AES – Eine Einführung in Kryptographie

Andreas Hofmeier

Betreuer:	Prof. Dr. Thomas Risse
Studiengang:	ESTI, 7. Semester
Martrikelnummer:	94453

Zusammenfassung

Im ersten und zweiten Abschnitt dieses Dokumentes wird eine Übersicht über die Grundlagen der Kryptographie gegeben. Es werden Summen-, Permutations- und Substitutionsverfahren erklärt.

Dieses Wissen wird im dritten Abschnitt angewendet, um AES, den neuen Verschlüsselungsstandard, zu untersuchen.

Der vierte Abschnitt gibt einen Überblick über grundlegende Methoden, AES anzuwenden. Dies wird im fünften Abschnitt konkretisiert, indem die Erstellung eines verschlüsselten Containers (Dateisystem) unter GNU Linux beschrieben wird.

Inhaltsverzeichnis

1 Was ist Kryptographie?

Kryptographie[1] bezeichnet Verfahren, welche entwickelt wurden, um Informationen vor unbefugtem Zugriff zu schützen. Dies ist notwendig, da es in der digitalen Welt möglich ist, Daten ohne nennenswerten Aufwand zu vervielfältigen. Dies hinterlässt im Allgemeinen keine Spuren an den Daten. Soll ein öffentliches Netzwerkes zur Datenübertragung genutzt werden, ist der Einsatz von Kryptographie anzuraten, da es nicht möglich ist, seine Nachricht in einen Briefumschlag zu stecken. Heutige Datennetzwerke sind eher mit dem Versand von Postkarten zu vergleichen: Jeder, der sich in den Weg der Postkarte (oder der Datenübertragung) einklinken kann, ist in der Lage mitzulesen. Wie bereits erwähnt, entstehen dabei keine Spuren an den übermittelten Daten.

Um eine Nachricht vor den Augen Neugieriger zu schützen, wandelt Kryptographie die Nachricht mit Hilfe eines mathematischen Verfahrens in die Chiffre[2] um. Wer die Nachricht lesen will, muss über ein Verfahren verfügen, welches diese Umwandlung rückgängig macht.

Soll sichergestellt werden, dass nur ein einziger Empfänger die Nachricht lesen kann, müssen sich Sender und Empfänger ein einzigartiges mathematisches Verfahren teilen. Es wäre sehr aufwendig, so viele mathematische Verfahren zu entwickeln und auf Sicherheit hin zu überprüfen. Aus diesem Grund wird das mathematische Verfahren so gestaltet, dass es von einem weiteren Parameter abhängig ist: dem Schlüssel. Die gleiche Nachricht, mit dem selben Verfahren aber mit unterschiedlichen Schlüsseln verschlüsselt, führt zu verschiedenen Chiffren. Um die Nachricht lesen zu können, muss auf der Empfängerseinte nicht nur das Verfahren, sondern auch der Schlüssel bekannt sein.

Diese Klasse von Verfahren wird als symmetrische Verschlüsselung bezeichnet, da zum Ver- und Entschlüsseln derselbe Schlüssel verwendet wird.

Von der symmetrischen Verschlüsselung wird die unsymmetrische Verschlüsselung unterschieden. Eine weiterführende Einführung in die Grundprinzipien der Kryptographie (symmetrische und unsymmetrische Verschlüsselung) wird unter `http://www.abmh.de/fhs/crypt/DerTunnel/DerTunnel.PDF.pdf` gegeben.

Diese Dokument beschäftigt sich dagegen mit AES, dem Advanced Encryption Standard, einem Standardverfahren zur symmetrischen Verschlüsselung.

[1] Verschlüsselung, Geheimcode

[2] oder Geheimschrift

2 Grundlegende Verfahren der Verschlüsselung

2.1 Summenverfahren

Bei dem Summenverfahren werden der Klartext und der Schlüssel aufsummiert um die Chiffre zu erhalten. Die Chiffre wird wieder lesbar gemacht, indem der Schlüssel von der Chiffre abgezogen wird.

Beispiel:

- Verschlüsselung
  Klartext (`6124627`) + Schlüssel (`4531773`)

$$\begin{array}{r} 6124627 \\ +4531773 \\ \hline 10656400 \end{array}$$

  Ergebnis: Chiffre (`10656400`)

- Entschlüsselung
  Chiffre (`10656400`) minus Schlüssel (`4531773`)

$$\begin{array}{r} 10656400 \\ -4531773 \\ \hline 6124627 \end{array}$$

  Ergebnis: Klartext (`6124627`)

Das Aufsummieren von Nachrichten gestaltet sich im allgemeinem schwieriger. Der Grund dafür ist, dass Nachrichten bzw. Daten normalerweise nicht als "eine Zahl" vorliegen, sondern wesentlich komplexere Strukturen aufweisen. So werden Nachrichten meist in Sprache formuliert. Diese Sprache besteht aus Sätzen, welche wiederum aus Wörtern bestehen. Wörter ihrerseits bestehen aus Buchstaben. In Computern werden Daten (also auch Nachrichten) auf unterster Ebene als Byte Strings (Aneinanderreihungen von Zeichen) behandelt. Solch ein Byte String kann mathematisch als Zahl aufgefasst werden. Praktisch übersteigt die größer dieser Zahl schnell alle technisch handhabbaren Dimensionen. Daher wird bei der Verschlüsselung nicht der gesamte Byte String als Zahl betrachtet, sondern jedes Byte für sich alleine.

Beispiel:

- Verschlüsselung
  Klartext (“`TREFFEN UM SIEBEN`”) + Schlüssel (“`ZEBRA`”)

```
  TREFFEN UM SIEBEN
+            ZEBRA
---------------------
       ???????????
```

.

Die folgenden Annahmen werden getroffen, um die Nachricht verschlüsseln zu können:

1. Jeder Buchstabe der Nachricht wird einzeln verschlüsselt.
2. Jedem Buchstabe wird aufgrund seine Position im Alphabet eine Zahl zugeordnet: `A = 0, B = 1, ... Z = 25, Freiraum = 26`.
3. Ist die Nachricht länger als der Schlüssel, wird dieser wiederholt angewendet.

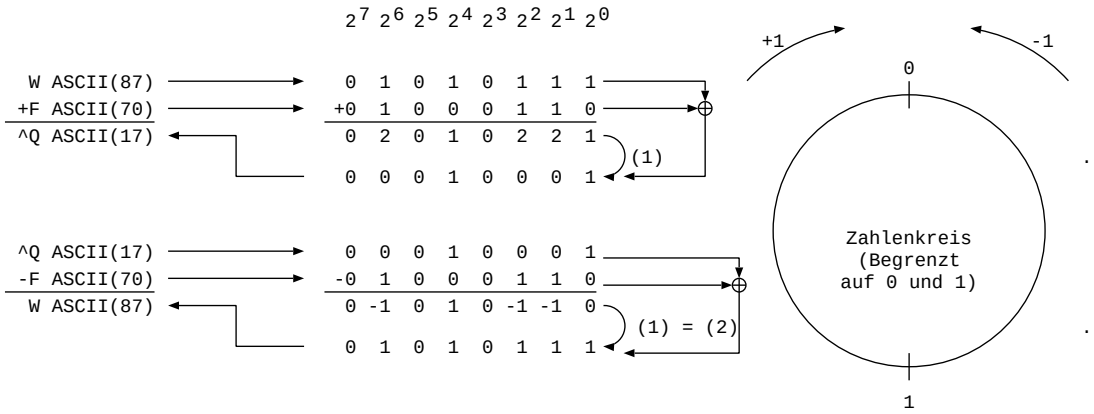
```
  TREFFEN UM SIEBEN  ------------>   19 17 04 05 05 04 13 26 20 12 26 18 08 04 01 04 13
 +ZEBRAZEBRAZEBRAZE  ------------>  +25 04 01 17 00 25 04 01 17 00 25 04 01 17 00 25 04
-------------------                ------------------------------------------------------
  RVFWFCRAKMYWJVBCR  <------------   44 21 05 22 05 29 17 27 37 12 51 22 09 21 01 29 17 )
                                                                                        (1)
                                     17 21 05 22 05 02 17 00 10 12 24 22 09 21 01 02 17 )
```

Wie im Beispiel zu erkennen ist, ergeben sich bei einigen Additionen Werte über 26. Diese können scheinbar keinem Buchstaben zugeordnet werden. Diesem Problem wird mit Hilfe der modulo27-Operarion abgeholfen.

Soll die Chiffre entschlüsselt werden, müssen die entgegengesetzten Rechenoperationen durchgeführt werden. Im Beispiel heißt dies, den Schlüssel von der Chiffre abzuziehen. Dazu werden den Buchstaben erneut Zahlen zugewiesen, welche dann voneinander subtrahiert werden können. Auch bei dieser Operation entsteht das Problem des begrenzten Zeichenvorrats, dem wieder durch modulo27 abgeholfen wird.

```
  RVFWFCRAKMYWJVBCR  ------------>   17 21 05 22 05 02 17 00 10 12 24 22 09 21 01 02 17
 -ZEBRAZEBRAZEBRAZE  ------------>  -25 04 01 17 00 25 04 01 17 00 25 04 01 17 00 25 04
-------------------                ------------------------------------------------------
  TREFFEN UM SIEBEN  <------------   -08 17 04 05 05-23 13-01-07 12-01 18 08 04 01-23 13 )
                                                                                        (2)
                                     19 17 04 05 05 04 13 26 20 12 26 18 08 04 01 04 13 )
```

Das folgende Beispiel zeigt, wie jedes Bit eines Bytes einzeln addiert wird. Es wird also nicht die gesamte Zahl, bestehend aus acht Bit, addiert, sondern die Bits unabhängig voneinander:



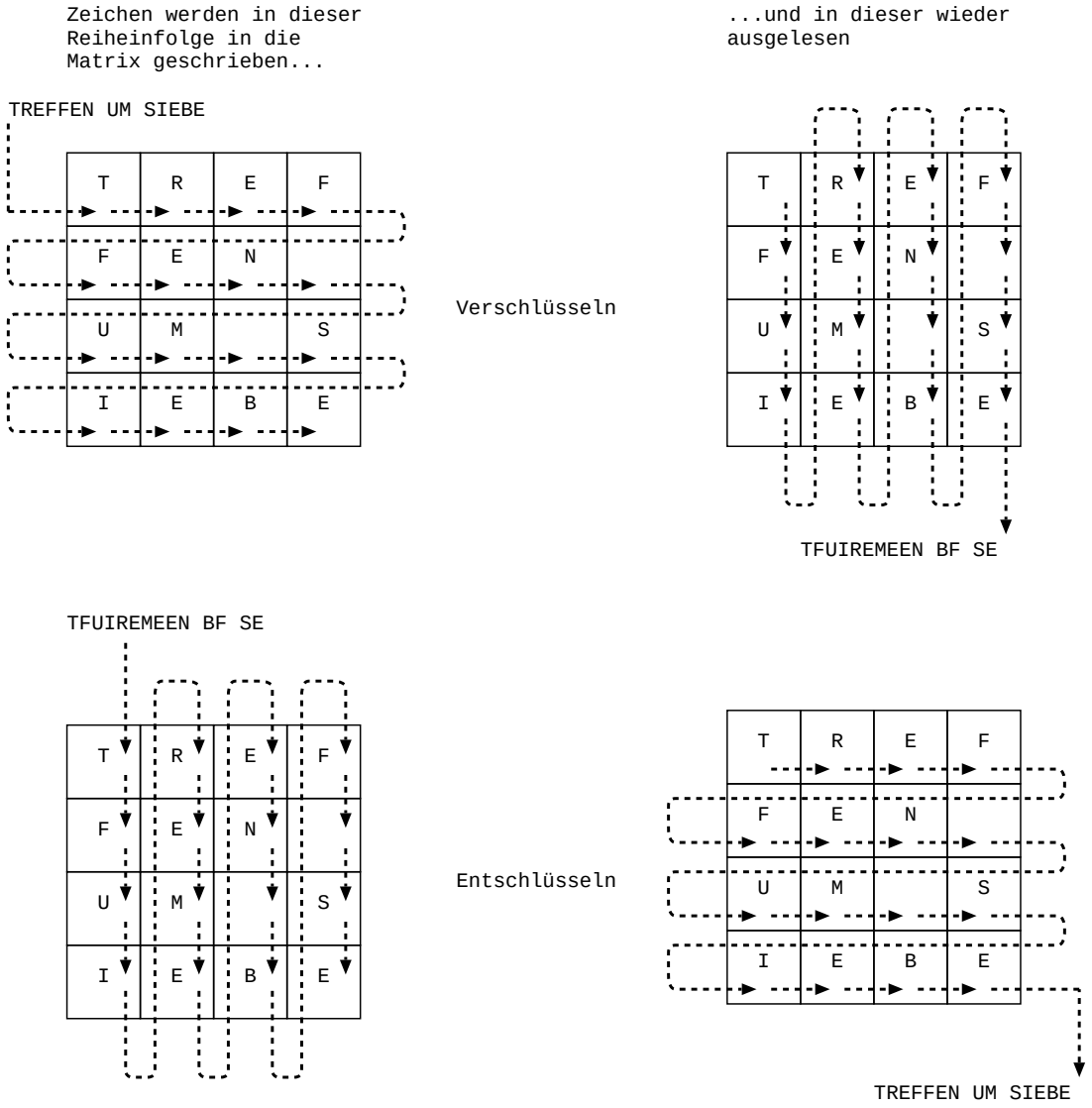
XOR = ⊕

a	b	x
0	0	0
0	1	1
1	0	1
1	1	0

Auf Grund der Symmetrie ist die Rechenoperation (1) und (2) in diesem Beispiel identisch. Die Addition zweier Bits mit anschließender Operation (1) ist eine weitläufig bekannte Rechenoperation in Computern: Die XOR oder Exklusiv-Oder-Operation. XOR kommt auch heutzutage bei Verschlüsselung sehr häufig zum Einsatz.

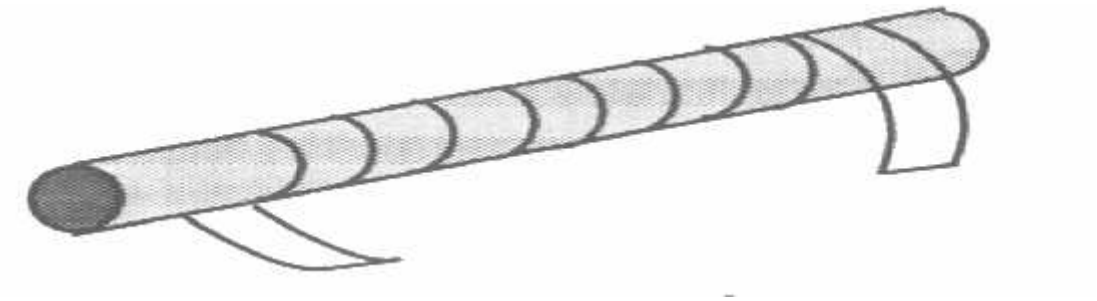
2.2 Permutationsverfahren

Das Permutationsverfahren stellt eine weitere Möglichkeit dar, Nachrichten zu verschlüsseln. Bei diesem Verfahren werden die Positionen der Zeichen verändert, nicht aber die Zeichen selber. Dies kann beispielsweise erreicht werden, indem die Nachricht von rechts nach links und von oben nach unten in eine Matrix geschrieben wird und danach von oben nach unten und von rechts nach links wieder herausgelesen wird.



Natürlich sind alle möglichen Formen von “Matrizen” und Aus- bzw. Einlesemustern vorstellbar. Deters (2002) beschreibt zum Beispiel eine historische Anwendung dieses Verfahrens: Die Regierung von Sparta zur hat bereits 500 v. Chr.

mit Hilfe eines Holzstabes (Skytale) verschlüsselt. Die Nachricht wurde vom Sender der Länge nach auf einen mit einem Pergamentsteifen umwickelten Holzstab geschrieben. Danach wurde das Pergament abgewickelt, nochmals abgeschrieben und versandt. Der Empfänger musste nun die empfangene Nachricht auf einen Pergamentstreifen schreiben und diesen um einen Holzstab mit dem richtigen Durchmesser wickeln, um die Nachricht zu entschlüsseln.



Beispiel einer historischen Verschlüsselung. Übernommen von Deters (2002).

2.3 Substitutionsverfahren

Das Substitutionsverfahren ersetzt Elemente der Nachricht nach bestimmten Regeln gegen andere Elemente. Deters (2002) führt hier das älteste bekannte Beispiel einer Verschlüsselung auf: die Cäsar-Chiffre. Diese Methode ist nach Cäsar benannt. Er schrieb das Alphabet zweimal untereinander, wobei er es beim zweiten Mal um drei Buchstaben nach rechts einrückte. (Das Lateinischen Alphabet verfügt nicht über die Buchstaben J, K, W, Y und Z.):

```
A B C D E F G H I L M N O P Q R S T U V X
U V X A B C D E F G H I L M N O P Q R S T
```

Übernommen von Deters (2002).

Um eine Nachricht zu verschlüsseln, werden alle Zeichen der Nachricht durch das korrespondierende Zeichen ersetzt:

```
TREFFEN UM SIEBEN ← Klartext
QOBCCBI RH PFBVBI ← Chiffre
```

3 Advanced Encryption Standard (AES)

3.1 Geschichte und Anforderungen an AES

AES ist der in einem öffentlichen Auswahlverfahren vom NIST[3] bestimmte Nachfolger von DES (Data Encryption Standard). Ein neuer Standard für Datenverschlüsselung wurde notwendig, da es laut Wilkin (2004) der Electronic Frontier Foundation 1998 gelungen war, eine Maschine zu bauen, die DES mittels einer Brute-Force-Attack aushebeln konnte. Im Gegensatz zu DES war die Auswahl von AES vollständig öffentlich. Jeder Interessierte konnte seinen Vorschlag in Form eines Verschlüsselungsalgorithmus einschicken oder sich an dem Test der Algorithmen beteiligen. Die NIST suchte den besten Algorithmus aus, wobei folgende Kriterien in Betracht gezogen wurden:

1. Sicherheit
   Hierbei handelte es sich um das wichtigste Auswahlkriterium. Es sollten keine Algorithmen verwendet werden, die mit einem bekannten Verfahren schneller als mit Brute-Force-Attack[4] ausgehebelt werden konnten. Es ist allerdings nicht möglich nachzuweisen, dass ein Algorithmus sicher ist. Nur das Gegenteil kann bewiesen werden.

2. Kosten

   - Der Algorithmus sollte vollkommen (kosten-) frei verwendbar sein. (frei von Patent- oder anderen Ansprüchen der Entwickler)
   - Die Implementierung und das Ver- bzw. Entschlüsseln sollte so billig wie möglich sein.

3. Eigenschaften der Implementierung

   - Der Algorithmus sollte sowohl in Hardware als auch in Software mit geringsten möglichen Mitteln umgesetzt werden können, also Platformunabhängig gestaltet sein.
   - Die Schlüssellänge sollte zwischen 128, 192 und 256 Bit gewählt werden können.

[3] U.S. National Institute of Standards and Technology (www.nist.gov)

[4] Bei dieser Form der Attacke werden alle möglichen Passwörter durchprobiert. Ist das Password zufällig gewählt, müssen im Mittel die Hälfte aller Passwörter durchprobiert werden, um es zu finden. Einfachere Passwörter, wie Wörter aus einem Wörterbuch, können mit dem durchprobieren aller bekannten Wörter schneller gefunden werden als Zufallsstrings.

- Die Blockgröße sollte 128 Bit betragen.
- Das Verfahren sollte einfach und verständlich sein.

3.2 Kandidaten und Auswahlverfahren

Die folgende Tabelle gibt eine Übersucht über alle Algorithmen, welche für den neuen Verschlüsselungsstandard AES vorgeschlagen wurden, von wem sie vorgeschlagen wurden und in welcher Runde sie ausgeschieden sind.

Name	eingereicht von		ausge. Runde
CAST-256	Entrust (CA)	Company	1^{1}
Crypton	Future Systems (KR)	Company	1^{2}
DEAL	Outerbride, Knudsen (USA-DK)	Forscher	1^{sp}
DFC	ENS-CNRS (FR)	Forscher	1^{3}
E2	NTT (JP)	Company	1^{4}
Frog	TecApro (CR)	Company	1^{s}
HPC	Schroeppel (USA)	Forscher	1^{s}
LOKI97	Brown et al. (AU)	Forscher	1^{s}
Magenta	Deutsche Telekom (DE)	Company	1^{sp}
Mars	IBM (USA)	Company	2
RC6	RSA (USA)	Company	2
Rijndael	Daemen and Rijmen (BE)	Forscher	WINNER
SAFER+	Cylink (USA)	Company	1^{p}
Serpent	Anderson, Biham, Kundsen (UK-IL-DK)	Forscher	2
Twofish	Counterpane (USA)	Company	2

Adaptiert von Daemen und Rijmen (2002).

1) Ausgeschieden in der ersten Runde. Auswahl der fünf Finalisten im März 1999.
2) Ausgeschieden in der zweiten Runde, Endausscheidung am 2. Oktober 2000
^{s}) Ausscheidung wegen massiver Sicherheitsprobleme
^{p}) Ausscheidung wegen massiver Leistungsprobleme
^{1}) Vergleichbar mit Serpent, Implementation aufwendiger.
^{2}) Vergleichbar mit Rijndael und Twofish aber unsicherer.
^{3}) Geringe Sicherheit, schlechte Performance außer auf 64-Bit Prozessoren.
^{4}) Vergleichbar mit der Struktur von Rijndael und Twofish, aufwendigere Implementation

Alle Kandidaten mussten bis zum 15. Mai 1998 eingereicht werden. Am 20. August begann dann die erste Runde, welche im März 1999 mit der Auswahl der fünf Finalisten endete. Am 2. Oktober 2000, nach einer weiteren Auswahl-Runde, wurde schließlich der Gewinner gekürt: Rijndael. Von nun an als AES bezeichnet. Dieser Algorithmus wurde Joan Daemen und Vincent Rijmen, zwei belgischen Wissenschaftlern, als Kandidat für die AES-Ausschreibung entwickelt.

Viele Beobachter waren darüber erstaunt, dass ein Algorithmus gewonnen hat, der weder aus den USA stammt noch von einer Firma aus dem Verschlüsselungsgeschäft entwickelt wurde. Von vielen Seiten wurde aus diesem Grund auf ein objektives Auswahlverfahren geschlossen, welches am Schluss den besten Algorithmus zum Gewinner kürte.

3.3 Aufbau

Die grundlegende Struktur des AES-Algorithmus ist in der folgenden Weise aufgebaut:

```
Rijndael(Data, Key) {
  KeyExpansion(Key, ExpandedKey);
  AddRoundKey(Data, ExpandedKey[0]);
  for (i = 0; i < N; i++) {
    Round(Data, ExpandedKey[i]);
  }
  FinalRound(Data, ExpandedKey[N]);
)

Round(Data, ExpandedKey) {
  SubBytes(Data);
  ShiftRows(Data);
  MixColums(Data);
  AddRoundKey(Data, ExpandedKey);
}

FinalRound(Data, ExpandedKey) {
  SubBytes(Data);
  ShiftRows(Data);
  AddRoundKey(Data, ExpandedKey);
}
```

Adaptiert von Daemen und Rijmen (2002).

Daemen und Rijmen (2002) betrachten Blöcke in der folgenden Weise. Es werden jeweils vier Bytes von oben nach unten geschrieben. Beim fünften Byte wird nach

rechts weitergegangen und wieder oben angefangen.

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

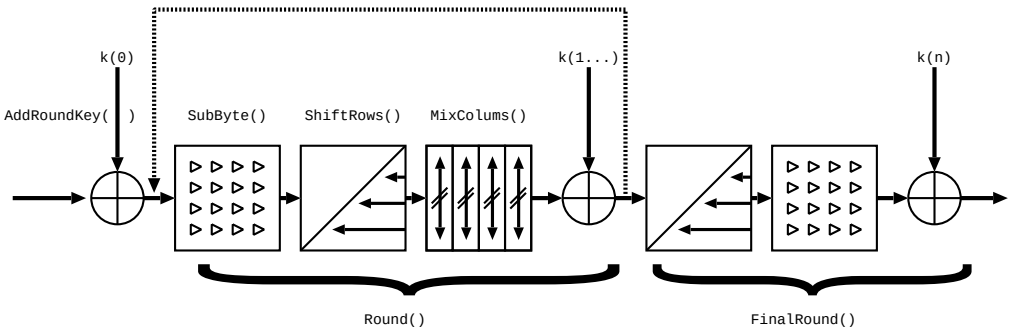
128 Bit-Block (16-Byte)

0	4	8	12	16	20	24	28
1	5	9	13	17	21	25	29
2	6	10	14	18	22	26	30
3	7	11	15	19	23	27	31

256 Bit-Block (32 Byte)

In der folgenden graphischen Darstellung wird von einer Blockgröße von 128 Bit ausgegangen. Die Blöcke ließen sich allerdings in Vier-Byte-Schritten (32 Bit) bis auf 256 Bit ausweiten. Dasselbe gilt für den Schlüssel. Daten-Blockgröße und Schlüssel-Block-Größe sind vollkommen unabhängig voneinander. Im AES-Standard sind allerdings lediglich Datenblockgrößen von 128 Bit und Schlüsselblocklängen von 128, 192 und 256 Bit vorgesehen. Dies ändert nichts daran, dass der Rijndael-Algorithmus mehr kann, was aber nicht notwendiger implementiert sein muss, wenn auf AES Bezug genommen ist.

Die graphische Darstellung verdeutlicht den Fluss der Daten:



Adaptiert von Daemen und Rijmen (2002).

3.3.1 Anzahl der Runden

Wie oben ersichtlich, wird die Funktion `Round()` N mal ausgeführt. Wobei N von der Schlüssel- und der Daten-Block-Größe abhängt. Die folgende Tabelle stellt die Länge des Schlüssels der Länge des Datenblockes gegenüber und gibt für jede Kombination eine Anzahl von Runden an. Alle fett dargestellten Fälle sind durch

den AEStandard definiert. Die restlichen Fälle werden von Rijndael unterstützt, sind aber im Standard nicht definiert.

Länge des Datenblockes	Länge des Schlüssels				
	128	160	**192**	224	**256**
128	**10**	11	**12**	13	**14**
160	11	11	12	13	14
192	12	12	12	13	14
224	13	13	13	13	14
256	14	14	14	14	14

3.3.2 Verwendung des Schlüssels: `KeyExpansion()`, `AddRoundKey()`

Vor der ersten und nach jeder Runde wird der Datenblock mit einem gleich großen Schlüsselblock ge`XOR`t. Dies wird mit Hilfe der Funktion `AddRoundKey()` durchgeführt. Die XOR-Funktion kann als eine Form des Summenverfahrens angesehen werden, welche im Abschnitt 2.1 behandelt wurde.

Die Funktion `KeyExpansion()` wurde implementiert, um den Schlüssel auf die richtige Länge zu erweitern. Denn sowohl der Schlüssel als auch der Datenblock können verschiedene, voneinander unabhängige, Längen aufweisen. Die Funktion erweitert die Länge des Schlüssels auf (Runden + 1) * Daten-Blockgröße, da vor der ersten und nach jeder Runde einmal ein Teil-Schlüssel von der Länge des Datenblockes gebraucht wird. Die hinzugefügten Bytes werden durch ein reproduzierbares mathematisches Verfahren erzeugt, welches recht unvorhersagbare vom Schlüssel abhängige Werte liefert. Diese Funktion kann mit einem Pseudo-Zufallszahlen-Generator verglichen werden. Derselbe Startwert führt immer zu der selben Abfolge von Zahlen, welche aber möglichst zufällig sein soll. So wird ein 128-Bit-Schlüssel auf den richtigen Wert (Hier: (Runden + 1) * Daten-Blockgröße) erweitert:

```
00000000,00000000,00000000,00000000
62636363,62636363,62636363,62636363
9b9898c9,f9fbfbaa,9b9898c9,f9fbfbaa
90973450,696ccffa,f2f45733,0b0fac99
ee06da7b,876a1581,759e42b2,7e91ee2b
7f2e2b88,f8443e09,8dda7cbb,f34b9290
ec614b85,1425758c,99ff0937,6ab49ba7
21751787,3550620b,acaf6b3c,c61bf09b
0ef90333,3ba96138,97060a04,511dfa9f
b1d4d8e2,8a7db9da,1d7bb3de,4c664941
b4ef5bcb,3e92e211,23e951cf,6f8f188e
      Uebernommen von Daemen und Rijmen (2002) und van Rees (2005)
```

Der hier aufgeführte erweiterte Schlüssel entsteht, wenn der übergebende Schlüssel aus Nullen (alle Bits sind Null) besteht.

Den Autor hat es verwundert, dass der Schlüssel nicht verwendet wurde, um die verändernden Funktionen (`SubBytes()`, `ShiftRows()` und `MixColums()`) direkt zu steuern. Stattdessen werden diese Funktionen immer in der gleichen Weise auf den Datenstrom abgewendet. Die Abhängigkeit vom Schlüssel wird über die Funktion `AddRoundKey()` eingeführt.

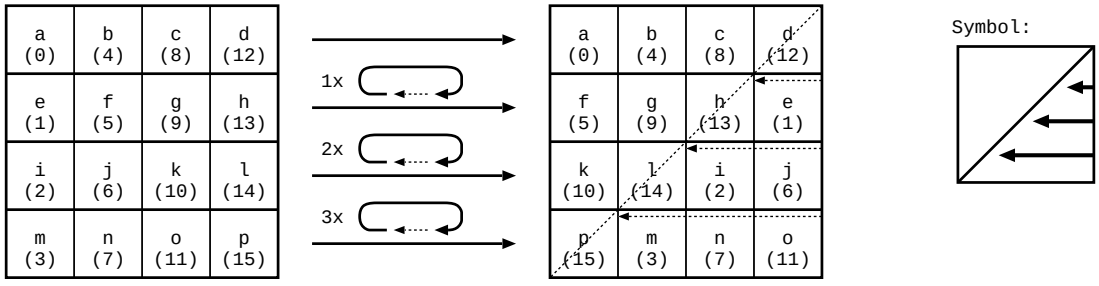
3.3.3 Permutationsschritt: `ShiftRows()`

Die Funktion `ShiftRows()` wendet ein Permutationsverfahren (siehe Abschnitt 2.2) auf den Datenstrom an. In diesem Fall wird der Block zeilenweise betrachtet. Bis auf die oberste wird jede Zeile um n Spalten nach links verschoben. Die Weite der Verschiebung hängt von der Blockgröße und der Zeile ab.

Die folgende Tabelle stellt die (Daten)Blockgröße mit den Zeilen gegenüber und gibt für jede Paarung einen Wert für die Linksverschiebung an. Im AEStandard ist lediglich der Fall des 128-Bit-Blocks definiert. Da der Rijndael-Algorithmus mit größeren Blöcken umgehen kann, ist dies hier ebenfalls aufgeführt.

	Länge des Datenblockes				
Zeile	**128**	160	192	224	256
1	**0**	0	0	0	0
2	**1**	1	1	1	1
3	**2**	2	2	2	3
4	**3**	3	3	4	4

Die folgende Graphik versucht dies am Beispiel eines 128Bit Blockes zu verdeutlichen.

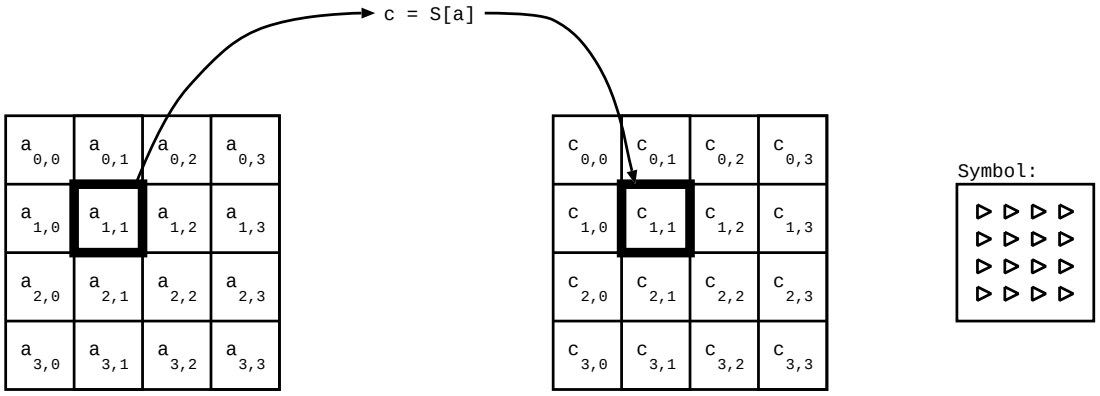


Die inverse Funktion (`InvShiftRows()`) schiebt die Zeilen um die gleiche Anzahl von Bytes in die entgegengesetzte Richtung.

3.3.4 Substitutionsschritt 1: `SubBytes()`

Die Funktionen `SubBytes()` und `MixColums()` wenden eine Form von Substitutionsverfahren (siehe Abschnitt 2.3) auf den Datenstrom an. Es besteht jedoch ein grundlegender Unterschied zu den in Abschnitt 2.3 beschriebenden Substitutionsverfahren. Die Substitution wird nicht wie beschrieben durch den Schlüssel bestimmt, sondern ist fest vorgegeben. Wie bereits erwähnt, wird die Abhängigkeit vom Schlüssel durch die XOR-Schritte bewirkt.

`SubBytes()` verarbeitet jedes Byte im Datenblock einzeln.



Adaptiert von Daemen und Rijmen (2002).

Es wird ein "Alphabet" von mit 256 (2^8 da 8Bit) Symbolen verwendet. Jedes der Symbole wird durch ein anderes Symbol (ebenfalls Teil des Alphabets) ersetzt. Dies Substitution ist durch den folgenden mathematischen Zusammenhang bestimmt:

Zuerst werden die Bits (a_7 bis a_0) in dem Byte (a) als Polynom betrachtet:

$$a = a_7a_6a_5a_4a_3a_2a_1a_0 \neq 0 \Rightarrow f(x) = a_7x^7+a_6x^6+a_5x^5+a_4x^4+a_3x^3+a_2x^2+a_1x^+a_0$$

Danach wird die Inverse dieses Polynoms gebildet. Diese Operation wird in $GF(2^8) = GF(256)$ ausgeführt.

$$g(x) = f(x)^{-1}$$

Dann wird das so gewonnene Polynom wieder in ein Byte zurückgewandelt:

$$g(x) = b_7x^7 + b_6x^6 + b_5x^5 + b_4x^4 + b_3x^3 + b_2x^2 + b_1x^+b_0 \Rightarrow b = b_7b_6b_5b_4b_3b_2b_1b_0$$

Für den nicht definierten Fall, dass a gleich 0 ist, wird angenommen, dass b ebenfalls 0 ist.

Danach werden die Bits des Bytes als Vektor betrachtet und der folgenden Vektor-Matrix-Multiplikation mit anschließendem XOR unterzogen:

$$c = \begin{bmatrix} c_7 \\ c_6 \\ c_5 \\ c_4 \\ c_3 \\ c_2 \\ c_1 \\ c_0 \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 \\ 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 \end{bmatrix} \times \begin{bmatrix} b_7 \\ b_6 \\ b_5 \\ b_4 \\ b_3 \\ b_2 \\ b_1 \\ b_0 \end{bmatrix} \oplus \begin{bmatrix} 0 \\ 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \end{bmatrix}$$

Die `InvSubBytes()` macht den so eben beschriebenen Vorgang rückgängig. Dazu wird zuerst die Vektor-Matrix-Multiplikation mit anschließendem XOR invertiert durchgeführt:

$$b = \begin{bmatrix} b_7 \\ b_6 \\ b_5 \\ b_4 \\ b_3 \\ b_2 \\ b_1 \\ b_0 \end{bmatrix} = \begin{bmatrix} 0 & 1 & 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 \\ 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 & 0 & 1 & 0 & 0 \end{bmatrix} \times \begin{bmatrix} c_7 \\ c_6 \\ c_5 \\ c_4 \\ c_3 \\ c_2 \\ c_1 \\ c_0 \end{bmatrix} \oplus \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 1 \\ 0 \\ 1 \end{bmatrix}$$

Danach wird das Ergebnis als Polynom betrachtet und invertiert. Dieser Schritt führt auf das originale a zurück.

Im Allgemeinen ist das Nachschlagen eines Wertes schneller als das Errechnen des selbigen. Dies trifft jedenfalls zu, wenn die Tabelle relativ klein ist. Dieses Vorgehen ist ebenfalls eine bei der Implementierung von AES angewendete Alternative zu der eben beschriebenen Rechnung. Denn die Tabelle besteht aus lediglich 256 (2^8) Einträgen. Praktisch kann die Liste in diesem Fall mit Hilfe eines 256 Byte langen Arrays dargestellt werden. Dazu wird an der jeweiligen Position im Array das zugehörige Ergebnis gespeichert. Wenn, zum Beispiel, die oben beschriebende mathematische Transformation mit einer Eingabe von 2 zu dem Ergebnis 119 führt, so steht an der dritten Stelle des Arrays eine 119. (Es handelt sich um die dritte Stelle, da im Computerbereich bei 0 angefangen wird zu Zählen.) Heiße das Array S, so ergibt sich folgender Zusammenhang:

```
output = S[input];
```

Wenn die Variable `input` mit 2 initialisiert wurde, steht nach dem Aufruf dieser Zeile 119 in `output`. Im Allgemeinen wird dieses Verfahren als Look-Up-Table bezeichnet. Die inverse Transformation funktioniert entsprechend, nur dass das entsprechende Array verwendet wird.

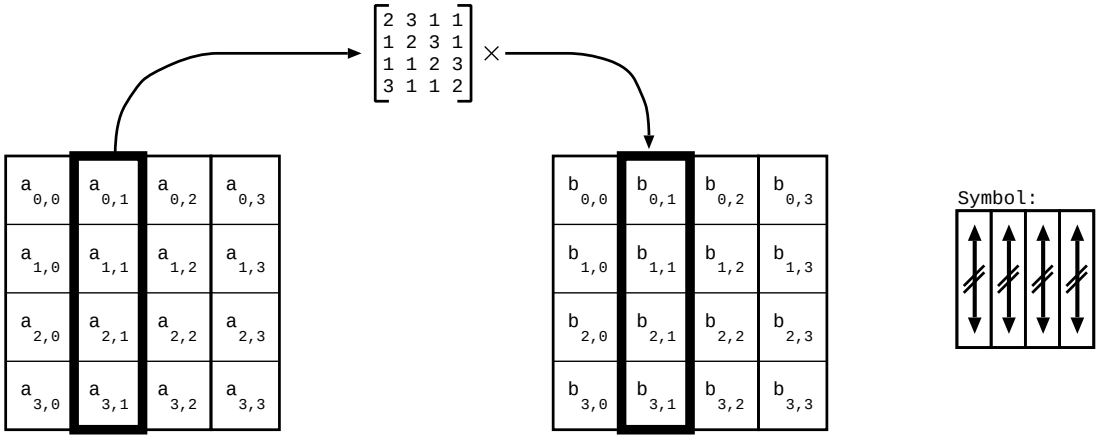
3.3.5 Substitutionsschritt 2: `MixColums()`

`MixColums()` wird jeweils auf 4-Bytes-Spalte angewendet. Wird diese Funktion als Substitution betrachtet, besteht das "Alphabet" aus 4 294 967 296 ($= 2^{32} = 2^{4*8}$) Symbolen. Bei dieser Menge an Symbolen ist an eine Verwendung einer Tabelle nicht mehr zu denken.

Die Umwandlung in diesem Schritt ist durch eine Matrix-Vektor-Multiplikation bestimmt. In diesem Fall wird ein Vektor von 4 Bytes mit einer Matrix von 4 × 4 Bytes Multipliziert:

$$b = \begin{bmatrix} b_{0,i} \\ b_{1,i} \\ b_{2,i} \\ b_{3,i} \end{bmatrix} = \begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix} \times \begin{bmatrix} a_{0,i} \\ a_{1,i} \\ a_{2,i} \\ a_{3,i} \end{bmatrix}$$

Wobei i der zu verarbeitenden Spalte entspricht. Im Falle eines 128 Bit Blockes sind vier Spalten zu verarbeiten.



Adaptiert von Daemen und Rijmen (2002).

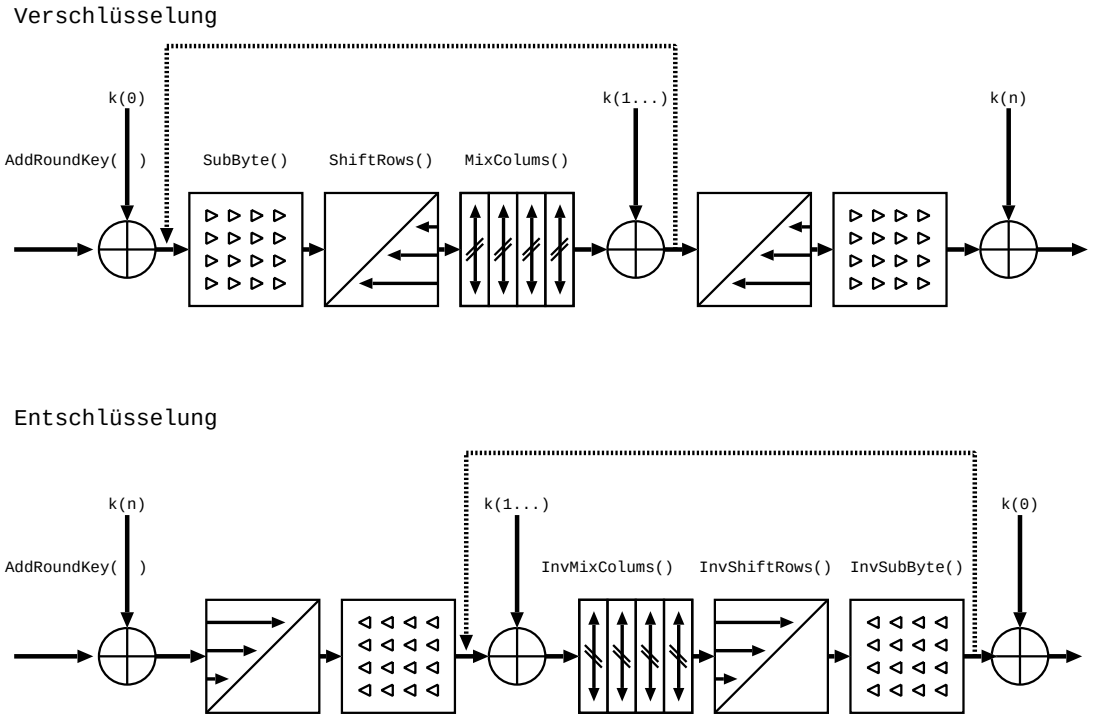
Um diese Umwandlung rückgängig zu machen, wurde die Funktion `InvMixColums()` entwickelt, die folgende mathematische Operation ausführt:

$$a = \begin{bmatrix} a_{0,i} \\ a_{1,i} \\ a_{2,i} \\ a_{3,i} \end{bmatrix} = \begin{bmatrix} 02 & 03 & 01 & 01 \\ 0E & 0B & 0D & 09 \\ 09 & 0E & 0B & 0D \\ 0B & 0D & 09 & 0E \end{bmatrix} \times \begin{bmatrix} b_{0,i} \\ b_{1,i} \\ b_{2,i} \\ b_{3,i} \end{bmatrix}$$

Die hier verwendete Matrix kann durch invertieren der in `MixColums()` verwendeten Matrix errechnet werden.

3.3.6 Entschlüsselung

Um die verschlüsselten Daten wieder in die Ausgangsdaten zurück zu wandeln, wird folgendermaßen vorgegangen: Die Reihenfolge der Schritte wird umgekehrt. Bei jedem Schritt wird der invertierende Schritt angewendet. Zum Beispiel `InvMixColums()` statt `MixColums()`. Die folgende Graphik versucht dies deutlich zu machen und stellt dazu den Ver- und Entschlüsselungsprozess gegenüber.



Adaptiert von Daemen und Rijmen (2002).

Es wurde gerade davon gesprochen, alle Schritte in umgekehrter Reihenfolge abzuarbeiten. In der Graphik wurden aber zwei Schritte nicht vertauscht. Warum?

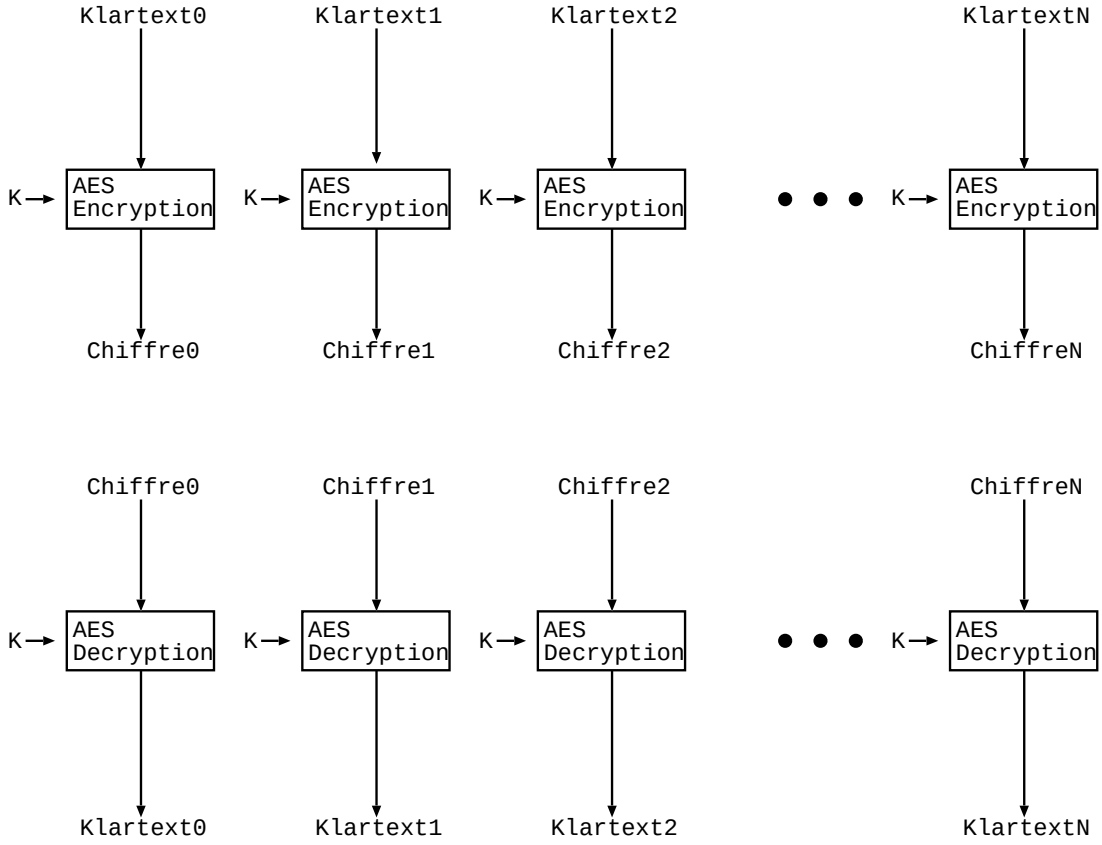
Es handelt sich um die letzten beiden Schritte bei der Verschlüsselung oder um die ersten beiden bei der Entschlüsselung: `ShiftRows()` und `SubBytes()`, bzw, `InvShiftRows()` und `InvSubBytes()`. Dies kann ganz einfach erklärt werden: Es macht keinen Unterschied, ob diese beiden Schritte getauscht sind oder nicht. `SubBytes()` wird auf jedes Byte einzeln angewendet, daher spielt es keine Rolle, ob `ShiftRows()` die Bytes vorher oder nachher verschiebt. Diese und andere hier nicht erwähnte Möglichkeiten, die Reihenfolge von Operationen zu vertauschen, werden zur Optimierung der Implementation verwendet.

4 Anwendung grundlegender Modi auf AES

AES verschlüsselt "nur" 128-Bit-Blöcke. Nur welche Nachrichten bzw. Dateien sind schon 128 Bit (16 Byte) lang?

4.1 ECB – Electronic Code Book

Es gibt die Möglichkeit, die zu verschlüsselnde Datei oder Nachricht in 128-Bit-Blöcke aufzuteilen. Wird AES auf jeden dieser Blöcke unabhängig angewendet, so nennt man dies das elektronische Code-Buch Verfahren (engl. ECB – Electronic Code Book):

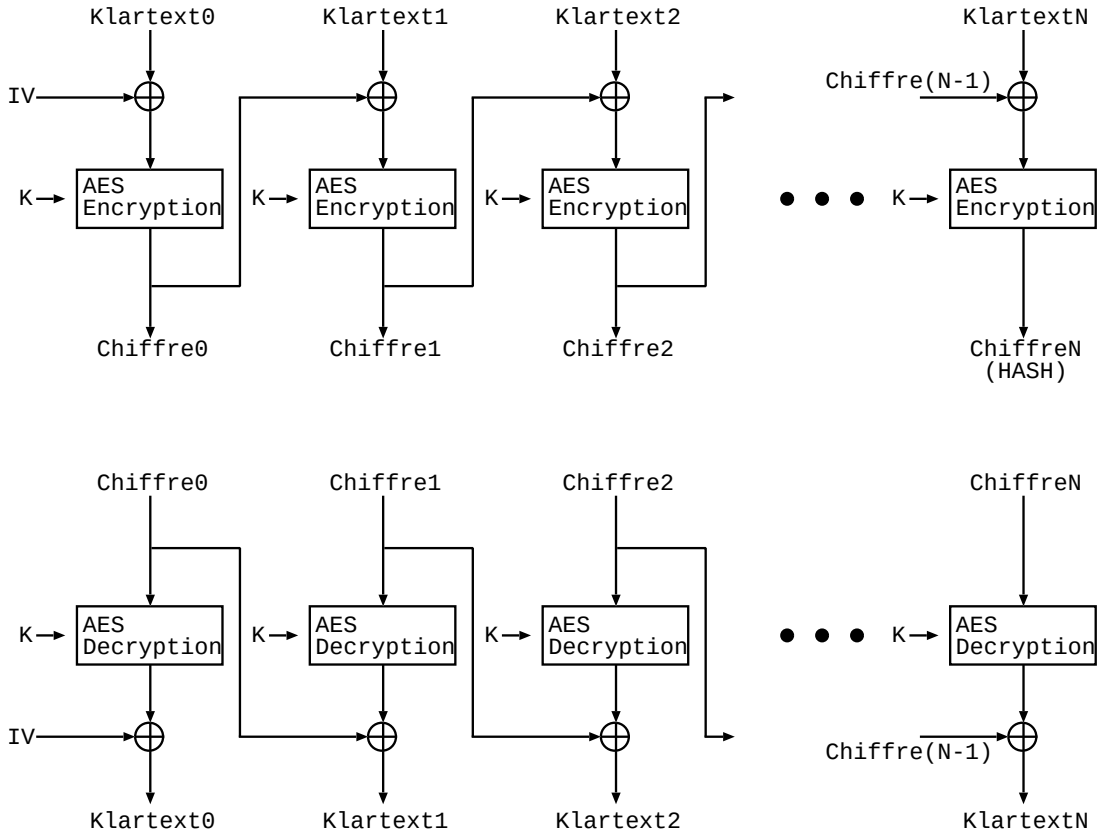


Diese Unabhängigkeit der Blöcke führt dazu, dass identische Klartext-Blöcke bei der Verschlüsselung die selben Chiffre-Blöcken ergeben. Die Verschlüsselung könnte als eine Form der Substitution vergleichen werden, nur dass das Alphabet in diesem Fall über $340282366920938463463374607431768211456 = 2^{128} \approx 3.4 * 10^{38}$ Symbole verfügt. Es wäre also von der Kapazität her nicht möglich, einer Liste

mit dieser Menge an Einträgen zu verwalten. Allerdings ist es möglich, häufig auftretende gleiche Blöcke zu identifizieren. Dies ist ein möglicher Angriffspunkt auf die Verschlüsselung.

4.2 CBC – Cipher Block Chaining

Um diesem Effekt vorzubeugen, werden die Blöcke im CBC-Mode voneinander abhängig gemacht. Dies geschieht, indem der Input vor der Verschlüsselung mit dem Output der vorherigen Verschlüsselung geXORt wird:



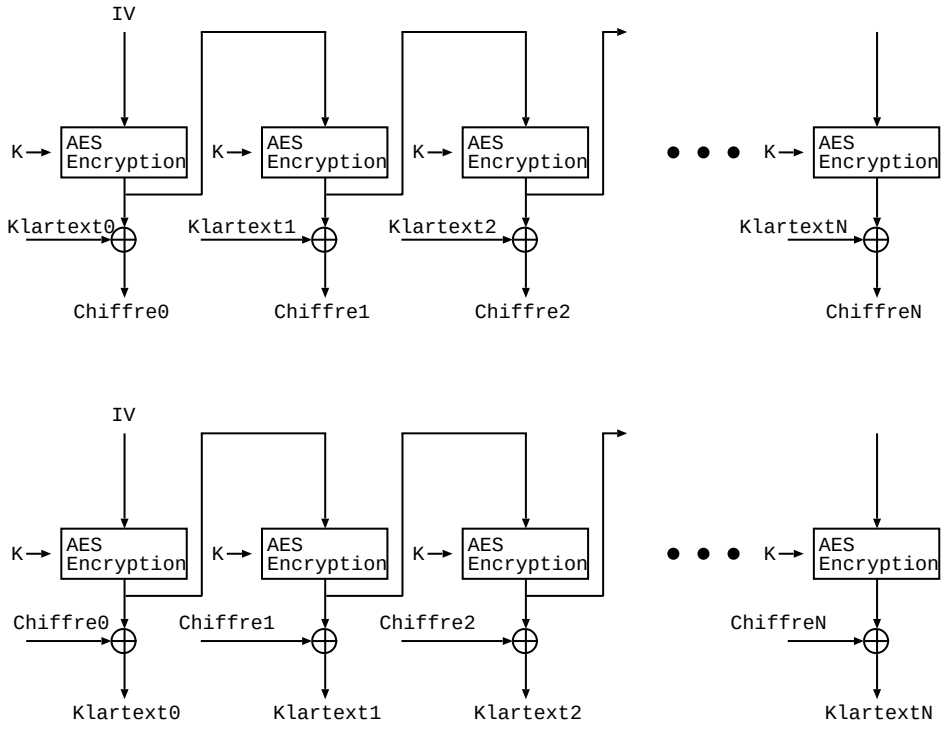
Die Chiffre vom aktuellen Klartextblock ist in diesem Fall nicht nur vom Klartextblock und dem Passwort abhängig, sondern zusätzlich vom vorherigen Chiffre-Block. Dieser wiederum ist seinerseits wieder abhängig vom Passwort, vom Klartextblock und vom vorherigen Chiffre-Block. Der erste Chiffre-Block ist seinerseits vom Passwort, dem Klartext-Block und dem IV abhängig. Der IV (Initial Vector) ist ein Anfangswert, welcher als Ersatz für den vorherigen Block mit dem Klartext geXORt wird. Die letzte Chiffre ist allerdings vom Password und von allen vorherigen und dem letzten Klartext abhängig. Aus diesem Grund könnte die letzte Chiffre auch als HASH-Wert des Datenstromes betrachtet werden.

Nachteile dieser beiden Lösungen sind, dass

- die Länge der Nachricht immer ein Vielfaches der Blockgröße sein muss. Ist dies nicht der Fall, muss der letzte Block aufgefüllt (engl. padding) werden. Bei einer Blockgröße von 16 Byte müssten maximal 15 Bytes aufgefüllt werden, was in den meisten Fällen ein vertretbarer Wert ist.
- die Nachricht immer vollständig bis zu dem gewünschten Block entschlüsselt werden muss. Soll, zum Beispiel, nur der letzte Block einer Datei gelesen werden, muss trotzdem die gesamte Datei entschlüsselt werden. Natürlich muss die Datei zu diesem Zweck vollständig vorliegen. Ist ein Block der Datei beschädigt, sind dieser und alle folgenden Blöcke unlesbar.

4.3 Key Stream Generators

Ein andere Möglichkeit ist, den Verschlüsselungsalgorithmus zu verwenden, um einen Schlüsselstrom zu erzeugen. Die zu verschlüsselnden Daten werden danach mit diesem Schlüsselstrom geXORt und somit verschlüsselt:

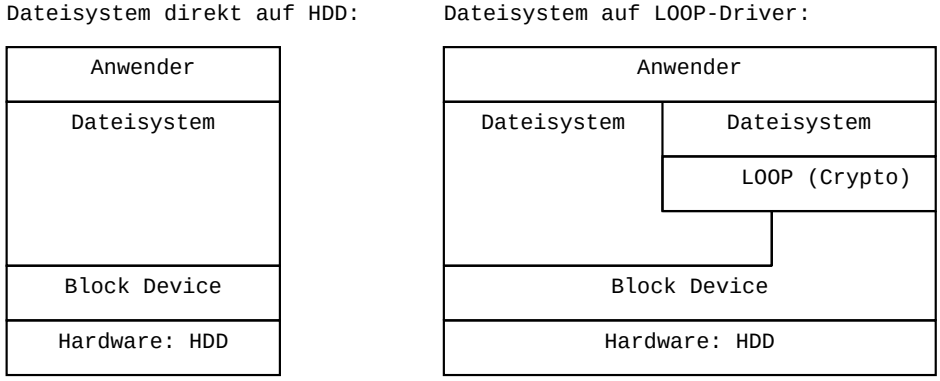


Vorteil dieser Variante ist, dass die Länge der Nachricht kein Vielfaches von der Blockgröße sein muss. Denn der letzte Block muss nicht vollständig sein. Ist der Block nur ein Byte lang, so kann dieses Byte mit dem ersten Byte aus dem aktuellen Keystream-Block verschlüsselt werden. Die restlichen Bytes des Keystream-Blocks bleiben dabei einfach ungenutzt.

5 Konkrete Anwendung: Crypto-LOOP-Driver in Linux

Der Loop[5]-Driver unter Linux wurde entwickelt, um normale Dateien oder andere Blockdevices[6] als virtuelle Blockdevices abzubilden. Dies ist notwendig, wenn ein Image[7] gemountet[8] werden soll. Auf Grund der blockorientierten Struktur kann eine Datei nicht direkt gemountet werden.

Die folgende Graphik versucht diesen Zusammenhang nochmals zu verdeutlichen.



Auf der linken Seite ist die “normale” Zugriffsstruktur abgebildet: Der Anwender greift auf das Dateisystem zu, welches in ein Verzeichnis gemountet ist. Das Dateisystem seinerseits verwendet die Blockdevices als Schnittstelle zur Hardware. Rechts wurde diese Struktur um ein virtuelles Blockdevice (Loop-Device)

[5]Das Wort “Loop” kommt aus dem Englischen und bedeutet soviel wie Schleife.

[6]Schnittstelle zum Zugriff auf blockorientierte Geräte wie Disketten, Festplatten (Partitionen), CD-ROMs, etc. Blockorientiert bedeutet, dass nur Blockweise auf diese Geräte zugegriffen werden kann.

[7]Ein Image ist eine Abbildung eines Blockgerätes (z.B. einer Partitionen) in Form einer Datei im Dateisystem

[8]Mounten bezeichnet den Vorgang des Verbindens eines Dateisystems auf einem Blockdevice mit einem Verzeichnis. Z.B. Mounten einer Diskette nach /mnt: `mount /dev/fd0 /mnt`

erweitert. Das Dateisystem, welches auf das virtuelles Blockdevice zugreift, sieht keinen Unterschied zu einem "normalem" Blockdevice (Außer den Namen: `/dev/loop`). Dieses virtuelles Blockdevice kann seinerseits auf ein anderes Blockdevice oder auf eine Datei in einem bereits gemounteten Dateisystem zugreifen. Der erste Fall scheint auf den ersten Blick sinnlos zu sein. Dies ist er allerdings ganz und gar nicht, da der Loop-Treiber noch ein bisschen mehr leisteten kann, als ein Blockdevice zu emulieren. Es kann nämlich ein Offset[9] schalten und verschlüsseln. Im folgenden wird auf die zweite Erweiterung näher eingegangen.

5.1 Eigenschaften des Crypto-LOOP-Drivers

Der Crypto-Loop-Driver arbeitet mit einer Blockgröße von 512 Bytes. Soll ein Dateisystem verwendet werden, welches kein Vielfaches dieser Blockgröße verwendet, sind Performanceeinbußen zu erwarten. Das selbe gilt, wenn Offsets kein Vielfaches von 512 sind.

Auf jeden 512-Byte Block wird der Verschlüsselungsalgorithmus im CBC-Mode unabhängig angewendet. Seit dem Kernel 2.6 ist AES im Loop-Treiber offiziell integriert.

Bei der Verwendung von Journaling Dateisystemen mit Crypto-Loop ist Vorsicht geboten, da Schreibzugriffe möglicherweise nicht sofort durchgeleitet werden. Dies wird von einem Journaling Dateisystemen aber normalerweise gefordert. Es gibt allerdings die Möglichkeit, dieses Problem zu umgehen. Bei den weiter unten beschriebenen Versuchen wurde tatsächlich festgestellt, dass Daten auch nach einem `sync`[10] nicht immer sofort zurückgeschrieben wurden. Das Problem liegt hierbei in der mehrfach gepufferten Struktur, denn gepuffert wird vom Dateisystem und vom Crypto-Loop-Treiber.

Um maximale Geschwindigkeit zu erreichen, wurde der AEStandard für Intel x86 und AMD64 Prozessoren in Assembler implementiert. Diese Optimierung kann allerdings nur auf diesen beiden Prozessoren verwendet werden. Wird Linux auf einer anderen Plattform übersetzt, wird der "normale" C-Code compiliert.

[9] Der Anfang (Null Position) von `/dev/loop` ist in der verbundenen Datei (oder Blockdevice) nicht null, sondern um `n` Bytes verschoben.

[10] Befehl um das schreiben aller vom Dateisystem gepufferten Daten zu erzwingen

5.2 Installation

Wie bereits erwähnt, wurde Crypto-Loop im 2.6er Kernel offiziell aufgenommen. Dies bedeutet, dass der Loop-Treiber AES unterstützt[11]. Zusätzlich können folgende Algorithmen geladen werden: Twofish, Blowfish, Serpent, Mars, rc6, und TripleDES.

Bevor mit der Installation begonnen wird, ist es auf jeden Fall sinnvoll zu überprüfen, ob der aktuelle Kernel nicht doch schon Crypto-Loop unterstützt. (Auch bei Kernelversionen vor 2.6.) Denn viele Hersteller von Distributionen bauen Crypto-Loop selbständig in ihre Kernel ein. Zur Überprüfung kann der weiter unten beschriebene Weg gegangen werden, um ein verschlüsseltes Dateisystem zu erzeugen.

Sollte wieder Erwarten der Kernel kein Crypto-Loop unterstützen, kann folgender Weg beschritten werden. Es ist dabei Voraussetzung, dass die Kernel-Sourcen des aktuell verwendeten Kernels installiert sind. (Oder des Kernels, welcher neu installiert werden soll) Zusätzlich müssen diese richtig konfiguriert (Plattform, etc.) und übersetzt worden sein. Zuerst ist die Datei `loop-AES-latest.tar.bz2` von `http://loop-aes.sourceforge.net/` herunterzuladen und mit `tar -xvjf loop- AES-latest.tar.bz2` zu entpacken. Danach ist in das entstandene Verzeichnis zu wechseln (`cd loop-AES-v3.1c`) und `make` auszuführen. Im Normalfall erkennt make automatisch, wo die Kernel Sourcen liegen.

Der manuellen Installation ist auf jeden Fall die automatische Installation der verwendeten Distribution vorzuziehen. Unter Debian kann der Kernel zum Beispiel folgendermaßen übersetzt werden. Zuerst müssen die Kernel Quellcodes (z.B. `kernel-source-2.4.27`) installiert werden. Zusätzlich sind die Quellcodes für Cryptoloop zu installieren. (Paket `loop-aes-source` und `loop-aes-ciphers-source`. Zweites nur, wenn zusätzliche Algorithmen gewünscht werden.) Zum Schluss sind noch die Tools zum Verwenden des Crypto-Loop-Treibers erforderlich: `loop-aes-utils`. Es ist natürlich auch möglich, die allerneuesten Kernelquellen von `www.kernel.org` herunter zu laden. Dabei muss allerdings auf die Namen der Verzeichnisse geachtet werden.

Im folgenden wird davon ausgegangen, dass die `bash`[12] als Shell (Eingabeinterpreter) verwendet wurde und dass alle Befehle mit Administrator-Rechten, also z.B. unter dem User `root` auszuführt werden.

[11] Voraussetzung hierfür ist, dass die Loop-AES Unterstützung nicht bei der Übersetzung des Kernels abgeschaltet wurde.

[12] `http://www.gnu.org/software/bash/bash.html`

Des weiteren wird davon ausgegangen, dass der Kernel 2.4.27 richtig installiert und konfiguriert ist. Zusätzlich zu diesem Kernel wird CryptoLOOP installiert. Der Kernel an sich wird nicht verändert. Trotzdem ist das Vorhandensein der vollständigen Kernel Quellen erforderlich.

```
apt-get install loop-aes-source        # Pakete installieren
apt-get install loop-aes-utils
apt-get install kernel-source-2.4.27
apt-get install kernel-package
cd /usr/src/                           # Auspacken
tar -xjf loop-aes.tar.bz2
tar -xjf kernel-source-2.4.27.tar.bz2
ln -s kernel-source-2.4.27 linux       # Damit auch wirklich jedes
                                       # Programm die Sourcen findet

cp /boot/config-2.4.27-2-686 /usr/src/linux/.config  # Konfig vom
                                       # aktuell installierten
                                       # Kernel übernehmen
cd /usr/src/linux/
make-kpkg configure                    # Kernel Konfigurieren,
                                       # Übersetzen und Debian
                                       # Pakete erzeugen
cd /usr/src/modules/loop-aes           # CryptoLOOP übersetzen
debian/rules KSRC=/usr/src/linux KVERS=2.4.27-2-686 kdist_image
                                       # ... und Installieren
dpkg -i /usr/src/modules/loop-aes-2.4.27-2-686_2.1c-1_i386.deb
```

Adaptiert[13] von Wenzel (2005).

5.3 Einrichten des CryptoLoop Drivers

Der CryptoLoop Driver verwaltet in der Standardeinstellung "nur" acht virtuelle Blockdevices. Dies kann schnell zu wenig sein. Daher empfiehlt es sich, das CryptoLoop Kernel Module (`loop.o` oder `loop.ko` bei 2.6er Kerneln) mit dem Parameter `max_loop` zu laden. Somit können bis zu 256 virtuelle Blockdevices verwendet werden:

Laden des Crypto-Loop Kernel Modules mit 32 virtuellen Blockdevices.

```
insmod loop max_loop=32
```

Für jedes virtuelle Blockdevices ist ein Interface zu erstellen. Bei diesem Interface handelt es sich um eine spezielle Datei mit dem Namen `/dev/loopX`. Die ersten

[13] Zu den übernommenden Befehlen wurden Kommentare hinzugefügt.

acht sind normalerweise vorhanden. Mit dem folgenden Befehl können Interface-Dateien für 32 virtuelle Blockdevices erstellt werden:

```
# Erstellen der speziellen Dateien (Interface)
for i in 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15; do
  mknod /dev/loop$(printf %X $i) b 7 $i 2> /dev/null
done
for i in 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15; do
  mknod /dev/loop$(printf "1"%X $i) b 7 $[$i+16] 2> /dev/null
done
# Setzen der Zugriffsrechte
chown root /dev/loop*
chgrp disk /dev/loop*
chmod ug+rw,o-rwx /dev/loop*
```

Das Ergebnis dieser Operation sollte folgendes sein:

```
root@LBlacky:/dev > ls -al loop*
brw-rw----  1 root disk 7,  0 Jun 13  2005 loop0
brw-rw----  1 root disk 7,  1 Jun 13  2005 loop1
brw-rw----  1 root disk 7, 16 Jul 27  2005 loop10
brw-rw----  1 root disk 7, 17 Jul 27  2005 loop11
brw-rw----  1 root disk 7, 18 Jul 27  2005 loop12
brw-rw----  1 root disk 7, 19 Jul 27  2005 loop13
brw-rw----  1 root disk 7, 20 Jul 27  2005 loop14
brw-rw----  1 root disk 7, 21 Jul 27  2005 loop15
brw-rw----  1 root disk 7, 22 Jul 27  2005 loop16
brw-rw----  1 root disk 7, 23 Jul 27  2005 loop17
brw-rw----  1 root disk 7, 24 Jul 27  2005 loop18
brw-rw----  1 root disk 7, 25 Jul 27  2005 loop19
brw-rw----  1 root disk 7, 26 Jul 27  2005 loop1A
brw-rw----  1 root disk 7, 27 Jul 27  2005 loop1B
brw-rw----  1 root disk 7, 28 Jul 27  2005 loop1C
brw-rw----  1 root disk 7, 29 Jul 27  2005 loop1D
brw-rw----  1 root disk 7, 30 Jul 27  2005 loop1E
brw-rw----  1 root disk 7, 31 Jul 27  2005 loop1F
brw-rw----  1 root disk 7,  2 Jun 13  2005 loop2
brw-rw----  1 root disk 7,  3 Jun 13  2005 loop3
brw-rw----  1 root disk 7,  4 Jun 13  2005 loop4
brw-rw----  1 root disk 7,  5 Jun 13  2005 loop5
brw-rw----  1 root disk 7,  6 Jun 13  2005 loop6
brw-rw----  1 root disk 7,  7 Jun 13  2005 loop7
brw-rw----  1 root disk 7,  8 Feb 11  2004 loop8
brw-rw----  1 root disk 7,  9 Feb 11  2004 loop9
brw-rw----  1 root disk 7, 10 Feb 11  2004 loopA
brw-rw----  1 root disk 7, 11 Feb 11  2004 loopB
brw-rw----  1 root disk 7, 12 Feb 11  2004 loopC
brw-rw----  1 root disk 7, 13 Feb 11  2004 loopD
brw-rw----  1 root disk 7, 14 Feb 11  2004 loopE
brw-rw----  1 root disk 7, 15 Feb 11  2004 loopF
```

Bei bedarf können folgende Module nachgeladen werden, um weitere Verschlüsselungsalgorithmen neben AES zu verwenden:

```
insmod loop_twofish loop_serpent loop_blowfish
```

5.4 Einrichten eines Containers

Ein Container ist eine Datei, in welchem sich ein verschlüsseltes Dateisystem befindet.

5.4.1 Erstellen des Containers – der Imagedatei

Um solch einen Container zu erstellen, muss zuerst eine Datei (ein Image) erstellt werden. Es ist zu bedenken, dass es nicht möglich ist, die Größe des Containers nachträglich zu verändern. Um eine Größenänderung zu bewerkstelligen, ist ein neuer Container zu erstellen. Danach sind alle Daten aus dem alten in den neuen Container zu kopieren.

Der folgende Befehl erstellt die Image-Datei, welche das Dateisystem enthalten soll. Die Datei wird mit Nullen (Byte `0x00`) aufgefüllt.

```
root@LBlacky:/ > dd if=/dev/zero of=/Container.img bs=1M count=25
25+0 records in
25+0 records out
26214400 bytes transferred in 0.218848 seconds (119783549 bytes/sec)
```

`dd` liest von der Eingabedatei (Input File), welche mit `if` spezifiziert ist und schreibt die gelesenen Daten nach `of` (Output File, Ausgabedatei). `/dev/zero` ist eine spezielle Datei, welche unendlich viele Nullen (Byte `0x00`) liefert. Der Container ist hier mit `/Container.img` benannt. Die Menge der Daten (die Größe des Containers) wird errechnet, indem `bs` (Block Size) mit `count` multipliziert wird. In diesem Fall handelt es sich um 25 MByte.

```
root@LBlacky:/ > ls -alh /Container.img
-rw-------  1 root root 25M Feb  9 09:14 Container.img
```

Mit Hilfe eines Hex-Editors kann an dieser Stelle nachgewiesen werden, dass die erstellte Datei tatsächlich nur Nullen enthält. Ein `hexdump` soll den Hexeditor an dieser Stelle ersetzen.

```
root@LBlacky:/ > hexdump /Container.img
0000000 0000 0000 0000 0000 0000 0000 0000 0000
```

```
*
1900000
```

Glücklicherweise komprimiert `hexdump` die Ausgabe selbstständig. Das `*` bedeutet, dass alle folgenden Zeilen mit der Ersten identisch sind. Die erste Nummer jeder Zeile gibt die Position der aktuellen Zeile in der Datei an. Es geht mit `0000000` los und endet mit `1900000`. 1900000 ist eine Hexadezimalzahl und ist gleich 26214400 Bytes. Das sind $25 * 1024 * 1024$ Bytes, also 25 MByte.

5.4.2 Besonderheiten bei Partitionen

Die Erstellung einer verschlüsselten Partition weicht nicht weit von dieser Beschreibung ab. Statt `/Container.img` muss die Partition (z.B. `/dev/hda5` für die erste logische Partition auf der ersten Festplatte) angegeben werden. Das Erstellen der "Datei" mittels `dd` entfällt. Sonst kann die hier gegebene Beschreibung angewendet werden. Hinweis: Bei Partitionen ist mit äußerster Vorsicht zu arbeiten, denn der Inhalt der verwendeten Partition wird überschrieben. Daher den Namen des verwendeten Blockdevices immer mehrmals überprüfen.

5.4.3 Mappen des Images auf ein virtuellen Blockdevice

Der nächste Schritt ist das Verbinden der Datei mit dem virtuellen Blockdevice. Dies wird mit Hilfe von `losetup` durchgeführt:

```
root@LBlacky:/ >  losetup -e AES256 -C 5000 /dev/loop4 /Container.img
Password:
```

Der zu verwendende Verschlüsselungsalgorithmus wird mit `-e` bekanntgemacht. Bei der ersten übergebenen Datei muss es sich um das virtuelle Blockdevice handeln. Die zweite ist die Datei, die auf das virtuelle Blockdevice gemappt werden soll. Werden die Parameter `-e` und `-C` weggelassen, so enthält `/dev/loop4` exakt dieselben Daten wie `/Container.img`, nur mit dem Unterschied, dass `/dev/loop4` als Blockdevice gemountet werden kann.

In diesem Fall werden alle Daten, welche nach `/dev/loop4` geschrieben werden, mit dem eingegebenen Password[14] verschlüsselt und dann nach `/Container.img` weitergereicht. Der Lesezugriff auf `/dev/loop4` funktioniert umgekehrt. Die Daten werden von `/Container.img` gelesen, entschlüsselt und dann an den lesenden Prozess weitergereicht.

[14] Oder bessergesagt, mit einem aus dem Password errechneten Schlüssel

Das Password wird nach der Eingabe gehasht. Dies führt dazu, dass der Schlüsselraum[15] optimaler ausgenutzt wird. Denn Zahlen und Buchstaben stellen nur eine kleine Auswahl des Zeichensatzes eines Computers dar. Durch Hashing wird eine Zeichenkette erzeugt, welche alle Zeichen enthalten kann. Diese wird dann als Schlüssel für AES verwendet.

Der Parameter `-C` n versucht, es möglichen Angreifern etwas schwerer zu machen. Die Ausgabe der Hash-Funktion wird $1000 * n$ mal mit AES256[16] verschlüsselt, bevor sie als Schlüssel für die eigentliche Verschlüsselung dient. Bei Verwendung des Parameters `-C 5000` ist eine deutliche Verzögerung zwischen der Eingabe des Passwortes und der Fertigstellung des Befehls zu beobachten. Wird der Parameter weggelassen, kann keine nennenswerte Verzögerung festgestellt werden. Diese Verzögerung ist im Normalfall nicht störend. Versucht jedoch jemand die Verschlüsselung zu brechen, indem er alle Wörter eines Wörterbuches durchprobiert, so wird ihm diese Verzögerung zum Verhängnis. Denn sie tritt bei jedem Versuch auf und lässt die Zeit zum Durchprobierten etwa um den Faktor $1000 * n$ steigen, was einen Angriff erheblich erschwert.

Der folgende Befehl gibt den aktuellen Status eines virtuellen Blockdevices aus. Es ist zu erkennen, dass AES zur Verschlüsselung verwendet wird und dass die gemappte Datei `/Container.img` ist.

```
root@LBlacky:/ > losetup /dev/loop4
/dev/loop4: [0306]:1819 (/Container.img) encryption=AES0
```

Eine kleine Schleife kann auch hier eingesetzt werden, um alle Devices abzufragen.

```
for i in /dev/loop*; do
  losetup $i
done
```

5.4.4 Erstellen eines Dateisystems im Container

Der nächste Schritt ist das Erstellen eines Dateisystems in dem Container. In diesem Fall wird ein ext2-Dateisystem erstellt.

```
root@LBlacky:/ > mke2fs /dev/loop4
mke2fs 1.37 (21-Mar-2005)
Filesystem label=
OS type: Linux
Block size=1024 (log=0)
Fragment size=1024 (log=0)
```

[15] Menge alle möglicher Schlüssel.
[16] AES Schlüssellänge: 256 Bit, aber AES Blockgröße 128Bit

```
6400 inodes, 25600 blocks
1280 blocks (5.00%) reserved for the super user
First data block=1
4 block groups
8192 blocks per group, 8192 fragments per group
1600 inodes per group
Superblock backups stored on blocks:
        8193, 24577

Writing inode tables: done
Writing superblocks and filesystem accounting information: done

This filesystem will be automatically checked every 27 mounts or
180 days, whichever comes first.  Use tune2fs -c or -i to override.
```

Es ist sinnvoll, das so eben erstellte Dateisystem noch etwas feiner abzustimmen. Normalerweise werden 5% des Speicherplatzes auf einem Dateisystem für den User root reserviert, um sicherzustellen, dass alle Systemdienste genug Speicherplatz haben. Hier wird davon ausgegangen, dass keine Systemdienste im Container laufen. Daher werden mit `-m0` 0% reserviert. Ext2 Dateisysteme werden nach eine einstellbaren Anzahl von Tagen oder von mount-Operationen vollständig getestet. Dies kann nicht abgeschaltet werden, aber es ist möglich, die größten möglichen Werte einzustellen. Test einmal im Jahr (12Monate) `-i 12m` und aller 9999 mounts `-c 9999`

```
root@LBlacky:/ > tune2fs -i 12m -m0 -c 9999 /dev/loop4
tune2fs 1.37 (21-Mar-2005)
Setting maximal mount count to 9999
Setting interval between check 31104000 seconds
Setting reserved blocks percentage to 0 (0 blocks)
```

5.4.5 Mounten, testen und umounten des erstellten Dateisystems

Nun ist das Dateisystem erstellt und kann gemountet werden. Dazu wird vorher der mount-point[17] erstellt.

```
mkdir /crypto
mount /dev/loop4 /crypto
```

Der Container kann jetzt verwendet werden. In diesem Fall wurde repräsentativ eine Datei in den Container kopiert:

```
cp /usr/src/modules/loop-aes/README /crypto/
```

[17] Verzeichnis, in welchem das neue Dateisystem sichtbar werden soll

Der Beginn dieser Datei wurde dann im Dateisystem gesucht und ausgegeben:

```
root@LBlacky:/ > hexdump -C -s 223232 -n 256 /dev/loop4
00036800  57 72 69 74 74 65 6e 20  62 79 20 4a 61 72 69 20  |Written by Jari |
00036810  52 75 75 73 75 20 3c 6a  61 72 69 72 75 75 73 75  |Ruusu <jariruusu|
00036820  40 75 73 65 72 73 2e 73  6f 75 72 63 65 66 6f 72  |@users.sourcefor|
00036830  67 65 2e 6e 65 74 3e 2c  20 4f 63 74 6f 62 65 72  |ge.net>, October|
00036840  20 32 36 20 32 30 30 34  0a 0a 43 6f 70 79 72 69  | 26 2004..Copyri|
00036850  67 68 74 20 32 30 30 31  2c 32 30 30 32 2c 32 30  |ght 2001,2002,20|
00036860  30 33 2c 32 30 30 34 20  62 79 20 4a 61 72 69 20  |03,2004 by Jari |
00036870  52 75 75 73 75 2e 0a 52  65 64 69 73 74 72 69 62  |Ruusu..Redistrib|
00036880  75 74 69 6f 6e 20 6f 66  20 74 68 69 73 20 66 69  |ution of this fi|
00036890  6c 65 20 69 73 20 70 65  72 6d 69 74 74 65 64 20  |le is permitted |
000368a0  75 6e 64 65 72 20 74 68  65 20 47 4e 55 20 50 75  |under the GNU Pu|
000368b0  62 6c 69 63 20 4c 69 63  65 6e 73 65 2e 0a 0a 0a  |blic License....|
000368c0  54 61 62 6c 65 20 6f 66  20 43 6f 6e 74 65 6e 74  |Table of Content|
000368d0  73 0a 7e 7e 7e 7e 7e 7e  7e 7e 7e 7e 7e 7e 7e 7e  |s.~~~~~~~~~~~~~~|
000368e0  7e 7e 7e 0a 31 2e 20 20  20 20 4c 6f 6f 70 20 64  |~~~.1.....Loop d|
000368f0  65 76 69 63 65 20 70 72  69 6d 65 72 0a 32 2e 20  |evice primer.2. |
```

Danach wurde die gleiche Stelle in der gemappten Datei untersucht

```
root@LBlacky:/ > hexdump -C -s 223232 -n 256 /Container.img
00036800  88 17 c4 78 15 18 1b 6a  68 6d 84 90 c7 01 5f d6  |...x...jhmÇ...Ö_|
00036810  2a 6c 5c 76 a9 83 ea a1  66 95 4e 87 f6 94 1a 42  |*l\ⓒvê¡.f.Nö...B|
00036820  50 27 98 77 01 da ea cf  f3 02 ec dc df d1 aa 56  |P'.wÚêÏóìÜßÑª..V|
00036830  8e 2c 9c 5a 9f 5b 88 b8  ad aa e0 d4 d0 22 e5 ed  |.,.Z¸àÔ■åí.[..."|
00036840  4a 6f 94 36 d7 fe 59 63  8f 90 1b 09 4f 71 cd 86  |Jo.6..Yc....ÍOq.|
00036850  a9 1d 8d e1 d0 7a ad eb  70 27 ae c4 9e 17 47 4f  |.....z..p'®Ä'..GO|
00036860  ac 90 04 cd bc 7f d6 27  91 8e 90 a9 52 d8 4e 1b  |.......'....ØRN.|
00036870  bb b6 05 6e f6 85 15 65  6e ed 74 48 91 ea 86 ef  |...ön..ínentHêï..|
00036880  d7 19 f4 59 c1 91 a3 18  9c 68 da f2 9b 94 a7 f5  |...ÁY■...Úòh■õ..|
00036890  ad 61 45 16 c4 16 60 d6  72 54 30 c7 d7 bf 38 60  |aEÄÖ..'ÇrT0¿.8'|
000368a0  30 e9 d9 1d 7c db fd 5c  46 5f 63 1a b2 f6 2e a9  |0...|..\F_cö.ⓒ...|
000368b0  ae 49 f6 1c c1 27 73 03  9d ac 49 c4 36 04 12 fb  |.öIÁ.'s...ÄI6û..|
000368c0  9e e1 e9 cc 38 ac 2d 8f  73 ec db 8f ab 77 0c 12  |....8...ìÛs■.w..|
000368d0  67 71 92 9c f5 10 49 7c  f2 03 eb 9c 95 74 62 38  |gqõ...Iòë|...tb8|
000368e0  bd e9 2d f3 80 37 2f 05  48 06 2b e4 dd 28 80 cb  |..-..7/.HäÝË.+(.|
000368f0  b6 62 43 b0 fe 06 8f da  95 63 f5 ad 05 41 92 1f  |.°■bCÚ...õc..A..|
00036900
```

Es ist zu erkennen, dass die kopierte Datei als Klartext in das Dateisystem, also auf das virtuellem Blockdevice geschrieben wurde. Aufgrund der eingeschalteten Verschlüsselung ist an derselben Stelle in der gemappten Datei nur "Datenschrott" vorzufinden.

Nach Benutzung eines Containers ist es wichtig, diesen wieder zu umounten, da sonst jeder auf die enthaltenen Daten zugreifen kann. Dabei ist es sehr wichtig daran zu denken, das virtuelle Blockdevice wieder freizugeben. Ist dies nicht ge-

schehen, kann ein Angreifer durch Ausführen eines entsprechenden mount-Befehls auf die Daten **ohne Eingabe des Passwortes** zugreifen.

```
root@LBlacky:/ > umount /crypto
root@LBlacky:/ > losetup -d /dev/loop4
root@LBlacky:/ > losetup /dev/loop4
loop: can't get info on device /dev/loop4: No such device or address
```

Der letzte Befehl fragt den Status des virtuellen Blockdevices ab. Dies sollte nach der Freigabe in einer Fehlermeldung resultieren.

5.5 Mounten und umounten mit je einem Befehl

Das alles geht auch einfacher. Ist der Container erstellt, so kann er direkt mit `mount` gemountet werden. In diesem Fall kümmert sich mount automatisch um die Vergabe und Freigabe des virtuellen Blockdevices.

```
root@LBlacky:/ > \
mount -o encryption=AES256,itercountk=5000 /Container.img /crypto
root@LBlacky:/ > diff /usr/src/modules/loop-aes/README /crypto/README
root@LBlacky:/ > for i in /dev/loop*; do losetup $i; done
/dev/loop0: [0306]:1819 (/Container.img) encryption=AES0
loop: can't get info on device /dev/loop1: No such device or address
[...]
root@LBlacky:/ > umount /crypto
root@LBlacky:/ > for i in /dev/loop*; do losetup $i; done
loop: can't get info on device /dev/loop0: No such device or address
loop: can't get info on device /dev/loop1: No such device or address
[...]
```

Natürlich ist es an dieser Stelle sehr zu empfehlen, das Passwort zu erinnern. `itercountk` definiert das gleiche wie bei `losetup` der Parameter `-C` und ist selbstverständlich identisch zu wählen. `diff` vergleicht die beiden übergebenden Dateien. Keine Ausgabe bedeutet, dass beide Dateien identisch sind. Nach diesem Vergleich werden die Zustände alle virtuellen Blockdevices ausgegeben. Mount wählte automatisch das erste freie, hier `/dev/loop0`. Das Freigeben des Containers geschieht mit `umount /crypto`. Wie zu erkennen ist, wurde das verwendete virtuelle Blockdevice ebenfalls freigegeben.

5.6 Swap-Partition und `/tmp`

Bei dem Verschlüsseln von Daten wird leicht vergessen, dass die verschlüsselten Daten für die Benutzung entschlüsselt werden. Wer eine geheime Datei von einer verschlüsselten Partition in den Editor lädt, kopiert die geheime Datei unverschlüsselt in den Hauptspeicher des Computers. Es ist daher darauf zu achten, dass keine Programme laufen, die den Hauptspeicher auslesen könnten. Des weiteren besteht die Gefahr, dass ein Teil des Hauptspeichers in die Swap-Partition ausgelagert wird. Nun ist ein Teil der geheimen Datei wieder **unverschlüsselt** auf der Festplatte. Dies kann vermieden werden, indem entweder keine Swap-Partition verwendet wird oder diese ebenfalls verschlüsselt wird. Die Verschlüsselung führt an dieser Stelle natürlich wieder zu Performanceverlust. Das gleiche Problem tritt bei dem Verzeichnis für temporäre Dateien (`/tmp`) auf. Viele Programme speichern dort Dateien zwischen, was dazu führen kann, dass geheime Dateien aus dem verschlüsselten Container unverschlüsselt auf der Festplatte landen.

Swap-Partition verschlüsseln:

Die folgende Zeile in der Datei `/etc/fstab` muss erweitert werden:

```
/dev/hdXX none swap sw 0 0
```

Erweiterte Zeile:

```
/dev/hdXX none swap sw,loop=/dev/loopX,encryption=AES128 0 0
```

Es ist ratsam, die Swap-Partition mit einem Hexeditor zu untersuchen. So kann schnell festgestellt werden, ob diese auch tatsächlich verschlüsselt ist.

5.7 An was noch zu denken ist

Die Sicherheit der verschlüsselten Daten ist immer (nur) so gut wie das Passwort. Ein zu einfaches Passwort kann schnell erraten werden. Ein vergessenes Passwort schützt die Daten sehr effektiv vor dem Eigentümer.

Wer Daten verschlüsselt, sollte sich darüber im klaren sein, dass er womöglich nicht mehr auf seine Daten zugreifen kann, wenn etwas schief gegangen ist. Es sollte daher selbstverständlich sein, immer mindestens zwei Wege zu haben, um seine Daten zu entschlüsseln. Ein einfacher und sehr guter “zweiter Weg” ist die Linux-

Live-CD Knoppix[18]. Von dieser CD kann ein vollständiges Linux mit Crypto-Loop-Unterstützung gebootet werden. Es ist sehr zu empfehlen, den Knoppix-Weg **vor** einem Systemcrash auszuprobieren. Denn dann ist sofort erkennbar, wenn etwas nicht klappt. (Z.B. Passwort, Algorithmus oder `itercountk`-Wert falsch). Das Notieren dieser Werte an einem sicheren Ort ist ebenfalls empfehlenswert. Denn nach einem halben Jahr ist der Algorithmus oder `itercountk`-Wert in Vergessenheit geraten, was zu Problemen führen kann.

Backups zu erstellen ist auch immer eine gute Idee. Wenn überhaupt, sollten Backups erst dann verschlüsselt werden, wenn man schon einige Erfahrung mit Verschlüsselung hat (und wenn der zweite Weg ausprobiert wurde). Auf jeden Fall sollte vor der Erstanwendung einer Verschlüsselung der Datenbestand gesichert werden!

5.8 Geschwindigkeitstest

Um einen Überblick über die Performance von AES bzw. von Crypto-Loop zu erhalten, wurde ein Performancevergleich durchgeführt. Dieser Test wurde auf einem Pentium III 850 MHz Computer einmal mit und einmal ohne Verschlüsselung durchgeführt. Es wurden jeweils 250MByte von der Festplatte gelesen.

Die Transferrate wurde mit Hilfe von `dd` gemessen. Ausgabedatei (`of`) war `/dev/null`. Alles was auf dieses spezielle Device geschrieben wird, hört schlicht und einfach auf zu existieren. Es wird nicht weiter verarbeitet.

“Normales” lesen von Festplatte:

```
root@LBlacky:/ > dd if=/dev/hda7 of=/dev/null bc=1M count=250
250+0 records in
250+0 records out
262144000 bytes transferred in 14.469067 seconds (18117547 bytes/sec)
```

Transferrate: ca. 17MByte/s. CPU-Last ca. 27%

Lesen eines verschlüsseltem Bereiches von Festplatte (inklusive entschlüsseln).

```
root@LBlacky:/ > dd if=/dev/loopE of=/dev/null bs=1M count=250
250+0 records in
250+0 records out
262144000 bytes transferred in 21.277416 seconds (12320293 bytes/sec)
```

Transferrate: ca. 12MByte/s. CPU-Last ca. 64%

[18] http://www.knoppix.org/

Es ist zu erkennen, dass die Entschlüsselung den Datentransfer um gut 35% verlangsamt und fast zweieinhalb mal soviel Rechenlast (CPU-Last) erzeugt.

11MByte/s entspricht 24063 512_Byte_Blöcken/s

Wird nun davon ausgegangen, dass statt der Entschlüsslung jedes 512_Byte_Blockes, immer der selbe Block mit einem anderen Password entschlüsselt wird, würde das Durchprobieren aller möglichen Passwörter ca. $1.5 * 10^{66}$ Jahre dauern. In dieser Rechnung ist weder das Analysieren des gewonnenen Klartextblockes (z.B. vergleichen mit einem bekannten Klartextblock) noch das Initialisieren des Verschlüsselungsalgorithmus inbegriffen. Diese Faktoren würden die Zeit noch verlängern.

$$\frac{2^{256} Moeglichkeiten}{22411 \frac{256\_Byte\_Bloecke}{s}} * \frac{1}{60*60*24*7*53} \approx 1.5 * 10^{66} Jahre$$

Wird davon ausgegangen, dass das Password aus nur einem Wort der englischen Sprache besteht, welches im Oxford Dictionary steht, so beschränkt sich die Anzahl der möglichen Passwörter auf 414800[19]. Das Durchprobieren aller dieser Möglichkeiten würde mit dem genannten Rechner gerade mal ca. 17 Sekunden dauern.

$$\frac{414800 Moeglichkeiten}{22411 \frac{256\_Byte\_Bloecke}{s}} \approx 17s$$

Es sind bei dem Crypto-Loop-Driver allerdings normalerweise nur Passwörter mit 20 oder mehr Zeichen erlaubt. Wird nun angenommen, dass das Password aus fünf Wörtern aus dem englischen Wörterbuch besteht, so ergeben sich $414800^5 \approx 1.2 * 10^{28}$ Möglichkeiten. Unter der Annahme, dass die Wörter rein zufällig zusammengesetzt wurden (also keinen Satz bzw. Sinn ergeben müssen) folgt daraus, dass es etwa $1.6 * 10^{16}$ Jahre dauern würde, alle Möglichkeiten durchzuprobieren.

$$\frac{1.2*10^{28} Moeglichkeiten}{22411 \frac{256\_Byte\_Bloecke}{s}} * \frac{1}{60*60*24*7*53} \approx 1.6 * 10^{16} Jahre$$

Im statistischen Mittel wird es notwendig sein, die Hälfte alle Passwörter durchzuprobieren, um das richtige zu finden. Das bedeutet, dass im statistischen Mittel nur die Hälfte der errechneten Zeit notwendig ist, um das Passwort zu finden.

Durch das Aktivieren der weiter oben genannten Option `-C` bei losetup bzw. `itercountk` bei mount kann diese Form der Attacken erheblich erschwert werden. Denn das Initialisieren des Verschlüsselungsverfahrens wird erheblich aufwendiger und langwieriger durch Einsatz dieses Parameters.

[19] Quelle: http://www.oed.com/about/facts.html

6 Bibliographie und Referenzen

Corbet et al (2004) A weak cryptoloop implementation in Linux? [Online] Available at `http://lwn.net/Articles/67216/` (accessed 22. December)

Daemen, J.; Rijmen, V. (2002) The Design of Rijndael. Berlin Heidelberg: Springer-Verlag. ISBN 3-540-42580-2.

Deters, D. (2002) Systematisierung der symmetrischen kryptographischen Verfahren [Online] Available at `http://www.informatik.hu-berlin.de/Forschung_Lehre/algorithmenII/Lehre/SS2002/Ana_krypt_Alg/02Systematisierung/Systematisierung.html` (accessed 20. December 2005)

Hölzer, R. (2004) Cryptoloop HOWTO. [Online] Available at `http://www.tldp.org/HOWTO/Cryptoloop-HOWTO/` (accessed 22. December 2005)

Ruusu, J (2004) "`README`" von Loop-AES. [Online] Available at `http://loop-aes.sourceforge.net/loop-AES.README`

van Rees, J (2005) Introduction to Cryptography and Cryptosystems: Assignments [Online] Available at `http://www.cs.umanitoba.ca/~cs414/assignments/(test2output.txt)` (accessed 26. December 2005)

Wenzel, R. (2005) Verschluesselte Dateisysteme und Container. [Online] Available at `http://www.ruwela.de/Linux-Befehls-Beispiele/node16.html` (accessed 22. December 2005)

Wilkin, C (2004) Der Algorithmus des "Advanced Encryption Standard". [Online] Available at `http://www.ainformatik.fh-trier.de/~scheerhorn/` (accessed 20. December 2005)