, n_row, n_col) table[(n_row) + (n_col)*  
           sizeof(uint64_t)]
```

```

32 #define RKCON 0xc6e8e5ed7b352d4
33
34
35 struct mea_t {
36     uint64_t* m_state;
37     uint8_t **r_seq;
38     uint64_t** r_keys;
39 };
40 typedef struct mea_t mea_t;
41
42 mea_t* mea_init();
43 int mea_del(mea_t* mea_ctx);
44
45 int mea_dimRotate(mea_t *mea_ctx, uint8_t dim);
46 int mea_invDimRotate(mea_t *mea_ctx, uint8_t dim);
47
48 int mea_verShiftColumns(mea_t *mea_ctx);
49 int mea_invVerShiftColumns(mea_t *mea_ctx);
50
51 int mea_horShiftRows(mea_t *mea_ctx);
52 int mea_invHorShiftRows(mea_t *mea_ctx);
53
54 int mea_mixColumns(mea_t *mea_ctx);
55 int mea_invMixColumns(mea_t *mea_ctx);
56
57 int mea_subBytes(mea_t *mea_ctx);
58 int mea_invSubBytes(mea_t *mea_ctx);
59
60 int mea_generateRKeys(mea_t *mea_ctx, uint64_t *mkey);
61 int mea_rSeqGen(mea_t *mea_ctx);
62
63 int mea_blockEncipher(mea_t *mea_ctx, uint64_t *plain, uint64_t *cipher
   );
64 int mea_blockDecipher(mea_t *mea_ctx, uint64_t *cipher, uint64_t *plain
   );
65
66 #endif

```

### 8.1.2 tables.h

```

1 /*
2  * Projekt : MEA
3  * Autor : Michael Engel
4  * Datei : tables.h
5 */
6
7 #ifndef TABLES_H
8 #define TABLES_H

```

```

9
10 #include <stdint.h>
11
12 extern uint8_t mds_matrix[8][8];
13 extern uint8_t mds_inv_matrix[8][8];
14
15 extern uint8_t mea_sbox[4][256];
16 extern uint8_t mea_invSbox[4][256];
17
18 #endif

```

### 8.1.3 tables.c

```

1 /*
2 * Projekt : MEA
3 * autor : Michael engel
4 * datei : tables.c
5 */
6
7 #include <stdint.h>
8
9 #include "mea.h"
10
11 uint8_t mds_matrix[8][8] = {
12     { 0x08, 0x06, 0x07, 0x04, 0x01, 0x01, 0x05, 0x01},
13     { 0x01, 0x08, 0x06, 0x07, 0x04, 0x01, 0x01, 0x05},
14     { 0x05, 0x01, 0x08, 0x06, 0x07, 0x04, 0x01, 0x01},
15     { 0x01, 0x05, 0x01, 0x08, 0x06, 0x07, 0x04, 0x01},
16     { 0x01, 0x01, 0x05, 0x01, 0x08, 0x06, 0x07, 0x04},
17     { 0x04, 0x01, 0x01, 0x05, 0x01, 0x08, 0x06, 0x07},
18     { 0x07, 0x04, 0x01, 0x01, 0x05, 0x01, 0x08, 0x06},
19     { 0x06, 0x07, 0x04, 0x01, 0x01, 0x05, 0x01, 0x08}
20 };
21
22 uint8_t mds_inv_matrix[8][8] = {
23     { 0x2f, 0x49, 0xd7, 0xca, 0xad, 0x95, 0x76, 0xa8},
24     { 0xa8, 0x2f, 0x49, 0xd7, 0xca, 0xad, 0x95, 0x76},
25     { 0x76, 0xa8, 0x2f, 0x49, 0xd7, 0xca, 0xad, 0x95},
26     { 0x95, 0x76, 0xa8, 0x2f, 0x49, 0xd7, 0xca, 0xad},
27     { 0xad, 0x95, 0x76, 0xa8, 0x2f, 0x49, 0xd7, 0xca},
28     { 0xca, 0xad, 0x95, 0x76, 0xa8, 0x2f, 0x49, 0xd7},
29     { 0xd7, 0xca, 0xad, 0x95, 0x76, 0xa8, 0x2f, 0x49},
30     { 0x49, 0xd7, 0xca, 0xad, 0x95, 0x76, 0xa8, 0x2f}
31 };
32
33 uint8_t mea_sbox[4][256] = {
34 {
35     0xce, 0xbb, 0xeb, 0x92, 0xea, 0xcb, 0x13, 0xc1, 0xe9, 0x3a, 0xd6, 0

```

```

xb2, 0xd2, 0x90, 0x17, 0xf8,
36   0x42, 0x15, 0x56, 0xb4, 0x65, 0x1c, 0x88, 0x43, 0xc5, 0x5c, 0x36, 0
xba, 0xf5, 0x57, 0x67, 0x8d,
37   0x31, 0xf6, 0x64, 0x58, 0x9e, 0xf4, 0x22, 0xaa, 0x75, 0x0f, 0x02, 0
xb1, 0xdf, 0x6d, 0x73, 0x4d,
38   0x7c, 0x26, 0x2e, 0xf7, 0x08, 0x5d, 0x44, 0x3e, 0x9f, 0x14, 0xc8, 0
xae, 0x54, 0x10, 0xd8, 0xbc,
39   0x1a, 0x6b, 0x69, 0xf3, 0xbd, 0x33, 0xab, 0xfa, 0xd1, 0x9b, 0x68, 0
x4e, 0x16, 0x95, 0x91, 0xee,
40   0x4c, 0x63, 0x8e, 0x5b, 0xcc, 0x3c, 0x19, 0xa1, 0x81, 0x49, 0x7b, 0
xd9, 0x6f, 0x37, 0x60, 0xca,
41   0xe7, 0x2b, 0x48, 0xfd, 0x96, 0x45, 0xfc, 0x41, 0x12, 0x0d, 0x79, 0
xe5, 0x89, 0x8c, 0xe3, 0x20,
42   0x30, 0xdc, 0xb7, 0x6c, 0x4a, 0xb5, 0x3f, 0x97, 0xd4, 0x62, 0x2d, 0
x06, 0xa4, 0xa5, 0x83, 0x5f,
43   0x2a, 0xda, 0xc9, 0x00, 0x7e, 0xa2, 0x55, 0xbf, 0x11, 0xd5, 0x9c, 0
xcf, 0x0e, 0xa0, 0x3d, 0x51,
44   0x7d, 0x93, 0x1b, 0xfe, 0xc4, 0x47, 0x09, 0x86, 0x0b, 0x8f, 0x9d, 0
x6a, 0x07, 0xb9, 0xb0, 0x98,
45   0x18, 0x32, 0x71, 0x4b, 0xef, 0x3b, 0x70, 0xa0, 0xe4, 0x40, 0xff, 0
xc3, 0xa9, 0xe6, 0x78, 0xf9,
46   0x8b, 0x46, 0x80, 0x1e, 0x38, 0xe1, 0xb8, 0xa8, 0xe0, 0x0c, 0x23, 0
x76, 0x1d, 0x25, 0x24, 0x05,
47   0xf1, 0x6e, 0x94, 0x28, 0x9a, 0x84, 0xe8, 0xa3, 0x4f, 0x77, 0xd3, 0
x85, 0xe2, 0x52, 0xf2, 0x82,
48   0x50, 0x7a, 0x2f, 0x74, 0x53, 0xb3, 0x61, 0xaf, 0x39, 0x35, 0xde, 0
xcd, 0x1f, 0x99, 0xac, 0xad,
49   0x72, 0x2c, 0xdd, 0xd0, 0x87, 0xbe, 0x5e, 0xa6, 0xec, 0x04, 0xc6, 0
x03, 0x34, 0xfb, 0xdb, 0x59,
50   0xb6, 0xc2, 0x01, 0xf0, 0x5a, 0xed, 0xa7, 0x66, 0x21, 0x7f, 0x8a, 0
x27, 0xc7, 0xc0, 0x29, 0xd7
51 },
52 {
53   0x14, 0x9d, 0xb9, 0xe7, 0x67, 0x4c, 0x50, 0x82, 0xca, 0xe5, 0x1d, 0
x31, 0xa0, 0xc6, 0xb2, 0x51,
54   0xa2, 0xd8, 0x54, 0x90, 0xd0, 0xce, 0x2d, 0x7d, 0xc7, 0x7e, 0xd7, 0
x94, 0xdf, 0x83, 0x8e, 0x6c,
55   0x66, 0xd2, 0x6f, 0x16, 0x1e, 0x76, 0xfe, 0xcc, 0xaa, 0x5a, 0x8f, 0
x17, 0xbd, 0x2c, 0xac, 0xea,
56   0x7b, 0x65, 0xa9, 0x10, 0xc0, 0x92, 0xee, 0xbe, 0x6a, 0x6e, 0x48, 0
x96, 0x95, 0xe9, 0x32, 0xbc,
57   0xa1, 0x42, 0xd5, 0xa7, 0x81, 0xb4, 0x5f, 0xe6, 0xc2, 0x5d, 0xad, 0
x3a, 0xb7, 0x0c, 0x8d, 0x01,
58   0x98, 0xfd, 0x12, 0x02, 0x75, 0x13, 0x0f, 0x6b, 0x22, 0xe2, 0xab, 0
xf7, 0x7f, 0xba, 0x97, 0xd1,
59   0x64, 0xd9, 0xc4, 0x59, 0xaf, 0x23, 0x33, 0x37, 0xde, 0xae, 0x60, 0
x05, 0x63, 0xa8, 0x52, 0xa5,
60   0x4e, 0xe0, 0xdd, 0x71, 0xf2, 0x24, 0x34, 0x57, 0x47, 0xa4, 0xb3, 0

```

```

x9e, 0x2f, 0xc1, 0xb8, 0xcb,
61   0x2b, 0xd4, 0x0d, 0x36, 0x91, 0x8b, 0x9c, 0x26, 0x25, 0x61, 0xa3, 0
xd6, 0xeb, 0x35, 0x53, 0xf4,
62   0x2e, 0x88, 0x80, 0xe4, 0x30, 0xdb, 0xfc, 0x0e, 0x77, 0x8c, 0x93, 0
xa6, 0x78, 0x06, 0xe1, 0xec,
63   0xf9, 0x03, 0xa0, 0x27, 0xda, 0xef, 0x5c, 0x00, 0x7a, 0x45, 0xe8, 0
x40, 0x1a, 0x4b, 0x5e, 0x73,
64   0xc3, 0xff, 0xf5, 0xf3, 0xb0, 0xc5, 0x49, 0x21, 0xfa, 0x11, 0x39, 0
x84, 0x43, 0x38, 0x85, 0x07,
65   0xf0, 0x79, 0x46, 0xf8, 0xe3, 0x1f, 0x09, 0xb6, 0xcd, 0x55, 0x1c, 0
x1b, 0xfb, 0x7c, 0xed, 0x6d,
66   0x15, 0x56, 0x86, 0x20, 0x68, 0x4a, 0x41, 0x4f, 0xd3, 0x99, 0x08, 0
xf6, 0x3f, 0x89, 0x62, 0x04,
67   0xcf, 0xc8, 0x69, 0x9f, 0x19, 0x5b, 0x44, 0x9b, 0x87, 0xb1, 0x3d, 0
xbb, 0xdc, 0x2a, 0xbf, 0x58,
68   0x3c, 0x8a, 0x18, 0x3e, 0x72, 0x0b, 0x28, 0x4d, 0xb5, 0x9a, 0xc9, 0
x74, 0x29, 0xf1, 0x3b, 0x70
69
70 },
71 {
72   0x68, 0x8d, 0xca, 0x4d, 0x73, 0x4b, 0x4e, 0x2a, 0xd4, 0x52, 0x26, 0
xb3, 0x54, 0x1e, 0x19, 0x1f,
73   0x22, 0x03, 0x46, 0x3d, 0x2d, 0x4a, 0x53, 0x83, 0x13, 0x8a, 0xb7, 0
xd5, 0x25, 0x79, 0xf5, 0xbd,
74   0x58, 0x2f, 0x0d, 0x02, 0xed, 0x51, 0x9e, 0x11, 0xf2, 0x3e, 0x55, 0
x5e, 0xd1, 0x16, 0x3c, 0x66,
75   0x70, 0x5d, 0xf3, 0x45, 0x40, 0xcc, 0xe8, 0x94, 0x56, 0x08, 0xce, 0
x1a, 0x3a, 0xd2, 0xe1, 0xdf,
76   0xb5, 0x38, 0x6e, 0x0e, 0xe5, 0xf4, 0xf9, 0x86, 0xe9, 0x4f, 0xd6, 0
x85, 0x23, 0xcf, 0x32, 0x99,
77   0x31, 0x14, 0xae, 0xee, 0xc8, 0x48, 0xd3, 0x30, 0xa1, 0x92, 0x41, 0
xb1, 0x18, 0xc4, 0x2c, 0x71,
78   0x72, 0x44, 0x15, 0xfd, 0x37, 0xbe, 0x5f, 0xaa, 0x9b, 0x88, 0xd8, 0
xab, 0x89, 0x9c, 0xfa, 0x60,
79   0xea, 0xbc, 0x62, 0x0c, 0x24, 0xa6, 0xa8, 0xec, 0x67, 0x20, 0xdb, 0
x7c, 0x28, 0xdd, 0xac, 0x5b,
80   0x34, 0x7e, 0x10, 0xf1, 0x7b, 0x8f, 0x63, 0xa0, 0x05, 0x9a, 0x43, 0
x77, 0x21, 0xbf, 0x27, 0x09,
81   0xc3, 0x9f, 0xb6, 0xd7, 0x29, 0xc2, 0xeb, 0xc0, 0xa4, 0x8b, 0x8c, 0
x1d, 0xfb, 0xff, 0xc1, 0xb2,
82   0x97, 0x2e, 0xf8, 0x65, 0xf6, 0x75, 0x07, 0x04, 0x49, 0x33, 0xe4, 0
xd9, 0xb9, 0xd0, 0x42, 0xc7,
83   0x6c, 0x90, 0x00, 0x8e, 0x6f, 0x50, 0x01, 0xc5, 0xda, 0x47, 0x3f, 0
xcd, 0x69, 0xa2, 0xe2, 0x7a,
84   0xa7, 0xc6, 0x93, 0x0f, 0xa0, 0x06, 0xe6, 0x2b, 0x96, 0xa3, 0x1c, 0
xaf, 0x6a, 0x12, 0x84, 0x39,
85   0xe7, 0xb0, 0x82, 0xf7, 0xfe, 0x9d, 0x87, 0x5c, 0x81, 0x35, 0xde, 0
xb4, 0xa5, 0xfc, 0x80, 0xef,
```

```

86     0xcb, 0xbb, 0x6b, 0x76, 0xba, 0x5a, 0x7d, 0x78, 0x0b, 0x95, 0xe3, 0
87     xad, 0x74, 0x98, 0x3b, 0x36,
88 }, {
89 {
90     0xa8, 0x43, 0x5f, 0x06, 0x6b, 0x75, 0x6c, 0x59, 0x71, 0xdf, 0x87, 0
91     x95, 0x17, 0xf0, 0xd8, 0x09,
92     0x6d, 0xf3, 0x1d, 0xcb, 0xc9, 0x4d, 0x2c, 0xaf, 0x79, 0xe0, 0x97, 0
93     xfd, 0x6f, 0x4b, 0x45, 0x39,
94     0x3e, 0xdd, 0xa3, 0x4f, 0xb4, 0xb6, 0x9a, 0x0e, 0x1f, 0xbf, 0x15, 0
95     xe1, 0x49, 0xd2, 0x93, 0xc6,
96     0x92, 0x72, 0x9e, 0x61, 0xd1, 0x63, 0xfa, 0xee, 0xf4, 0x19, 0xd5, 0
97     xad, 0x58, 0xa4, 0xbb, 0xa1,
98     0xdc, 0xf2, 0x83, 0x37, 0x42, 0xe4, 0x7a, 0x32, 0x9c, 0xcc, 0xab, 0
99     x4a, 0x8f, 0x6e, 0x04, 0x27,
100    0x2e, 0xe7, 0xe2, 0x5a, 0x96, 0x16, 0x23, 0x2b, 0xc2, 0x65, 0x66, 0
101    x0f, 0xbc, 0xa9, 0x47, 0x41,
102    0x34, 0x48, 0xfc, 0xb7, 0x6a, 0x88, 0xa5, 0x53, 0x86, 0xf9, 0x5b, 0
103    xdb, 0x38, 0x7b, 0xc3, 0x1e,
104    0x22, 0x33, 0x24, 0x28, 0x36, 0xc7, 0xb2, 0x3b, 0x8e, 0x77, 0xba, 0
105    xf5, 0x14, 0x9f, 0x08, 0x55,
106    0x9b, 0x4c, 0xfe, 0x60, 0x5c, 0xda, 0x18, 0x46, 0xcd, 0x7d, 0x21, 0
107    xb0, 0x3f, 0x1b, 0x89, 0xff,
108    0xeb, 0x84, 0x69, 0x3a, 0x9d, 0xd7, 0xd3, 0x70, 0x67, 0x40, 0xb5, 0
109    xde, 0x5d, 0x30, 0x91, 0xb1,
110    0x78, 0x11, 0x01, 0xe5, 0x00, 0x68, 0x98, 0xa0, 0xc5, 0x02, 0xa6, 0
111    x74, 0x2d, 0x0b, 0xa2, 0x76,
112    0xb3, 0xbe, 0xce, 0xbd, 0xae, 0xe9, 0x8a, 0x31, 0x1c, 0xec, 0xf1, 0
113    x99, 0x94, 0xaa, 0xf6, 0x26,
114    0x2f, 0xef, 0xe8, 0x8c, 0x35, 0x03, 0xd4, 0x7f, 0xfb, 0x05, 0xc1, 0
115    x5e, 0x90, 0x20, 0x3d, 0x82,
116    0xf7, 0xea, 0xa0, 0xd, 0x7e, 0xf8, 0x50, 0x1a, 0xc4, 0x07, 0x57, 0
117    xb8, 0x3c, 0x62, 0xe3, 0xc8,
118    0xac, 0x52, 0x64, 0x10, 0xd0, 0xd9, 0x13, 0x0c, 0x12, 0x29, 0x51, 0
119    xb9, 0xcf, 0xd6, 0x73, 0x8d,
120    0x81, 0x54, 0xc0, 0xed, 0x4e, 0x44, 0xa7, 0x2a, 0x85, 0x25, 0xe6, 0
121    xca, 0x7c, 0x8b, 0x56, 0x80
122 }
123 };
124
125 uint8_t mea_invSbox[4][256] = {
126 {
127     0x83, 0xf2, 0x2a, 0xeb, 0xe9, 0xbf, 0x7b, 0x9c, 0x34, 0x96, 0x8d, 0
128     x98, 0xb9, 0x69, 0x8c, 0x29,
129     0x3d, 0x88, 0x68, 0x06, 0x39, 0x11, 0x4c, 0x0e, 0xa0, 0x56, 0x40, 0
130     x92, 0x15, 0xbc, 0xb3, 0xdc,

```

```

114     0x6f, 0xf8, 0x26, 0xba, 0xbe, 0xbd, 0x31, 0xfb, 0xc3, 0xfe, 0x80, 0
115     x61, 0xe1, 0x7a, 0x32, 0xd2,
116     0x70, 0x20, 0xa1, 0x45, 0xec, 0xd9, 0x1a, 0x5d, 0xb4, 0xd8, 0x09, 0
117     xa5, 0x55, 0x8e, 0x37, 0x76,
118     0xa9, 0x67, 0x10, 0x17, 0x36, 0x65, 0xb1, 0x95, 0x62, 0x59, 0x74, 0
119     xa3, 0x50, 0x2f, 0x4b, 0xc8,
120     0xd0, 0x8f, 0xcd, 0xd4, 0x3c, 0x86, 0x12, 0x1d, 0x23, 0xef, 0xf4, 0
121     x53, 0x19, 0x35, 0xe6, 0x7f,
122     0x5e, 0xd6, 0x79, 0x51, 0x22, 0x14, 0xf7, 0x1e, 0x4a, 0x42, 0x9b, 0
123     x41, 0x73, 0x2d, 0xc1, 0x5c,
124     0xa6, 0xa2, 0xe0, 0x2e, 0xd3, 0x28, 0xbb, 0xc9, 0xae, 0x6a, 0xd1, 0
125     x5a, 0x30, 0x90, 0x84, 0xf9,
126     0xb2, 0x58, 0xcf, 0x7e, 0xc5, 0xcb, 0x97, 0xe4, 0x16, 0x6c, 0xfa, 0
127     xb0, 0x6d, 0x1f, 0x52, 0x99,
128     0xd, 0x4e, 0x03, 0x91, 0xc2, 0x4d, 0x64, 0x77, 0x9f, 0xdd, 0xc4, 0
129     x49, 0x8a, 0x9a, 0x24, 0x38,
130     0xa7, 0x57, 0x85, 0xc7, 0x7c, 0x7d, 0xe7, 0xf6, 0xb7, 0xac, 0x27, 0
131     x46, 0xde, 0xdf, 0x3b, 0xd7,
132     0x9e, 0x2b, 0x0b, 0xd5, 0x13, 0x75, 0xf0, 0x72, 0xb6, 0x9d, 0x1b, 0
133     x01, 0x3f, 0x44, 0xe5, 0x87,
134     0xfd, 0x07, 0xf1, 0xab, 0x94, 0x18, 0xea, 0xfc, 0x3a, 0x82, 0x5f, 0
135     x05, 0x54, 0xdb, 0x00, 0x8b,
136     0xe3, 0x48, 0x0c, 0xca, 0x78, 0x89, 0x0a, 0xff, 0x3e, 0x5b, 0x81, 0
137     xee, 0x71, 0xe2, 0xda, 0x2c,
138     0xb8, 0xb5, 0xcc, 0x6e, 0xa8, 0x6b, 0xad, 0x60, 0xc6, 0x08, 0x04, 0
139     x02, 0xe8, 0xf5, 0x4f, 0xa4,
140     0xf3, 0xc0, 0xce, 0x43, 0x25, 0x1c, 0x21, 0x33, 0x0f, 0xaf, 0x47, 0
141     xed, 0x66, 0x63, 0x93, 0xaa
142 },
143 {
144     0xa7, 0x4f, 0x53, 0xa1, 0xdf, 0x6b, 0x9d, 0xbf, 0xda, 0xc6, 0x0c, 0
145     xf5, 0x4d, 0x82, 0x97, 0x56,
146     0x33, 0xb9, 0x52, 0x55, 0x00, 0xd0, 0x23, 0x2b, 0xf2, 0xe4, 0xac, 0
147     xcb, 0xca, 0xa, 0x24, 0xc5,
148     0xd3, 0xb7, 0x58, 0x65, 0x75, 0x88, 0x87, 0xa3, 0xf6, 0xfc, 0xed, 0
149     x80, 0x2d, 0x16, 0x90, 0x7c,
150     0x94, 0x0b, 0x3e, 0x66, 0x76, 0x8d, 0x83, 0x67, 0xbd, 0xba, 0x4b, 0
151     xfe, 0xf0, 0xea, 0xf3, 0xdc,
152     0xab, 0xd6, 0x41, 0xbc, 0xe6, 0xa9, 0xc2, 0x78, 0x3a, 0xb6, 0xd5, 0
153     xad, 0x05, 0xf7, 0x70, 0xd7,
154     0x06, 0x0f, 0x6e, 0x8e, 0x12, 0xc9, 0xd1, 0x77, 0xef, 0x63, 0x29, 0
155     xe5, 0xa6, 0x49, 0xae, 0x46,
156     0x6a, 0x89, 0xde, 0x6c, 0x60, 0x31, 0x20, 0x04, 0xd4, 0xe2, 0x38, 0
157     x57, 0x1f, 0xcf, 0x39, 0x22,
158     0xff, 0x73, 0xf4, 0xaf, 0xfb, 0x54, 0x25, 0x98, 0x9c, 0xc1, 0xa8, 0
159     x30, 0xcd, 0x17, 0x19, 0x5c,
160     0x92, 0x44, 0x07, 0x1d, 0xbb, 0xbe, 0xd2, 0xe8, 0x91, 0xdd, 0xf1, 0
161     x85, 0x99, 0x4e, 0x1e, 0x2a,

```

```

139     0x13, 0x84, 0x35, 0x9a, 0x1b, 0x3c, 0x3b, 0x5e, 0x50, 0xd9, 0xf9, 0
140     xe7, 0x86, 0x01, 0x7b, 0xe3,
141     0xa2, 0x40, 0x10, 0x8a, 0x79, 0x6f, 0x9b, 0x43, 0x6d, 0x32, 0x28, 0
142     x5a, 0x2e, 0x4a, 0x69, 0x64,
143     0xb4, 0xe9, 0x0e, 0x7a, 0x45, 0xf8, 0xc7, 0x4c, 0x7e, 0x02, 0x5d, 0
144     xeb, 0x3f, 0x2c, 0x37, 0xee,
145     0x34, 0x7d, 0x48, 0xb0, 0x62, 0xb5, 0x0d, 0x18, 0xe1, 0xfa, 0x08, 0
146     x7f, 0x27, 0xc8, 0x15, 0xe0,
147     0x14, 0x5f, 0x21, 0xd8, 0x81, 0x42, 0x8b, 0x1a, 0x11, 0x61, 0xa4, 0
148     x95, 0xec, 0x72, 0x68, 0x1c,
149     0x71, 0x9e, 0x59, 0xc4, 0x93, 0x09, 0x47, 0x03, 0xaa, 0x3d, 0x2f, 0
150     x8c, 0x9f, 0xce, 0x36, 0xa5,
151     0xc0, 0xfd, 0x74, 0xb3, 0x8f, 0xb2, 0xdb, 0x5b, 0xc3, 0xa0, 0xb8, 0
152     xcc, 0x96, 0x51, 0x26, 0xb1
153 },
154 {
155     0xb2, 0xb6, 0x23, 0x11, 0xa7, 0x88, 0xc5, 0xa6, 0x39, 0x8f, 0xc4, 0
156     xe8, 0x73, 0x22, 0x43, 0xc3,
157     0x82, 0x27, 0xcd, 0x18, 0x51, 0x62, 0x2d, 0xf7, 0x5c, 0x0e, 0x3b, 0
158     xfd, 0xca, 0x9b, 0x0d, 0x0f,
159     0x79, 0x8c, 0x10, 0x4c, 0x74, 0x1c, 0x0a, 0x8e, 0x7c, 0x94, 0x07, 0
160     xc7, 0x5e, 0x14, 0xa1, 0x21,
161     0x57, 0x50, 0x4e, 0xa9, 0x80, 0xd9, 0xef, 0x64, 0x41, 0xcf, 0x3c, 0
162     xee, 0x2e, 0x13, 0x29, 0xba,
163     0x34, 0x5a, 0xae, 0x8a, 0x61, 0x33, 0x12, 0xb9, 0x55, 0xa8, 0x15, 0
164     x05, 0xf6, 0x03, 0x06, 0x49,
165     0xb5, 0x25, 0x09, 0x16, 0x0c, 0x2a, 0x38, 0xfc, 0x20, 0xf4, 0xe5, 0
166     x7f, 0xd7, 0x31, 0x2b, 0x66,
167     0x6f, 0xff, 0x72, 0x86, 0xf0, 0xa3, 0x2f, 0x78, 0x00, 0xbc, 0xcc, 0
168     xe2, 0xb0, 0xf1, 0x42, 0xb4,
169     0x30, 0x5f, 0x60, 0x04, 0xec, 0xa5, 0xe3, 0x8b, 0xe7, 0x1d, 0xbf, 0
170     x84, 0x7b, 0xe6, 0x81, 0xf8,
171     0xde, 0xd8, 0xd2, 0x17, 0xce, 0x4b, 0x47, 0xd6, 0x69, 0x6c, 0x19, 0
172     x99, 0x9a, 0x01, 0xb3, 0x85,
173     0xb1, 0xf9, 0x59, 0xc2, 0x37, 0xe9, 0xc8, 0xa0, 0xed, 0x4f, 0x89, 0
174     x68, 0x6d, 0xd5, 0x26, 0x91,
175     0x87, 0x58, 0xbd, 0xc9, 0x98, 0xdc, 0x75, 0xc0, 0x76, 0xf5, 0x67, 0
176     x6b, 0x7e, 0xeb, 0x52, 0xcb,
177     0xd1, 0x5b, 0x9f, 0x0b, 0xdb, 0x40, 0x92, 0x1a, 0xfa, 0xac, 0xe4, 0
178     xe1, 0x71, 0x1f, 0x65, 0x8d,
179     0x97, 0x9e, 0x95, 0x90, 0x5d, 0xb7, 0xc1, 0xaf, 0x54, 0xfb, 0x02, 0
180     xe0, 0x35, 0xbb, 0x3a, 0x4d,
181     0xad, 0x2c, 0x3d, 0x56, 0x08, 0x1b, 0x4a, 0x93, 0x6a, 0xab, 0xb8, 0
182     x7a, 0xf2, 0x7d, 0xda, 0x3f,
183     0xfe, 0x3e, 0xbe, 0xea, 0xaa, 0x44, 0xc6, 0xd0, 0x36, 0x48, 0x70, 0
184     x96, 0x77, 0x24, 0x53, 0xdf,
185     0xf3, 0x83, 0x28, 0x32, 0x45, 0x1e, 0xa4, 0xd3, 0xa2, 0x46, 0x6e, 0
186     x9c, 0xdd, 0x63, 0xd4, 0x9d

```

```

164 },
165 {
166     0xa4, 0xa2, 0xa9, 0xc5, 0x4e, 0xc9, 0x03, 0xd9, 0x7e, 0x0f, 0xd2, 0
167     xad, 0xe7, 0xd3, 0x27, 0x5b,
168     0xe3, 0xa1, 0xe8, 0xe6, 0x7c, 0x2a, 0x55, 0x0c, 0x86, 0x39, 0xd7, 0
169     x8d, 0xb8, 0x12, 0x6f, 0x28,
170     0xcd, 0x8a, 0x70, 0x56, 0x72, 0xf9, 0xbf, 0x4f, 0x73, 0xe9, 0xf7, 0
171     x57, 0x16, 0xac, 0x50, 0xc0,
172     0x9d, 0xb7, 0x47, 0x71, 0x60, 0xc4, 0x74, 0x43, 0x6c, 0x1f, 0x93, 0
173     x77, 0xdc, 0xce, 0x20, 0x8c,
174     0x99, 0x5f, 0x44, 0x01, 0xf5, 0x1e, 0x87, 0x5e, 0x61, 0x2c, 0x4b, 0
175     x1d, 0x81, 0x15, 0xf4, 0x23,
176     0xd6, 0xea, 0xe1, 0x67, 0xf1, 0x7f, 0xfe, 0xda, 0x3c, 0x07, 0x53, 0
177     x6a, 0x84, 0x9c, 0xcb, 0x02,
178     0x83, 0x33, 0xdd, 0x35, 0xe2, 0x59, 0x5a, 0x98, 0xa5, 0x92, 0x64, 0
179     x04, 0x06, 0x10, 0x4d, 0x1c,
180     0x97, 0x08, 0x31, 0xee, 0xab, 0x05, 0xaf, 0x79, 0xa0, 0x18, 0x46, 0
181     x6d, 0xfc, 0x89, 0xd4, 0xc7,
182     0xff, 0xf0, 0xcf, 0x42, 0x91, 0xf8, 0x68, 0xa, 0x65, 0x8e, 0xb6, 0
183     xfd, 0xc3, 0xef, 0x78, 0x4c,
184     0xcc, 0x9e, 0x30, 0x2e, 0xbc, 0x0b, 0x54, 0x1a, 0xa6, 0xbb, 0x26, 0
185     x80, 0x48, 0x94, 0x32, 0x7d,
186     0xa7, 0x3f, 0xae, 0x22, 0x3d, 0x66, 0xaa, 0xf6, 0x00, 0x5d, 0xbd, 0
187     x4a, 0xe0, 0x3b, 0xb4, 0x17,
188     0x8b, 0x9f, 0x76, 0xb0, 0x24, 0x9a, 0x25, 0x63, 0xdb, 0xeb, 0x7a, 0
189     x3e, 0x5c, 0xb3, 0xb1, 0x29,
190     0xf2, 0xca, 0x58, 0x6e, 0xd8, 0xa8, 0x2f, 0x75, 0xdf, 0x14, 0xfb, 0
191     x13, 0x49, 0x88, 0xb2, 0xec,
192     0xe4, 0x34, 0x2d, 0x96, 0xc6, 0x3a, 0xed, 0x95, 0x0e, 0xe5, 0x85, 0
193     x6b, 0x40, 0x21, 0x9b, 0x09,
194     0x19, 0x2b, 0x52, 0xde, 0x45, 0xa3, 0xfa, 0x51, 0xc2, 0xb5, 0xd1, 0
195     x90, 0xb9, 0xf3, 0x37, 0xc1,
196     0xd, 0xba, 0x41, 0x11, 0x38, 0x7b, 0xbe, 0xd0, 0xd5, 0x69, 0x36, 0
197     xc8, 0x62, 0x1b, 0x82, 0x8f
198 }
199 ;

```

### 8.1.4 mea.c

```

1 /*
2 * Projekt : MEA
3 * Autor : Michael Engel
4 * Datei : mea.c
5 */
6
7 #include <stdint.h>
8 #include <stdio.h>
9 #include <stdlib.h>

```

```

10 #include <string.h>
11
12 #include "mea.h"
13 #include "tables.h"
14
15 int mea_xorRoundKey(mea_t *mea_ctx, int round);
16
17 uint64_t __reverseWord(uint64_t in) { return __builtin_bswap64(in); }
18
19 uint8_t *__wordsToBytes(uint64_t *in) { return (uint8_t *)in; }
20
21 uint64_t *__bytesToWords(uint8_t *in) { return (uint64_t *)in; }
22
23 uint8_t __multiplyGF(uint8_t a, uint8_t b) {
24     uint8_t res = 0, hbs = 0;
25
26     for (int i = 0; i < 0x08; i++) {
27         if ((b & 0x01) == 1) {
28             res ^= a;
29         }
30
31         hbs = (a & 0x80);
32         a <<= 1;
33
34         if (hbs == 0x80) {
35             a ^= 0x11d; // m(x) = x8 + x4 + x3 + x2 +1
36         }
37         b >>= 1;
38     }
39
40     return res;
41 }
42
43 int __matrixMultiplywState(mea_t *mea_ctx, uint8_t in_matrix[8][8]) {
44     int n_col, n_row, b;
45     uint8_t pr;
46     uint64_t res;
47     uint8_t *pmstate = __wordsToBytes(mea_ctx->m_state);
48
49     for (n_col = 0; n_col < MEA_NW_STATE; n_col++) {
50         res = 0;
51         for (n_row = sizeof(uint64_t) - 1; n_row >= 0; n_row--) {
52             pr = 0;
53             for (b = sizeof(uint64_t) - 1; b >= 0; b--) {
54                 pr ^= __multiplyGF(BYTE_TO_M_STATE(pmstate, b, n_col),
55                                     in_matrix[n_row][b]);
56             }
57             res |= (uint64_t)pr << (n_row * sizeof(uint64_t));

```

```

58     }
59     mea_ctx->m_state[n_col] = res;
60 }
61
62 return 0;
63 }
64
65 int __returnFncRnd(meas_t *mea_ctx, uint8_t *in, int i, int rKP, int dRP
66 ) {
67 if (in[i] == MEA_FNC_HRSR)
68     mea_horShiftRows(mea_ctx);
69 else if (in[i] == MEA_FNC_SBB)
70     mea_subBytes(mea_ctx);
71 else if (in[i] == MEA_FNC_VRSC)
72     mea_verShiftColumns(mea_ctx);
73 else if (in[i] == MEA_FNC_MXCL)
74     mea_mixColumns(mea_ctx);
75 else if (in[i] == MEA_FNC_DRT)
76     mea_dimRotate(mea_ctx, dRP);
77 else if (in[i] == MEA_FNC_XRK)
78     mea_xorRoundKey(mea_ctx, rKP);
79 else
80     return -1;
81
82 return 1;
83 }
84
85
86
87
88 }
89
90 int __returnInvFncRnd(meas_t *mea_ctx, uint8_t *in, int i, int rKP, int
91 dRP) {
92 if (in[i] == MEA_FNC_HRSR)
93     mea_invHorShiftRows(mea_ctx);
94 else if (in[i] == MEA_FNC_SBB)
95     mea_invSubBytes(mea_ctx);
96 else if (in[i] == MEA_FNC_VRSC)
97     mea_invVerShiftColumns(mea_ctx);
98 else if (in[i] == MEA_FNC_MXCL)
99     mea_invMixColumns(mea_ctx);
100 else if (in[i] == MEA_FNC_DRT)
101
102
103

```

```

104     mea_invDimRotate(mea_ctx, dRP);
105
106     else if (in[i] == MEA_FNC_XRK)
107         mea_xorRoundKey(mea_ctx, rKP);
108
109     else
110         return -1;
111
112     return 1;
113 }
114
115 uint8_t __returnnVInt(uint8_t in) {
116     if (in <= 0x2A) // 42
117         return 0x00;
118
119     else if (in <= 0x54) // 84
120         return 0x01;
121
122     else if (in <= 0x7E) // 126
123         return 0x02;
124
125     else if (in <= 0xA8) // 168
126         return 0x03;
127
128     else if (in <= 0xD2) // 210
129         return 0x04;
130
131     else if (in <= 0xFC) // 252
132         return 0x05;
133     else
134         return 0x05;
135 }
136
137 mea_t *mea_init() {
138     mea_t *mea_ctx = (mea_t *)malloc(sizeof(mea_t));
139
140     if (mea_ctx == NULL)
141         return NULL;
142
143     mea_ctx->m_state = (uint64_t *)calloc(MEA_NW_STATE, sizeof(uint64_t))
144         ;
145     if (mea_ctx->m_state == NULL)
146         return NULL;
147
148     mea_ctx->r_seq = calloc(MEA_SUB_ROUNDS * MEA_M_ROUNDS / 2, sizeof(
149         uint8_t *));
149     if (mea_ctx->r_seq == NULL)
149         return NULL;

```

```

150
151     for (int i = 0; i < MEA_SUB_ROUNDS * MEA_M_ROUNDS / 2; i++) {
152         mea_ctx->r_seq[i] = (uint8_t *)calloc(MEA_M_ROUNDS, sizeof(uint8_t))
153     );
154
155     if (mea_ctx->r_seq[i] == NULL)
156         return NULL;
157 }
158
159     mea_ctx->r_keys =
160         (uint64_t **)calloc(MEA_SUB_ROUNDS * MEA_M_ROUNDS, sizeof(
161             uint64_t **));
162
163     if (mea_ctx->r_keys == NULL)
164         return NULL;
165
166     for (int i = 0; i < MEA_SUB_ROUNDS * MEA_M_ROUNDS; i++) {
167         mea_ctx->r_keys[i] = (uint64_t *)calloc(MEA_NW_KEY, sizeof(uint64_t
168             ));
169
170     if (mea_ctx->r_keys[i] == NULL)
171         return NULL;
172 }
173
174     return mea_ctx;
175 }
176
177 int mea_del(mea_t *mea_ctx) {
178     free(mea_ctx->m_state);
179
180     for (int i = 0; i < MEA_SUB_ROUNDS * MEA_M_ROUNDS; i++) {
181         free(mea_ctx->r_keys[i]);
182     }
183
184     free(mea_ctx->r_keys);
185     free(mea_ctx->r_seq);
186     free(mea_ctx);
187
188     mea_ctx = NULL;
189     return 0;
190 }
191
192 int mea_generateRKeys(mea_t *mea_ctx, uint64_t *mkey) {
193     uint64_t *ntmp;
194

```

```

195  for (int r = 0; r < MEA_SUB_ROUNDS * MEA_M_ROUNDS; r++) {
196      int tmp, tmp2;
197      uint64_t *inpoi;
198
199      ntmp = mea_ctx->r_keys[r];
200      if (r == 0)
201          inpoi = mkey;
202
203      else
204          inpoi = mea_ctx->r_keys[r - 1];
205
206      for (int l = 0; l < MEA_NW_KEY; l++) {
207          ntmp[l] = inpoi[l] ^ RKCON;
208      }
209
210      for (int i = 0; i < MEA_NW_KEY; i++) {
211          ntmp[i] =
212              mea_sbox[0x01][(ntmp[i] & 0x00000000000000FF) | 0x08]
213              ((uint64_t)mea_sbox[0x00][(ntmp[i] & 0x000000000000FF00) >> 0
214                  << 0x08) |
215              ((uint64_t)mea_sbox[0x03][(ntmp[i] & 0x0000000000FF0000) >> 0
216                  << 0x10) |
217              ((uint64_t)mea_sbox[0x02][(ntmp[i] & 0x00000000FF000000) >> 0
218                  << 0x18) |
219              ((uint64_t)mea_sbox[0x03][(ntmp[i] & 0x000000FF00000000) >> 0
220                  << 0x20) |
221              ((uint64_t)mea_sbox[0x00][(ntmp[i] & 0x0000FF0000000000) >> 0
222                  << 0x28) |
223              ((uint64_t)mea_sbox[0x01][(ntmp[i] & 0x00FF000000000000) >> 0
224                  << 0x30) |
225              ((uint64_t)mea_sbox[0x02][(ntmp[i] & 0xFF00000000000000) >> 0
226                  << 0x38);
227      }
228
229      uint8_t *tmp_key = __wordsToBytes(ntmp);
230      for (int z = 0; z < MEA_MS_DIM + 1; z++) {
231          for (int i = 1; i < MEA_MS_ROW; i++) {
232              int s = 0;
233              while (s < i) {
234                  tmp = tmp_key[MEA_MS_IN_DIM * z + i];
235

```

```

236     for (int k = 1; k < MEA_MS_ROW; k++) {
237         tmp_key[MEA_MS_IN_DIM * z + (MEA_MS_ROW * (k - 1)) + i] =
238             tmp_key[MEA_MS_IN_DIM * z + MEA_MS_ROW * k + i];
239     }
240
241     tmp_key[MEA_MS_IN_DIM * z + (MEA_MS_ROW * (MEA_MS_ROW - 1)) +
242     i] =
242         tmp;
243     s++;
244 }
245 }
246 }
247
248 for (int i = 0; i < MEA_MS_ROW; i++) {
249     for (int k = 0; k < MEA_MS_ROW; k++) {
250         tmp2 = tmp_key[MEA_MS_IN_DIM * 0x02 + MEA_MS_ROW * i + k];
251         tmp_key[MEA_MS_IN_DIM * 0x02 + MEA_MS_ROW * i + k] =
252             tmp_key[MEA_MS_IN_DIM * 0x03 + MEA_MS_ROW * i + k];
253         tmp_key[MEA_MS_IN_DIM * 0x03 + MEA_MS_ROW * i + k] = tmp2;
254     }
255 }
256
257 ntmp = __bytesToWords(tmp_key);
258 for (int i = 0; i < MEA_NW_KEY; i++) {
259     ntmp[i] = ntmp[i] ^ inpoi[i];
260 }
261 }
262
263 for (int i = 0; i < MEA_M_ROUNDS * MEA_SUB_ROUNDS / 2; i++) {
264     uint8_t *tmp = mea_ctx->r_seq[i];
265
266     for (int j = 0; j < MEA_SUB_ROUNDS; j++) {
267         tmp[j] = j;
268     }
269 }
270
271 mea_rSeqGen(mea_ctx);
272 return 0;
273 }
274
275 int mea_rSeqGen(mea_t *mea_ctx) {
276     for (int w = 0; w < MEA_M_ROUNDS; w++) {
277         for (int i = 0; i < MEA_M_ROUNDS * MEA_SUB_ROUNDS / 2; i++) {
278             uint8_t *tmp_key = (uint8_t *)mea_ctx->r_keys[i + w];
279             uint8_t *tmp_nseq = mea_ctx->r_seq[i];
280
281             for (int j = 0; j < MEA_SUB_ROUNDS - 1; j++) {
282                 uint8_t tmp_p = __returnVInt(tmp_key[j]);

```

```

283     uint8_t tmp_op;
284
285     tmp_op = tmp_nseq[tmp_p];
286     tmp_nseq[tmp_p] = tmp_nseq[_returnVInt(tmp_key[j + 1])];
287     tmp_nseq[_returnVInt(tmp_key[j + 1])] = tmp_op;
288 }
289 }
290 }
291 return 0;
292 }
293
294 int mea_blockEncipher(mea_t *mea_ctx, uint64_t *plain, uint64_t *cipher
295 ) {
296     memcpy(mea_ctx->m_state, plain, MEA_NW_STATE * sizeof(uint64_t));
297
298     for (int i = 0; i < MEA_M_ROUNDS; i++) {
299         for (int j = 0; j < MEA_SUB_ROUNDS; j++) {
300             if (j % 2 == 0) {
301                 mea_horShiftRows(mea_ctx);
302                 mea_subBytes(mea_ctx);
303
304                 mea_verShiftColumns(mea_ctx);
305                 mea_mixColumns(mea_ctx);
306
307                 mea_dimRotate(mea_ctx, j / 2);
308                 mea_xorRoundKey(mea_ctx, i * MEA_M_ROUNDS + j);
309             } else {
310                 uint8_t *tmp_seq = mea_ctx->r_seq[((i * MEA_M_ROUNDS + j + 1) /
311                     2) - 1];
312
313                 for (int rndR = 0; rndR < MEA_SUB_ROUNDS; rndR++) {
314
315                     __returnFncRnd(mea_ctx, tmp_seq, rndR, i * MEA_M_ROUNDS + j,
316                                     (j + 1) / 2);
317
318                 }
319             }
320         }
321
322         memcpy(cipher, mea_ctx->m_state, MEA_NW_STATE * sizeof(uint64_t));
323     return 0;
324 }
325
326 int mea_blockDecipher(mea_t *mea_ctx, uint64_t *cipher, uint64_t *plain
327 ) {
328     memcpy(mea_ctx->m_state, cipher, MEA_NW_STATE * sizeof(uint64_t));

```

```

328
329     for (int i = MEA_M_ROUNDS; i > 0; i--) {
330         for (int j = MEA_SUB_ROUNDS; j > 0; j--) {
331
332             if ((j - 1) % 2 == 0) {
333                 mea_xorRoundKey(mea_ctx, (i - 1) * MEA_M_ROUNDS + (j - 1));
334                 mea_invDimRotate(mea_ctx, ((j - 1) / 2));
335
336                 mea_invMixColumns(mea_ctx);
337                 mea_invVerShiftColumns(mea_ctx);
338
339                 mea_invSubBytes(mea_ctx);
340                 mea_invHorShiftRows(mea_ctx);
341             } else {
342                 uint8_t *tmp_seq = mea_ctx->r_seq[((i - 1) * MEA_M_ROUNDS + j - 2) / 2];
343
344                 mea_xorRoundKey(mea_ctx, (i - 1) * MEA_M_ROUNDS + (j - 1));
345                 for (int rndR = MEA_SUB_ROUNDS; rndR > 0; rndR--) {
346
347                     __returnInvFncRnd(mea_ctx, tmp_seq, rndR - 1,
348                                         (i - 1) * MEA_M_ROUNDS + (j - 1), j / 2);
349
350                 }
351             }
352         }
353     }
354     memcpy(plain, mea_ctx->m_state, MEA_NW_STATE * sizeof(uint64_t));
355     return 0;
356 }
357
358 int mea_verShiftColumns(mea_t *mea_ctx) {
359     uint8_t z, i, k, s, tmp;
360     uint8_t *pmstate = __wordsToBytes(mea_ctx->m_state);
361
362     for (z = 0; z < MEA_MS_DIM + 1; z++) {
363         for (i = 1; i < MEA_MS_ROW; i++) {
364             s = 0;
365             while (s < i) {
366                 tmp = pmstate[MEA_MS_IN_DIM * z + (MEA_MS_ROW * (MEA_MS_ROW - 1)) + i];
367
368                 for (k = MEA_MS_ROW - 1; k > 0; k--) {
369                     pmstate[MEA_MS_IN_DIM * z + MEA_MS_ROW * k + i] =
370                         pmstate[MEA_MS_IN_DIM * z + (MEA_MS_ROW * (k - 1)) + i];
371                 }
372
373                 pmstate[MEA_MS_IN_DIM * z + i] = tmp;

```

```

374         s++;
375     }
376   }
377 }
378
379 mea_ctx->m_state = __bytesToWords(pmstate);
380 return 0;
381 }
382
383 int mea_invVerShiftColumns(mea_t *mea_ctx) {
384   uint8_t z, i, k, s, tmp;
385   uint8_t *pmstate = __wordsToBytes(mea_ctx->m_state);
386
387   for (z = 0; z < MEA_MS_DIM + 1; z++) {
388     for (i = 1; i < MEA_MS_ROW; i++) {
389       s = 0;
390       while (s < i) {
391         tmp = pmstate[MEA_MS_IN_DIM * z + i];
392
393         for (k = 1; k < MEA_MS_ROW; k++) {
394           pmstate[MEA_MS_IN_DIM * z + (MEA_MS_ROW * (k - 1)) + i] =
395             pmstate[MEA_MS_IN_DIM * z + MEA_MS_ROW * k + i];
396         }
397
398         pmstate[MEA_MS_IN_DIM * z + (MEA_MS_ROW * (MEA_MS_ROW - 1)) + i]
399       ] = tmp;
400       s++;
401     }
402   }
403
404   mea_ctx->m_state = __bytesToWords(pmstate);
405   return 0;
406 }
407
408 int mea_horShiftRows(mea_t *mea_ctx) {
409   uint8_t z, i, k, s, tmp;
410   uint8_t *pmstate = __wordsToBytes(mea_ctx->m_state);
411
412   for (z = 0; z < MEA_MS_DIM + 1; z++) {
413     for (i = 1; i < MEA_MS_ROW; i++) {
414       s = 0;
415       while (s < i) {
416         tmp = pmstate[MEA_MS_IN_DIM * z + MEA_MS_ROW * i + MEA_MS_ROW -
417           1];
418
419         for (k = MEA_MS_ROW - 1; k > 0; k--) {
420           pmstate[MEA_MS_IN_DIM * z + MEA_MS_ROW * i + k] =

```

```

420             pmstate[MEA_MS_IN_DIM * z + MEA_MS_ROW * i + k - 1];
421         }
422
423         pmstate[MEA_MS_IN_DIM * z + MEA_MS_ROW * i] = tmp;
424         s++;
425     }
426 }
427 }
428
429 mea_ctx->m_state = __bytesToWords(pmstate);
430 return 0;
431 }
432
433 int mea_invHorShiftRows(mea_t *mea_ctx) {
434     uint8_t z, i, k, s, tmp;
435     uint8_t *pmstate = __wordsToBytes(mea_ctx->m_state);
436
437     for (z = 0; z < MEA_MS_DIM + 1; z++) {
438         for (i = 1; i < MEA_MS_ROW; i++) {
439             s = 0;
440             while (s < i) {
441                 tmp = pmstate[MEA_MS_IN_DIM * z + MEA_MS_ROW * i + 0];
442
443                 for (k = 1; k < MEA_MS_ROW; k++) {
444                     pmstate[MEA_MS_IN_DIM * z + MEA_MS_ROW * i + k - 1] =
445                         pmstate[MEA_MS_IN_DIM * z + MEA_MS_ROW * i + k];
446                 }
447
448                 pmstate[MEA_MS_IN_DIM * z + MEA_MS_ROW * i + MEA_MS_ROW - 1] =
449                     tmp;
450                     s++;
451             }
452         }
453
454     mea_ctx->m_state = __bytesToWords(pmstate);
455     return 0;
456 }
457
458 int mea_subBytes(mea_t *mea_ctx) {
459     for (int i = 0; i < MEA_NW_STATE; i++) {
460         mea_ctx->m_state[i] =
461             mea_sbox[0x00][(mea_ctx->m_state[i] & 0xffffffff000000FF) |
462             ((uint64_t)
463                 mea_sbox[0x01][(mea_ctx->m_state[i] & 0x000000000000FF00)
464             >> 0x08]
465                 << 0x08) |
466             ((uint64_t)

```

```

466             mea_sbox[0x02][(mea_ctx->m_state[i] & 0x0000000000FF0000)
467             >> 0x10]
468             << 0x10) |
469             ((uint64_t)
470             mea_sbox[0x03][(mea_ctx->m_state[i] & 0x00000000FF000000)
471             >> 0x18]
472             << 0x18) |
473             ((uint64_t)
474             mea_sbox[0x00][(mea_ctx->m_state[i] & 0x000000FF00000000)
475             >> 0x20]
476             << 0x20) |
477             ((uint64_t)
478             mea_sbox[0x01][(mea_ctx->m_state[i] & 0x0000FF0000000000)
479             >> 0x28]
480             << 0x28) |
481             ((uint64_t)
482             mea_sbox[0x02][(mea_ctx->m_state[i] & 0x00FF000000000000)
483             >> 0x30]
484             << 0x30) |
485             ((uint64_t)
486             mea_sbox[0x03][(mea_ctx->m_state[i] & 0xFF00000000000000)
487             >> 0x38]
488             << 0x38);
489     }
490
491     return 0;
492 }

493 int mea_invSubBytes(mea_t *mea_ctx) {
494     for (int i = 0; i < MEA_NW_STATE; i++) {
495         mea_ctx->m_state[i] =
496             mea_invSbox[0x00][(mea_ctx->m_state[i] & 0x00000000000000FF)] |
497             ((uint64_t)
498             mea_invSbox[0x01]
499                 [(mea_ctx->m_state[i] & 0x000000000000FF00) >>
500                 0x08]
501                 << 0x08) |
502                 ((uint64_t)
503                 mea_invSbox[0x02]
504                     [(mea_ctx->m_state[i] & 0x0000000000FF0000) >>
505                     0x10]
506                     << 0x10) |
507                     ((uint64_t)
508                     mea_invSbox[0x03]
509                         [(mea_ctx->m_state[i] & 0x00000000FF000000) >>
510                         0x18]
511                         << 0x18) |
512                         ((uint64_t)

```

```

505         mea_invSbox [0x00]
506             [(mea_ctx->m_state[i] & 0x000000FF00000000) >>
507             0x20]
508             << 0x20) |
509             ((uint64_t)
510                 mea_invSbox [0x01]
511                     [(mea_ctx->m_state[i] & 0x0000FF0000000000) >>
512                     0x28]
513                     << 0x28) |
514                     ((uint64_t)
515                         mea_invSbox [0x02]
516                             [(mea_ctx->m_state[i] & 0x00FF000000000000) >>
517                             0x30]
518                             << 0x30) |
519                             ((uint64_t)
520                                 mea_invSbox [0x03]
521                                     [(mea_ctx->m_state[i] & 0xFF00000000000000) >>
522                                     0x38]
523                                     << 0x38);
524     }
525
526     return 0;
527 }
528
529 int mea_dimRotate(mea_t *mea_ctx, uint8_t dim) {
530     uint8_t i, s, k;
531     uint8_t tmp[MEA_MS_IN_DIM];
532     uint8_t *pmstate = __wordsToBytes(mea_ctx->m_state);
533
534     for (int l = 0; l < MEA_MS_IN_DIM; l++) {
535         tmp[l] = pmstate[MEA_MS_IN_DIM * dim + l];
536     }
537
538     for (i = 0; i < MEA_MS_ROW; i++) {
539         for (s = 0; s < MEA_MS_ROW; s++) {
540             k = s + 1;
541             pmstate[MEA_MS_IN_DIM * dim + MEA_MS_ROW * (MEA_MS_ROW - k) + i] =
542                 tmp[MEA_MS_ROW * i + s];
543         }
544     }
545
546     mea_ctx->m_state = __bytesToWords(pmstate);
547     return 0;
548 }
549
550 int mea_invDimRotate(mea_t *mea_ctx, uint8_t dim) {
551     uint8_t i, s, k;

```

```

548     uint8_t tmp[MEA_MS_IN_DIM];
549     uint8_t *pmstate = __wordsToBytes(mea_ctx->m_state);
550
551     for (int l = 0; l < MEA_MS_IN_DIM; l++) {
552         tmp[l] = pmstate[MEA_MS_IN_DIM * dim + l];
553     }
554
555     for (i = 0; i < MEA_MS_ROW; i++) {
556         for (s = 0; s < MEA_MS_ROW; s++) {
557             k = s + 1;
558             pmstate[MEA_MS_IN_DIM * dim + MEA_MS_ROW * i + s] =
559                 tmp[MEA_MS_ROW * (MEA_MS_ROW - k) + i];
560         }
561     }
562
563     mea_ctx->m_state = __bytesToWords(pmstate);
564     return 0;
565 }
566
567 int mea_mixColumns(mea_t *mea_ctx) {
568     __matrixMultiplywState(mea_ctx, mds_matrix);
569     return 0;
570 }
571
572 int mea_invMixColumns(mea_t *mea_ctx) {
573     __matrixMultiplywState(mea_ctx, mds_inv_matrix);
574     return 0;
575 }
576
577 int mea_xorRoundKey(mea_t *mea_ctx, int round) {
578     for (int i = 0; i < MEA_NW_STATE; i++) {
579         mea_ctx->m_state[i] = mea_ctx->m_state[i] ^ mea_ctx->r_keys[round][
580             i];
581     }
582     return 0;
583 }
```

### 8.1.5 main.c

```

1 /*
2  * Projekt : MEA
3  * Autor : Michael Engel
4  * Datei : main.c
5 */
6
7 #include <stdio.h>
8 #include <stdint.h>
```

```
9
10 #include "mea.h"
11 #include "tables.h"
12
13 int print(uint64_t *input);
14
15 int main(){
16     uint64_t plain[8] = {0xd23412e140d67e3e, 0x09671b7823148bee, 0
17         x0c2549512aed62fb, 0x033152cb267d449e,
18         0xff7a6618caa9e1b8, 0x4de9e7b02bfe66e8, 0
19         x4313b4ed71bf8735, 0x35ea92cd2f442bfc};
20
21     uint64_t key[8] = {0x8ff47276b13a6427, 0xf8c902c9acb386bb, 0
22         x9d9be5eac2575ac1, 0x5ac16c57cb722825,
23         0x984f111a6a1c0cf4, 0x1c379112094de69a, 0
24         xa573aa28564707b2, 0x263c23787ef5323d};
25
26     uint64_t cipher[8];
27     uint64_t dcipher[8];
28
29     mea_t* ctxenc_mea = mea_init();
30     mea_t* ctxdec_mea = mea_init();
31
32     mea_generateRKeys(ctxenc_mea, key);
33     mea_generateRKeys(ctxdec_mea, key);
34
35     mea_blockEncipher(ctxenc_mea, plain, cipher);
36     mea_blockDecipher(ctxdec_mea, cipher, dcipher);
37
38     printf("%s\n", "Daten:");
39     print(plain);
40
41     printf("%s\n", "Verschluesselte Daten:");
42     print(cipher);
43
44     printf("%s\n", "Entschluesselte Daten:");
45     print(dcipher);
46
47     mea_del(ctxenc_mea);
48     mea_del(ctxdec_mea);
49
50     return 0;
51 }
52
53 int print(uint64_t *input){
54     for(int i = 0; i < MEA_NW_STATE; i++){
55         printf("%llx", input[i]);
56     }
57 }
```

```
53     printf("\n\n");
54     return 0;
55 }
56 }
```

# Literatur

- [1] Federal Information Processing Standards Publication 197. "Announcing the ADVANCED ENCRYPTION STANDARD (AES)". In: *NIST GOV* (2001).
- [2] Oleksandr Kazymyrov und Valentyna Kazymyrova und Roman Oliynykov. "A Method For Generation Of High-Nonlinear S-Boxes Based On Gradient Descent". In: *Cryptology ePrint Archive* (2013).