Universität Paderborn
Fachbereich 17, Mathematik-Informatik

Sicherheit und Anwendungen des Advanced Encryption Standards (AES) Rijndael

Diplomarbeit

Volker Krummel
Richterstr. 10
33100 Paderborn

vorgelegt bei
Prof. Dr. Johannes Blömer
und
Prof. Dr. Joachim von zur Gathen

Paderborn, den 2. November 2001

Vorwort

Zunächst möchte ich mich herzlich bei Prof. Dr. Blömer und Alexander May bedanken, die wesentlich zur Themenfindung der Diplomarbeit beigetragen haben. Dankbar bin ich auch für die intensive Betreuung und die vielen hilfreichen Diskussionen und Ratschläge.
Alexander May möchte ich auch besonders für die konstruktive Kritik sowie für das Korrekturlesen beim Anfertigen der Arbeit danken.
Besonderer Dank gilt meiner Frau Tina für ihre Rücksicht und ihr Verständnis vor allem in den letzten Wochen und meinem Sohn Timmy, der mir jeden Tag vor Augen führte, daß es immer noch etwas Neues zu entdecken gibt.

Erklärung
Ich versichere, daß ich diese Diplomarbeit selbständig verfaßt und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet sowie Zitate kenntlich gemacht habe.

Inhaltsverzeichnis

1. Einleitung

Am 2.Oktober 2000 traf das National Institute of Standards and Technology (NIST) der USA die Entscheidung über den neuen Verschlüsselungsstandard "Advanced Encryption Standard" (AES). Die Wahl fiel auf Rijndael, entwickelt von den beiden Belgiern Vincent Rijmen und Joan Daemen.

Vorausgegangen war ein fast 4-jähriger Auswahlprozeß, bei dem Verschlüsselungssysteme aus der ganzen Welt gegeneinander antraten. Am 2.Januar 1997 hatte das NIST offiziell bekanntgegeben, einen neuen Verschlüsselungsstandard für "unclassified" Dokumente zu entwickeln. AES sollte den bisher verwendeten Standard, den "Data Encryption Standard" (DES) ablösen, da dieser aufgrund seiner kurzen Schlüssellänge nicht mehr als sicher eingestuft wurde. Die Electronic Frontier Foundation (EFF) veröffentlichte 1997 Pläne zum Bau einer speziellen Hardware, dem "DES-Cracker", der eine vollständige Suche nach dem richtigen Schlüssel in weniger als 3 Tagen durchführen konnte. Die Kosten dieser Maschine beliefen sich auf ca. 250.000 \$. Von da an galt DES als vollständig gebrochen.

Am 12.September 1997 erging dann der "Call for Algorithms" für AES, so daß jeder seinen Algorithmus vorschlagen konnte. Der Algorithmus sollte vollständig offengelegt werden und frei von Lizenzansprüchen sein. Gesucht wurde eine symmetrische Blockchiffre mit 128 Bit Blocklänge und möglichen Schlüssellängen von 128, 192 und 256 Bit.

Zur ersten AES Konferenz am 20.August 1998 hatte das NIST 15 Kandidaten von privaten Firmen und öffentlichen Institutionen aus 12 Ländern registriert (vgl. Tabelle 1.1).

Der weitere Auswahlprozeß war so angelegt, daß Kommentare und Untersuchungen der Öffentlichkeit ausdrücklich erwünscht waren. Dies war bei der Wahl des Vorgängers, dem DES, anders. Damals hatte man die Wahl des Algorithmus hinter verschlossenen Türen durchgeführt und nachträglich sogar Änderungen an dem gewählten Algorithmus durchgeführt. Aufgrund dieses "undurchsichtigen" Verhaltens konnte nie der Verdacht ganz ausgeräumt werden, daß die amerikanische National Security Agency (NSA) sogenannte Hintertüren in das Verfahren eingebaut hat, die es (nur ihr) ermöglichen sollten, verschlüsselte Nachrichten zu entziffern. Durch die intensive Einbeziehung der Öffentlichkeit wollte man diesen Verdacht von vorneherein ausschließen. Nach ungefähr einem halben Jahr wurden im März 1999 die

CAST-256	Entrust	**Magenta**	Dt.Telekom
Crypton	Future Systems	**Mars**	IBM
DEAL	Knudsen	**RC6**	RSA
DFC	ENS-CNRS	**Rijndael**	Daemen,Rijmen
E2	NTT	**SAFER+**	Cylink
Frog	TecApro	**Serpent**	Anderson, Biham, Knudsen
HPC	Schroeppel		
LOKI97	Brown et al.	**Twofish**	Counterpane

Tabelle 1.1.: Die 15 Kandidaten der ersten Runde

eingegangenen Analysen und Kommentare zu den 15 Kandidaten auf der zweiten AES Konferenz diskutiert und ein weiteres halbes Jahr später, im August 1999, gab das NIST die Auswahl der 5 Kandidaten bekannt, die es in die zweite Runde geschafft hatten. Die Kriterien für die Auswahl waren

1. Sicherheit
2. Effizienz
3. Flexibilität

Sicherheitsprobleme wurden bei den Verfahren DEAL, Frog, HPC, LOKI97 und Magenta gefunden, so daß sie nicht mehr als Standard in Frage kamen. Bei der Effizienz konnten DEAL, Frog, Magenta und SAFER+ nicht mit den restlichen Verfahren mithalten. Da ein Standard aber in allen erdenklichen Situationen Verwendung finden soll, ist die Effizienz entscheidend für seinen Einsatz. Crypton schaffte es nicht in die zweite Runde, weil seine Struktur zu ähnlich zur Rijndael-Struktur war. Die Anzahl der Kandidaten, die die zweite Runde erreichen sollten, war auf max. 5 festgelegt worden. Deswegen wurden CAST-256, DFC und E2 "ausgesiebt", obwohl keinerlei Probleme bei ihnen gefunden wurden. Übrig blieben nur Mars, RC6, Rijndael, Serpent und Twofish.

Diese 5 übrig gebliebenen Kandidaten wurden wieder zur öffentlichen Diskussion gestellt. Auf der dritten AES Konferenz im April 2000 wurden die Beiträge analysiert und bewertet und am 2.Oktober 2000 fiel dann die Entscheidung zugunsten von Rijndael.

Bezüglich der Sicherheit hatte das NIST bei keinem der 5 Kandidaten Bedenken, entscheidend für die Wahl waren also die Geschwindigkeit und die Flexibilität des Algorithmus. Hier konnte Rijndael die meisten Pluspunkte für sich verbuchen. Sein elegantes Design, daß auf streng mathematischen Grundlagen beruht, erlaubt einfache und effiziente Implementierungen sowohl auf High-end Prozessoren als auch auf Smartcards.

In dieser Arbeit untersuchen wir Sicherheitsaspekte von Rijndael. Zunächst betrachten wir die Sicherheit beim Verschlüsseln. Hier untersuchen wir den bisher stärksten Angriff, den sogenannten SQUARE-Angriff und liefern eine Verbesserung um ca. Faktor 2. Im Anschluß daran betrachten wir Rijndael als Grundbaustein für Hashfunktionen und untersuchen 2 der gängigsten Schemata. Entscheidend für die Qualität der Hashfunktion ist, daß die Ausgabe der Chiffre pseudozufällig ist. Wir beschreiben zwei Unterscheider, die diese Anforderung für 3 bzw. 4 Runden Rijndael widerlegen.

Diese Diplomarbeit gliedert sich in folgende Kapitel:

wir das Zusammenspiel von einer bzw. zwei Runden Rijndael mit diesen Konstruktionen. Entscheidend für die Qualität der Hashfunktion ist, daß die Ausgaben der Chiffre pseudozufällig sind. Zum Ende der Arbeit geben wir zwei sogenannte Unterscheider an, die die Ausgaben von 3 bzw. 4 Runden Rijndael von "echtem Zufall" unterscheiden können. Der erste Unterscheider ist in der Lage, 3 Runden Rijndael von der zufälligen Gleichverteilung unterscheiden zu können. Als zweites analysieren wir einen Unterscheider von Gilbert und Minier, der bis zu 4 Runden Rijndael von der Gleichverteilung unterscheiden kann. Die Verwendung von bis zu 4 Runden Rijndael als Grundbaustein für Hashfunktionen ist also nicht zu empfehlen.

2. Mathematische Grundlagen

« Darum, Studenten, studiert Mathematik und baut nicht ohne Fundamente! »

Leonardo da Vinci [1]

In diesem Kapitel werden die nötigen mathematischen Grundlagen beschrieben, die wir in den folgenden Kapiteln verwenden. Dabei beschränken wir uns auf die für das Verständnis nötigen Spezialfälle.

2.1. Der Körper $\mathbb{F}_{2^8}$

Zu jeder Primpotenz q mit $q = p^n$, p prim, existiert genau ein endlicher Körper $\mathbb{F}_q$ mit q Elementen (siehe z.B. [LN86]). Es gibt zwar unterschiedliche Möglichkeiten, die Elemente dieses Körpers darzustellen, aber alle Darstellungen sind isomorph.

Für die Darstellung der Addition und Multiplikation, betrachten wir die Elemente von $\mathbb{F}_{2^8}$ in der klassischen Polynomdarstellung modulo eines irreduziblen Polynoms m vom Grad 8. Ein Polynom heißt irreduzibel, falls es nur durch konstante Polynome und durch sich selbst teilbar ist. Der Grad $\deg f$ eines Polynoms $f \neq 0$ ist das größte $n \in \mathbb{N}$ mit $f_n \neq 0$.

Ein Element $a \in \mathbb{F}_{2^8}$ wird dargestellt in Form von Bits $a_7 \dots a_0$, die wir als Koeffizienten des Polynoms

$$a = a_7x^7 + a_6x^6 + a_5x^5 + a_4x^4 + a_3x^3 + a_2x^2 + a_1x + a_0 \in \mathbb{F}_2[x]/\langle m \rangle = \mathbb{F}_{2^8}$$

auffassen. Wir werden die Elemente der Einfachheit halber aber auch als Binärzahl z.B. 01010111_2, als Dezimalzahl z.B. 87_{10} und als Hexadezimalzahl z.B. 57_{16} darstellen. Um ein Element von der Dezimaldarstellung bzw. der Hexadezimaldarstellung in die Polynomdarstellung zu konvertieren, bilden wir zunächst die Binärdarstellung und fassen die einzelnen Bits als Koeffizienten auf.

Über $\mathbb{F}_{2^8}$ sind die beiden Operationen Addition und Multiplikation definiert, die wir kurz vorstellen wollen.

[1] Codex Quaderni I, Folio 17 r, zitiert nach Theodor Lücke "Leonardo da Vinci - Tagebücher und Aufzeichnungen", 1940, Paul List Verlag Leipzig

2.1.1. Die Addition über $\mathbb{F}_{2^8}$

Zwei Elemente $a, b \in \mathbb{F}_{2^8}$ werden addiert, indem man in der Polynomdarstellung die Koeffizienten modulo 2 addiert (**XOR**).

$$a + b = \sum_{i=0}^{7} (a_i \oplus b_i) \cdot x^i$$

Eine Addition entspricht also einer bitweisen Verknüpfung der beiden Koeffizientenvektoren per **XOR** .

Beispiel 2.1.1 *Sei* $a = 57_{16} = 87_{10} = 01010111_2$ *bzw. in Polynomdarstellung* $x^6 + x^4 + x^2 + x + 1$ *und* $b = 83_{16} = 131_{10} = 10000011_2$ *bzw.* $x^7 + x + 1$. *In der Polynomdarstellung bilden wir die Summe*

$$a + b = (x^6 + x^4 + x^2 + x + 1) + (x^7 + x + 1) = x^7 + x^6 + x^4 + x^2 = c$$

c *entpsricht dabei* $11010100_2 = 212_{10} = D4_{16}$.

2.1.2. Die Multiplikation über $\mathbb{F}_2[x]/\langle x^8 + x^4 + x^3 + x + 1\rangle$

Die Multiplikation von a, b über $\mathbb{F}_{2^8}$ entspricht der Multiplikation der beiden Polynome modulo eines irreduziblen Polynoms $m \in \mathbb{F}_2[x]$ vom Grad 8.

Ein Polynom heißt irreduzibel, wenn es als Faktoren nur konstante Polynome und sich selbst hat. In diesem Fall hat jedes von Null verschiedene Element modulo m ein multiplikatives Inverses, das wir mit dem erweiterten Euklidischen Algorithmus (EEA) berechnen können. (Vergleiche hierzu z.B. [vG99].)

Wir wählen hier als Modul das irreduzible Polynom

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

Also ist $a \cdot b \mod m$ das Produkt von $a, b \in \mathbb{F}_{2^8}$.

Beispiel 2.1.2 *Seien* $a = 57_{16}$ *und* $b = 83_{16}$ *und wir wollen* $a \cdot b = c \in \mathbb{F}_{2^8}$ *berechnen. Die Binärdarstellung von* $a = 01010111_2$ *liefert das Polynom* $x^6 + x^4 + x^2 + x + 1$ *und die Binärdarstellung von* $b = 10000011_2$ *liefert das Polynom* $x^7 + x + 1$. *Wir berechnen nun*

$$\begin{aligned} & (x^6 + x^4 + x^2 + x + 1) \cdot (x^7 + x + 1) \\ \equiv\ & x^{13} + x^{11} + x^9 + x^8 + x^6 + x^5 + x^4 + x^3 + 1 \\ \equiv\ & x^7 + x^6 + 1 \pmod{x^8 + x^4 + x^3 + x + 1} \end{aligned}$$

In der Binärdarstellung entspricht dies 11000001_2 *bzw. in der Hexadezimaldarstellung* $C1_{16}$

Zur Berechnung des Inversen a^{-1} eines Elements $a \in \mathbb{F}_2[x]/\langle m \rangle$ verwenden wir nun wie oben erwähnt, den erweiterten Euklidischen Algorithmus. Da m irreduzibel ist, gilt $ggT(a,m) = 1$ und der EEA berechnet zu den beiden Eingabewerten a, m zwei Elemente $s, t \in \mathbb{F}_{2^8}$ mit

$$a \cdot s + m \cdot t = \text{ggT }(a,m) = 1.$$

Reduzieren wir diese Gleichung modulo m, so verschwindet der Term $m \cdot t$ und anhand der resultierenden Kongruenz $a \cdot s \equiv 1 \mod m$ sieht man, daß s das Inverse zu a in $\mathbb{F}_2[x]/\langle m \rangle$ ist.

2.1.3. Die Multiplikation über $\mathbb{F}_2[x]/\langle x^8 + 1 \rangle$

Im folgenden wollen wir noch die Multiplikation in $\mathbb{F}_2[x]$ modulo eines nicht irreduziblen Polynoms

$$m = x^8 + 1$$

betrachten. Wir operieren also nicht mehr über einem Körper, sondern $R = \mathbb{F}_2[x]/\langle x^8 + 1 \rangle$ ist nur ein Ring, in dem nicht jedes von 0 verschiedene Element ein multiplikatives Inverses besitzt. Um die eindeutige Rückrechnung zu gewährleisten, multiplizieren wir mit einem festen Polynom

$$b = x^4 + x^3 + x^2 + x + 1.$$

Daß b ein multiplikatives Inverses besitzt, kann man sich leicht überlegen:
Über $\mathbb{F}_2[x]$ zerfällt $x^8 + 1$ in die Faktoren $(x+1)^8$, d.h. 1 ist die einzige Nullstelle[2]. b hat aber nur Nullstellen $\neq 1$, so daß b und m teilerfremd sind und somit b modulo m invertierbar ist. Das Inverse zu b lautet

$$b^{-1} = x^6 + x^3 + x \in \mathbb{F}_2[x]/\langle x^8 + 1 \rangle$$

Eine Multiplikation zweier beliebiger Polynome vom Grad ≤ 7 ergibt ein Polynom c vom Grad ≤ 14. Die Reduktion modulo $x^8 + 1$ ist nun einfach durch Reduktion der Exponenten modulo 8 möglich, denn für alle $i \geq 0$ gilt:

$$x^i \equiv x^{(i \mod 8)} \mod x^8 + 1$$

Wenn wir c in einen unteren Teil $c_0 \in \mathbb{F}_2[x]$ und einen oberen Teil $c_1 \in \mathbb{F}_2[x]$ zerlegen, jeweils mit $\deg c_0, \deg c_1 \leq 7$ und

$$c = c_1 \cdot x^8 + c_0,$$

so können wir die Reduktion modulo $x^8 + 1$ durch Addition von $c_0 + c_1$ über $\mathbb{F}_{2^8}$ berechnen.

[2]Es gilt der "Freshman's Dream": $x^8 + 1^8 = (x+1)^8$

Beispiel 2.1.3 *Als Beispiel multiplizieren wir* $a = x^9+x^7+x^5+x^4+x^3+x^2+x+1$ *und* $b = x^4+x^3+x^2+x+1$ *modulo* x^8+1.

$$\begin{aligned}(x^9+x^7+x^5+x^4+x^3+x^2+x+1)\cdot(x^4+x^3+x^2+x+1) &= \\ x^{13}+x^{12}+x^9+x^8+x^5+x^4+x^2+1 &= c\end{aligned}$$

Wir zerlegen nun c *in den oberen Teil*

$$x^{13}+x^{12}+x^9+x^8 = \underbrace{(x^5+x^4+x+1)}_{=:c_1}\cdot x^8$$

und den unteren Teil

$$c_0 = x^5+x^4+x^2+1$$

und addieren $c_0 + c_1$ *über* $\mathbb{F}_2[x]$*:*

$$\begin{array}{rccccc} & x^5 & +x^4 & & +x & +1 \\ \oplus\ (& x^5 & +x^4 & +x^2 & & +1\ ) \\ \hline & & & x^2 & +x & \end{array}$$

Also gilt

$$(x^9+x^7+x^5+x^4+x^3+x^2+x+1)\cdot(x^4+x^3+x^2+x+1) \equiv x^2+x \mod x^8+1$$

2.2. Der Ring $\mathbb{F}_{2^8}[x]$

In diesem Abschnitt betrachten wir Polynome mit Koeffizienten aus $\mathbb{F}_{2^8}$. Wir beschränken uns hier auf Polynome f mit $\deg f \leq 3$, so daß wir jedes Polynom als Koeffizientenvektor mit 4 Bytes schreiben können:

$$f = (f_3, f_2, f_1, f_0) \in \mathbb{F}_{2^8}^4$$

2.2.1. Die Addition

Um zwei Polynome $f, g \in \mathbb{F}_{2^8}[x]$ zu addieren, brauchen wir nur die entsprechenden Koeffizienten bitweise modulo 2 zu addieren (**XOR**).

2.2.2. Die Multiplikation über $\mathbb{F}_{2^8}[x]/\langle x^4+1\rangle$

Die Multiplikation zweier Polynome $f = f_3x^3 + f_2x^2 + f_1x + f_0$ und $g = g_3x^3 + g_2x^2 + g_1x + g_0$ ergibt ein Polynom h mit $\deg h \leq 6$ und den Koeffizienten:

$$h_k = \sum_{0\leq k\leq 6, i+j=k} f_i \cdot g_j$$

Die Reduktion modulo $m = x^4 + 1$ ist wieder einfach möglich durch

$$x^i \equiv x^{(i \mod 4)} \mod x^4 + 1$$

Da m über $\mathbb{F}_{2^8}[x]$ nicht irreduzibel ist, ist die Multiplikation im allgemeinen nicht invertierbar.

Wir betrachten daher nur die Multiplikation mit einem festen Polynom

$$g = 03_{16} \cdot x^3 + 01_{16} \cdot x^2 + 01_{16} \cdot x + 02_{16} \in \mathbb{F}_{2^8}[x]/\langle x^4 + 1\rangle,$$

das teilerfremd zu m ist ($\gcd(g, m) = 1$), so daß das multiplikative Inverse

$$g^{-1} = 0\mathrm{B}_{16} \cdot x^3 + 0\mathrm{D}_{16} \cdot x^2 + 09_{16} \cdot x + 0\mathrm{E}_{16} \in \mathbb{F}_{2^8}[x]/\langle x^4 + 1\rangle$$

existiert.

3. Beschreibung von Rijndael

« The enemy knows the system being used »

C.E. Shannon (1945) [1]

3.1. Allgemein

Rijndael (gesprochen: "reindahl") [2] ist eine symmetrische Blockchiffre, die von den beiden belgischen Kryptologen Vincent Rijmen und Joan Daemen entwickelt wurde [DR99]. Anders als viele gängige Verfahren (z.B. DES) basiert Rijndael nicht auf einem Feistel-Netzwerk, sondern auf einer sogenannten SQUARE-Struktur (benannt nach der Chiffre SQUARE [DKR97], die ebenfalls von den beiden Rijndael-Erfindern und Lars R. Knudsen entwickelt wurde.) Die grundlegenden Design-Ziele für Rijndael waren:

1. **Sicherheit**
   Das Design sollte größtmögliche Sicherheit gegenüber den bekannten Angriffen wie z.B. der linearen oder differentiellen Kryptoanalyse oder dem SQUARE-Angriff bieten.

2. **Geschwindigkeit und kompakter Code**
   Rijndael sollte aufgrund einer guten Performance (sowohl in Hardware als auch in Software) auf unterschiedlichen Systemen (8-Bit, 32-Bit, 64-Bit) ein breites Einsatzgebiet finden. Ein sehr kompakter Code sollte auch Anwendungen auf speicherkritischen Systemen wie z.B. Smartcards möglich machen.

3. **Einfachheit des Designs**
   Durch ein möglichst einfaches Design sollte die Analyse erleichtert und somit der Verdacht auf sogenannten Trapdoors bzw. verdeckte Schwachstellen ausgeräumt werden. Darüberhinaus wird auch die Implementierung erleichtert und Implementierungsfehler können vermieden werden.

[1] Kerckhoffs Prinzip nach Jean Guillaume Hubert Victor François Alexandre Auguste Kerckhoffs von Nieuwenhof (1835-1903), flämischer Professor, Verfasser des Buches "La cryptographie militaire" (1883)

[2] altflämisch für Rheintal

In diesem Kapitel werden die einzelnen Komponenten im Detail sowie der gesamte Ablauf einer Ver- bzw. Entschlüsselung beschrieben.

Zunächst aber wollen wir einige allgemeine Grundlagen zu symmetrischen Blockchiffren behandeln.

3.2. Hintergrund: symmetrische Blockchiffren

In diesem Abschnitt geben wir eine kurze Übersicht über allgemeine Aspekte und Begriffe symmetrischer Blockchiffren.

Die heutigen kryptographischen Verfahren kann man seit der Einführung der asymmetrischen Kryptographie von Diffie und Hellman im Jahre 1976 [DH76] in symmetrische und asymmetrische Verfahren unterteilen. Asymmetrische Verfahren verwenden zum Verschlüsseln und Entschlüsseln unterschiedliche Schlüssel (öffentlich und privat), so daß vor einer Kommunikation keine geheime Schlüsselinformation ausgetauscht werden muß.

In dieser Arbeit wollen wir uns aber mit den symmetrischen Verschlüsselungsverfahren, auch symmetrische Chiffren genannt, beschäftigen. Die symmetrischen Verschlüsselungsverfahren verwenden nur einen geheimen Schlüssel für die Ver- und Entschlüsselung. Für eine vertrauliche Kommunikation müssen also beide Teilnehmer den geheimen Schlüssel besitzen. Formal definiert:

Definition 3.2.1 (symmetrisches Verschlüsselungsverfahren) *Ein symmetrisches Verschlüsselungsverfahren ist ein Algorithmus, der aus der Eingabe des Klartextes $P \in \mathcal{P}$ (engl. Plaintext) und der Eingabe des Chiffrierschlüssels $K \in \mathcal{K}$ (engl. Key) als Ausgabe den (eindeutigen) Chiffretext $C \in \mathcal{C}$ (engl. Ciphertext) erzeugt. Die Verschlüsselung schreiben wir als $E_K(P) = C$. Zur Entschlüsselung wird der gleiche Schlüssel K verwendet (Symmetrie), mit $D_K(C) = P$.*

Man kann Verschlüsselungsverfahren aber auch danach einteilen, wie sie den Klartext verarbeiten. Betrachtet ein Verfahren den Klartext als Strom von einzelnen Bits bzw. Informationseinheiten und verschlüsselt jedes Bit abhängig von der Position, so sprechen wir von einer Stromchiffre. Eine Blockchiffre hingegen, verarbeitet einen Klartext blockweise, d.h. sie zerlegt den Klartext in Blöcke fester Länge und verschlüsselt jeden Block einzeln (siehe [Knu98] und [Rob94]). Eine Blockchiffre, die Blöcke der Länge n-Bit verarbeitet, bezeichnen wir als n-Bit Blockchiffre.

Definition 3.2.2 (symmetrische Blockchiffre) *Sei T_n die Menge aller Textblöcke der Länge n. Eine n-Bit Block-Chiffre ist eine Funktion $E : T_n \times \mathcal{K} \to T_n$, so daß für jeden Schlüssel $K \in \mathcal{K}$, $E_K(P) = C$ (Verschlüsselung) eine invertierbare Abbildung liefert. Die inverse Abbildung ist $D_K(C) = P$ (Entschlüsselung) mit demselben Schlüssel K.*

In der Kryptographie sind Blockchiffren so beliebt, weil man sie als Grundbaustein für andere kryptographische Anwendungen einsetzen kann. Außer der Verschlüsselung als reine Blockchiffre wären hier z.B. noch Stromchiffren, kryptographische Hashfunktionen, Message Authentication Codes (MAC's) und Pseudozufallsgeneratoren zu nennen. Bekannte Beispiele für symmetrische Blockchiffren sind z.B. DES [oS77], IDEA [LM90] und natürlich Rijndael [DR99].

3.2.1. Design-Aspekte von Blockchiffren

Eine grundlegende Bedingung für die problemlose Verwendung einer Blockchiffre ist, daß der Klartextblock aus einem Chiffretextblock eindeutig rekonstruierbar sein muß. Die Abbildung von der Menge der Klartexte $\mathcal{P}$ in die Menge der Chiffretexte $\mathcal{C}$ sollte also injektiv sein. Bei einer Blocklänge von n Bit müssen 2^n mögliche Klartextblöcke auf 2^n Chiffretextblöcke abgebildet werden. Betrachtet man eine solche Abbildung als Permutation der Klartextblöcke, so erkennt man, daß es $2^n!$ mögliche Abbildungen gibt. Eine Blockchiffre wählt anhand des geheimen Schlüssels eine solche Abbildung für die Verschlüsselung aus. Die Länge des Schlüssels ist ausschlaggebend dafür, wie viele verschiedene Abbildungen aus den $2^n!$ gewählt werden können. Bei einer Schlüssellänge von m Bits, existieren 2^m mögliche Schlüssel, von denen jeder eine Abbildung der Klartextblöcke auf die Chiffretextblöcke bestimmt. Es steht also für praktikable Schlüssellängen und Blocklängen (64 - 512 Bit) nur ein "winziger" Bruchteil der möglichen Abbildung zur Verfügung, nämlich $2^m/(2^n)!$. Für eine Chiffre mit gleicher Block- und Schlüssellänge ergibt sich ein Anteil von $1/(2^n-1)!$.

Ziel eines guten Designs ist es, die geheime Schlüsselinformation so einzusetzen, daß quasi zufällige Abbildungen gewählt werden. Shannon bezeichnet eine Chiffre, die die Abbildungen echt zufällig wählt in [Sha49] als «random cipher». Da man "echten" Zufall aber nicht in einer effizienten Blockchiffre einsetzen kann, sollte der Chiffretextblock zumindest auf möglichst komplizierte Weise von Klartextblock und Schlüssel abhängen. Hierzu führte Shannon in [Sha49] die Konzepte der Konfusion und Diffusion ein, die noch heute die Grundlage der Entwicklung von Blockchiffren bilden.

Konfusion steht für die Eigenschaft der Chiffre, die Beziehung zwischen Chiffretext und Schlüssel so kompliziert wie möglich zu machen. Konfusion kann z.B. durch Substitution erreicht werden, bei der ein Klartextzeichen durch ein anderes Zeichen ersetzt wird.

Diffusion steht für eine komplizierte Beziehung zwischen Klar- und Chiffretext. Dabei soll die Chiffre den Einfluß jedes Klartextzeichens über den gesamten Chiffretext streuen, um somit statistische Eigenschaften des Klartextes zu verbergen. Für diese Eigenschaft eignet sich eine Transposition, die die einzelnen Klartextzeichen permutiert mit einer anschließenden “Streufunktion”, die den Einfluß eines Bytes auf mehrere Bytes verteilt.

3.2.2. Produktchiffren

Ein naheliegender Weg, ein Verschlüsselungsverfahren mit guter Konfusion und guter Diffusion zu erhalten, ist die Hintereinanderausführung verschiedener Chiffren, z.B. einer Substitution und einer Transposition. Ein solches Verschlüsselungsverfahren wird nach Shannon [Sha49] **Produktchiffre** genannt.

Um die Komplexität (und damit hoffentlich die Sicherheit) einer unsicheren Chiffre zu erhöhen, kann man diese öfter hintereinander auf den Klartext anwenden. Die Chiffre wird dabei als Rundenfunktion bezeichnet und die resultierende Chiffre wird **iterierte Blockchiffre** oder auch **rundenbasierte Blockchiffre** genannt. Die Rundenfunktion sollte dabei aus mindestens 2 Operationen bestehen, deren Reihenfolge nicht vertauschbar ist. Sloane vergleicht dieses Prinzip Shannons einer “guten Durchmischung” in [Slo83] mit der Herstellung von Blätterteig (siehe Abb. 3.1). Die beiden nicht vertauschbaren “Operationen” Ausrollen und Falten bilden

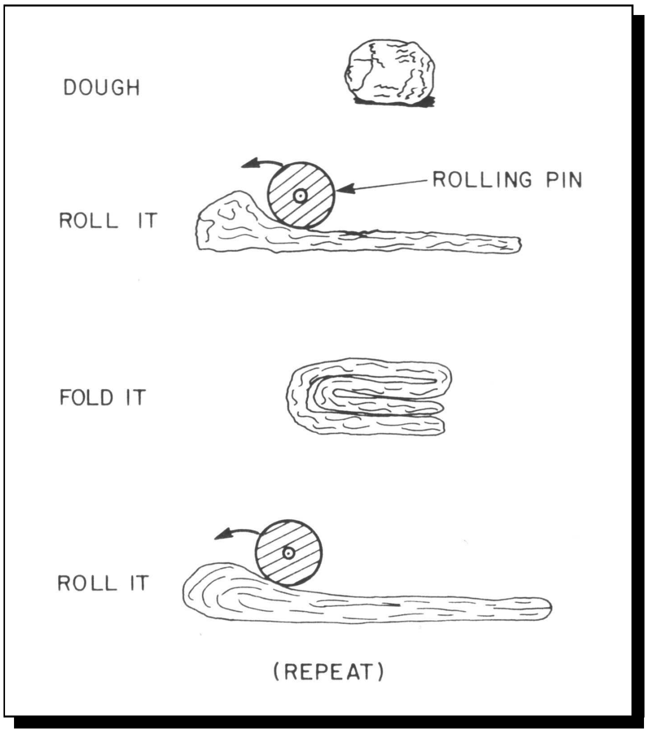

Abbildung 3.1.: Herstellung von Blätterteig

die Rundenfunktion, die nach genügend häufiger Ausführung einen guten Teig liefert. Aber wie oft muß man rollen und falten bis der Teig gut ist ?

An dem Beispiel des Blätterteigs wird der natürliche Trade-Off beim Design einer iterierten Blockchiffre erkennbar. Die Anzahl der Runden ist entscheidend für die Performance und die Sicherheit: Eine hohe Rundenzahl verspricht eine hohe Sicherheit, aber eine schlechte Performance und umgekehrt.
O'Connor schreibt in [OG94]:

> «... almost all product ciphers become immune ... after sufficient many rounds»

Massey kontert

> « most ciphers are too slow after sufficient many rounds »

Die notwendige Anzahl der Runden hängt von der Qualität der Rundenfunktion ab. Die rundenbasierte Blockchiffre sollte auf jeden Fall Sicherheit gegenüber den bekannten Angriffen bieten. In Kapitel 4 beschäftigen wir uns intensiver mit dem Thema Angriffe auf Blockchiffren.

3.3. Grundlegendes zu Rijndael

Rijndael ist eine symmetrische Blockchiffre, die unabhängig voneinander sowohl Blocklängen als auch Schlüssellängen von $128, 192$ bzw. 256 Bits erlaubt. Für den Advanced Encryption Standard wurde jedoch die Blocklänge auf 128 Bit beschränkt.

Eingabe und Ausgabe von Rijndael sind Textblöcke, also Bit-Strings mit Blocklänge (128,192 oder 256 Bits). Intern werden Zwischenergebnisse beim Ver- bzw. Entschlüsseln aber in sogenannten Zuständen (engl. states) gespeichert.

Definition 3.3.1 (Zustand) *Sei $N_b \in \{4, 6, 8\}$ die Blocklänge geteilt durch 32. Ein Zustand ist ein zweidimensionales Byte-Feld bestehend aus N_b Spalten mit jeweils 4 Bytes (32 Bit), in dem ein Zwischenergebnis der Ver- bzw. Entschlüsselung abgelegt wird.*

Ein Byte innerhalb eines Zustands Z wird mit

$$Z_{i,j}, i \in \{0, \ldots, 3\}, j \in \{0, \ldots, N_b - 1\}$$

beschrieben.

Transformation eines Strings in einen Zustand Die Bytes des Klartextblocks P werden spaltenweise in dem Byte-Array Z angeordnet, d.h. die ersten 4 Bytes werden in die erste Spalte kopiert, die zweiten 4 Bytes in die zweite Spalte usw. Das Byte P_i des Klartextblocks P wird an Position (y, x), mit

$$x = \lfloor i/4 \rfloor$$

$$y = i \mod 4$$

des Zustands geschrieben.

Seien P_i mit $0 \leq i \leq$ Blocklänge/8 die Bytes des Klartextblocks. Für eine Blocklänge von 128 Bit (16 Bytes), also $N_b = 4$, erfolgt die Umsetzung folgendermaßen:

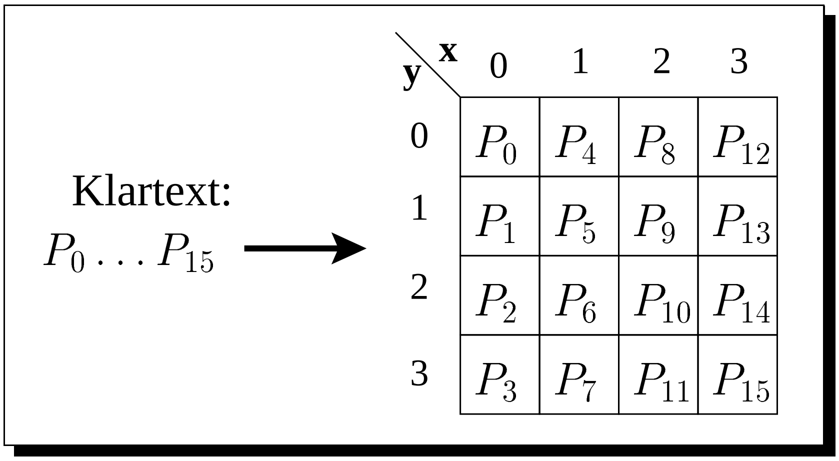

Abbildung 3.2.: Transformation eines 128 Bit Klartextblocks in einen Zustand

Die Ausgabe nach der Ent- bzw. Verschlüsselung erfolgt wieder als eindimensionales Byte-Feld, wobei die Umwandlung nach dem gleichen Schema wieder spaltenweise durchgeführt wird.

Der Chiffrierschlüssel wird ebenfalls nach diesem Schema in ein Byte-Array bestehend aus 4 Zeilen und $N_k = 4, 6$ bzw. 8 Spalten geschrieben, je nach Schlüssellänge ($N_k =$ Schlüssellänge/32).

Die Ver- bzw. Entschlüsselung erfolgt in Runden, deren Anzahl N_r von dem Maximum der Blockgröße (bzw. N_b) und der Schlüssellänge (bzw. N_k) abhängt:

$$N_r = \max(N_b, N_k) + 6$$

Tabelle 3.1 listet die Rundenanzahl für alle möglichen Fälle auf.

N_r	$N_b = 4$	$N_b = 6$	$N_b = 8$
$N_k = 4$	10	12	14
$N_k = 6$	12	12	14
$N_k = 8$	14	14	14

Tabelle 3.1.: Anzahl der Runden (N_r) in Abhängigkeit von N_b und N_k

$\mathbf{N_b}$	= Blockgröße/32	Anzahl der Spalten eines Zustands
$\mathbf{N_k}$	= Schlüssellänge/32	Anzahl der Spalten des Chiffrierschlüssels
$\mathbf{N_r}$	$= \max(N_b, N_k) + 6$	Anzahl der Runden

Abbildung 3.3.: Übersicht über die wichtigsten Parameter

3.4. Ver- und Entschlüsselung mit Rijndael

Bei der Verschlüsselung werden in einer Runde 4 Transformationen auf den Zustand angewendet. Im folgenden wird zunächst jede der 4 Transformationen ByteSub, ShiftRow, MixColumn und KeyAdd detailliert beschrieben und danach der Ablauf der gesamten Verschlüsselung dargestellt.

3.4.1. ByteSub

ByteSub ist eine nichtlineare Substitution, die jedes Byte eines Zustands unabhängig von den anderen Bytes ersetzt. Sie besteht aus 2 Teilen, einer Invertierung und einer affinen Abbildung.

Verschlüsselung Für die Invertierung fassen wir jedes Byte p als Element des Körpers $\mathbb{F}_2[x]/\langle m\rangle = \mathbb{F}_{2^8}$ auf, mit irreduziblem Polynom

$$m = x^8 + x^4 + x^3 + x + 1 \in \mathbb{F}_2[x]$$

Da m irreduzibel ist, hat jedes Element $p \neq 0$ ein multiplikatives Inverses p^{-1} und wir definieren die Abbildung inv: $\mathbb{F}_{2^8} \to \mathbb{F}_{2^8}$

$$inv(p) = \begin{cases} 0 & \text{falls } p = 0 \\ p^{-1} & \text{sonst} \end{cases}$$

Für die anschließende affine Abbildung betrachten wir jedes $p = (p_0, \ldots, p_7)$ als Element des Rings $R = \mathbb{F}_2[x]/\langle x^8 + 1 \rangle$. ($x^8 + 1$ ist über $\mathbb{F}_2[x]$ nicht irreduzibel !) In diesem Ring wird nun die Abbildung

$$(x^4 + x^3 + x^2 + x + 1) \cdot p + (x^6 + x^5 + x + 1) \mod x^8 + 1$$

durchgeführt, bzw. als Matrixmultiplikation

$$\begin{bmatrix} 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 \end{bmatrix} \cdot \begin{bmatrix} p_0 \\ p_1 \\ p_2 \\ p_3 \\ p_4 \\ p_5 \\ p_6 \\ p_7 \end{bmatrix} + \begin{bmatrix} 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \\ 0 \end{bmatrix}$$

Beispiel 3.4.1 *Berechne die Ausgabe von ByteSub für $p = 123_{10}$ Die Binärdarstellung von $123_{10} = 01111011_2$ führt zu dem Polynom*

$$a = x^6 + x^5 + x^4 + x^3 + x + 1 \in \mathbb{F}_2[x]/\langle m \rangle$$

Von diesem Polynom wird nun das Inverse $\mod m$ berechnet:

$$a^{-1} \equiv x^2 + x \pmod{m} =: b \in \mathbb{F}_2[x]/\langle m \rangle$$

Im zweiten Teil der Substitution wird nun die affine Abbildung

$$(x^4 + x^3 + x^2 + x + 1) \cdot b + (x^6 + x^5 + x + 1) \equiv x^5 + 1 \pmod{x^8 + 1}$$

angewendet und das Ergebnis wieder über die Binärdarstellung $00100001_2 = 33_{10}$ in das Dezimalsystem umgewandelt.

Entschlüsselung Damit eine Umkehrung von ByteSub eindeutig möglich ist, muß jede einzelne Operation invertierbar sein. Da die Invertierung, als erster Teil von ByteSub, über einem Körper stattfindet, ist sie bijektiv und somit auch umkehrbar. Die affine Abbildung dagegen wird über dem Ring $R = \mathbb{F}_2[x]/\langle x^8 + 1 \rangle$ berechnet, in dem eine Umkehrung einer Multiplikation im allgemeinen nicht möglich ist. Da aber $ggT(x^4 + x^3 + x^2 + x + 1, x^8 + 1) = 1$ ist, ist auch hier die inverse Abbildung eindeutig definiert.

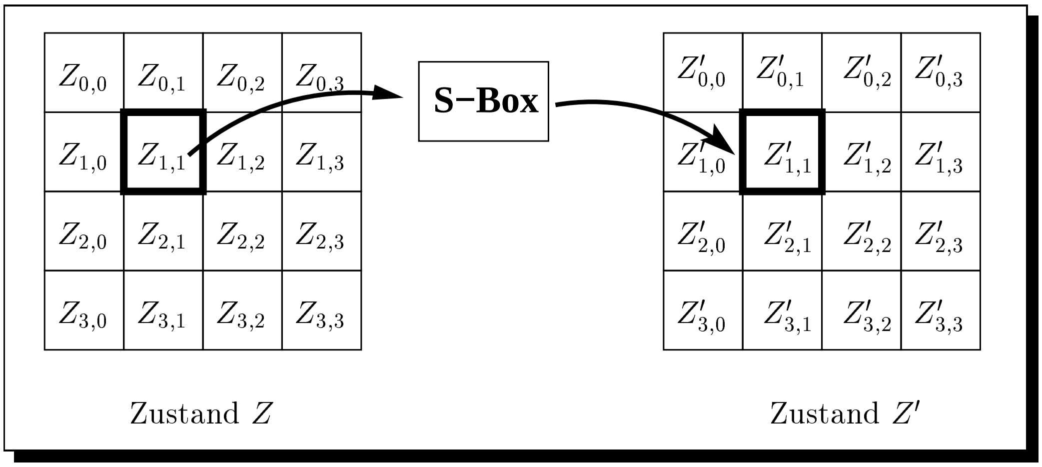

Abbildung 3.4.: ByteSub

Bemerkungen Für eine effizientere Implementierung kann man die ByteSub Transformation für alle 256 möglichen Werte von p vorberechnen und in einer sogenannten Substituions-Box (kurz S-Box) ablegen. Im weiteren Verlauf werden wir eine Substitution eines Bytes p mit $\mathbf{S}[p]$ abkürzen.

ByteSub ist die einzige nichtlineare Komponente von Rijndael. Kaisa Nyberg hat in [Nyb93] gezeigt, daß eine Invertierung über $GF(2^8)$ sehr gute Eigenschaften für eine Substitutions-Box liefert. Diese Eigenschaft ist entscheidend für die Resistenz von Rijndael gegenüber differentieller und linearer Analyse.

3.4.2. ShiftRow

Verschlüsselung Beim ShiftRow werden die einzelnen Zeilen eines Zustands zyklisch um verschiedene Offsets nach links geschoben. Die Zeile 0 wird nicht geshiftet, Zeile 1 um $C1$ Bytes Zeile 2 um $C2$ Bytes und Zeile 3 um $C3$ Bytes. Die Werte $C1, C2, C3$ hängen von dem Parameter N_b ab und sind in Tabelle 3.2 definiert.

Für den Andvanced Encryption Standard ist $N_b = 4$, so daß die erste Zeile nicht verschoben wird, die zweite Zeile um 1 Byte, die zweite Zeile um 2 Bytes und die dritte Zeile um 3 Bytes.

Entschlüsselung Für die Entschlüsselung kann ebenfalls Tabelle 3.2 verwendet werden, nur wird dann nach rechts anstatt nach links geshiftet.

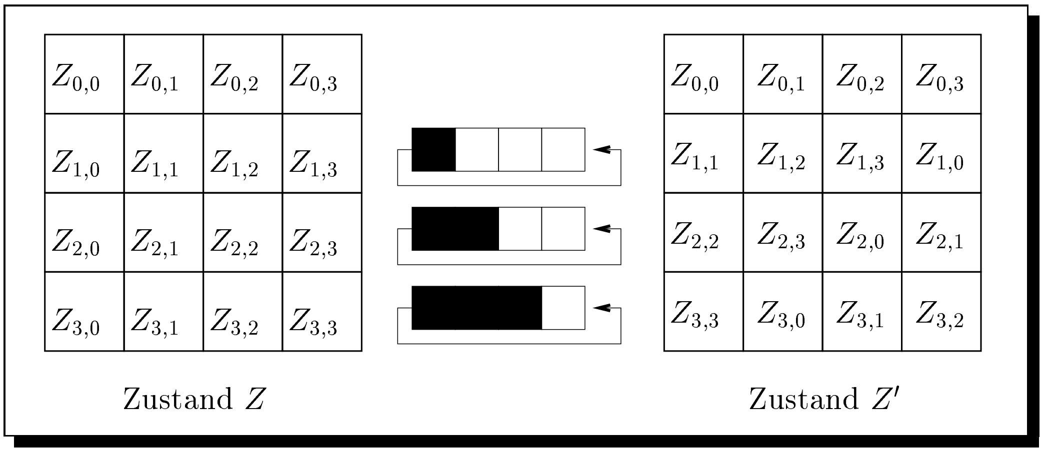

Abbildung 3.5.: ShiftRow

N_B	$C1$	$C2$	$C3$
4	1	2	3
6	1	2	3
8	1	3	4

Tabelle 3.2.: Shift-Offsets in Abhängigkeit der Blocklänge

3.4.3. MixColumn

MixColumn bildet Linearkombinationen der 4 Bytes einer Spalte. Hier wird eine Spalte des Zustands (4 Bytes) als Polynom über dem Ring $\mathbb{F}_{2^8}[x]/\langle x^4+1\rangle$ aufgefaßt und mit einem festen Polynom

$$c = 03 \cdot x^3 + 01 \cdot x^2 + 01 \cdot x + 02 \in \mathbb{F}_{2^8}[x]$$

multipliziert (siehe Abb.3.6).

Verschlüsselung Eine Spalte des Zustands bestehend aus den Bytes (m_0, m_1, m_2, m_3) wird als Polynom

$$m = m_3x^3 + m_2x^2 + m_1x + m_0 \in \mathbb{F}_{2^8}[x]$$

verstanden. Es erfolgt eine Multiplikation mit einem festen Polynom c

$$n = m \cdot \underbrace{(03x^3 + 01x^2 + 01x + 02)}_{c} \pmod{x^4+1}$$

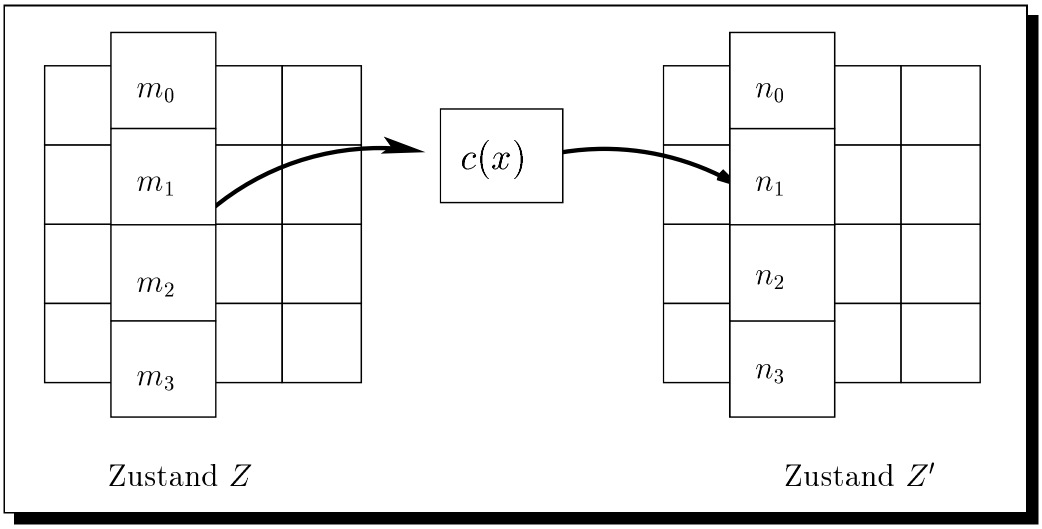

Abbildung 3.6.: MixColumn

über dem Ring $\mathbb{F}_{2^8}[x]/\langle x^4+1\rangle$. (Die Reduktion ist einfach durch

$$x^i \equiv x^{(i \mod 4)} \mod x^4+1$$

möglich.) Diese Transformation ist auch als Matrixmultiplikation darstellbar:

$$\underbrace{\begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix}}_{=\mathbf{A}\in\mathbb{F}_{2^8}^{4\times 4}} \cdot \begin{bmatrix} m_0 \\ m_1 \\ m_2 \\ m_3 \end{bmatrix} = \begin{bmatrix} n_0 \\ n_1 \\ n_2 \\ n_3 \end{bmatrix}$$

Die Wahl des Polynoms c beruht, neben der Eigenschaft invertierbar zu sein, auf Performancegründen: Es tauchen nur folgende Koeffizienten aus $\mathbb{F}_{2^8}$ auf:

01 (keine Multiplikation nötig),

02 (Multiplikation entspricht zyklischem Shift um 1 Bit nach links) und

03 ($= 01 \oplus 02$) (Multiplikation entspricht zyklischem Shift um 1 Bit nach links und anschließendem **XOR** mit dem ursprünglichen Byte)

Diese Koeffizienten ermöglichen eine effiziente Implementierung von MixColumn.

Beispiel 3.4.2 *Sei* $M = \begin{bmatrix} m_0 \\ m_1 \\ m_2 \\ m_3 \end{bmatrix}$ *eine Spalte eines Zustands z.B.* $M = \begin{bmatrix} 01 \\ 02 \\ 03 \\ 04 \end{bmatrix}$.
Die Multiplikation mit $\mathbf{A}$ *ergibt:*

$$\begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix} \cdot \begin{bmatrix} 01 \\ 02 \\ 03 \\ 04 \end{bmatrix} = \begin{bmatrix} 03 \\ 04 \\ 09 \\ 10 \end{bmatrix}$$

Entschlüsselung $x^4 + 1$ ist zwar über $\mathbb{F}_{2^8}[x]$ nicht irreduzibel, aber da $x^4 + 1$ und c teilerfremd sind, ist auch die Umkehrabbildung eindeutig definiert. Das für die Entschlüsselung benötigte feste Polynom in Hexadezimalschreibweise lautet:

$$d = (0Bx^3 + 0Dx^2 + 09x + 0E) \in \mathbb{F}_{2^8}[x]$$

bzw. die Matrix

$$\mathbf{A^{-1}} = \begin{bmatrix} 0E & 0B & 0D & 09 \\ 09 & 0E & 0B & 0D \\ 0D & 09 & 0E & 0B \\ 0B & 0D & 09 & 0E \end{bmatrix}$$

Wenn man die Matrix $\mathbf{A^{-1}}$ mit der Matrix $\mathbf{A}$ vergleicht, so stellt man fest, daß hier die Einträge wesentlich größer sind und eine "echte" Multiplikation erfordern. Daraus ergibt sich für die Entschlüsselung der Nachteil, daß das inverse MixColumn auf schwachen Prozessoren (8 Bit) deutlich mehr Zeit benötigt, da hier insbesondere die Vereinfachungen wie beim Verschlüsseln nicht angewendet werden können.

Bemerkungen

Lemma 3.4.1 *Bei Eingabe von* 2 *Spalten, die sich in* $l \in \{1, 2, 3, 4\}$ *Bytes unterscheiden, differieren die beiden Ausgabespalten in mindestens* $5 - l$ *Bytes.*

Beweis. Wir beweisen hier nur den Fall $l = 1$, den wir später im Kapitel über Kryptoanalyse anwenden.
Zu zeigen ist also, daß bei Eingabe von 2 Spalten, die sich in einem Byte unterscheiden, die beiden Ausgabespalten in allen 4 Bytes unterscheiden.
Seien M und M' Spalten zweier Zustände. O.B.d.A. sei $M = (m_0, m_1, m_2, m_3)$ und $M' = (m'_0, m_1, m_2, m_3)$ mit $m_0 \neq m'_0$ und N bzw. N' die Ausgabespalten.

Betrachten wir das MixColumn in Matrixschreibweise für die Spalte M

$$\underbrace{\begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix}}_{=\mathbf{A}\in\mathbb{F}_{2^8}^{4\times 4}} \cdot \underbrace{\begin{bmatrix} m_0 \\ m_1 \\ m_2 \\ m_3 \end{bmatrix}}_{M} = \begin{bmatrix} 02\cdot m_0 \oplus 03\cdot m_1 \oplus 01\cdot m_2 \oplus 01\cdot m_3 \\ 01\cdot m_0 \oplus 02\cdot m_1 \oplus 03\cdot m_2 \oplus 01\cdot m_3 \\ 01\cdot m_0 \oplus 01\cdot m_1 \oplus 02\cdot m_2 \oplus 03\cdot m_3 \\ 03\cdot m_0 \oplus 01\cdot m_1 \oplus 01\cdot m_2 \oplus 02\cdot m_3 \end{bmatrix} = \underbrace{\begin{bmatrix} n_0 \\ n_1 \\ n_2 \\ n_3 \end{bmatrix}}_{=:N}$$

und M'

$$\underbrace{\begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix}}_{=\mathbf{A}\in\mathbb{F}_{2^8}^{4\times 4}} \cdot \underbrace{\begin{bmatrix} m_0' \\ m_1 \\ m_2 \\ m_3 \end{bmatrix}}_{M'} = \begin{bmatrix} 02\cdot m_0' \oplus 03\cdot m_1 \oplus 01\cdot m_2 \oplus 01\cdot m_3 \\ 01\cdot m_0' \oplus 02\cdot m_1 \oplus 03\cdot m_2 \oplus 01\cdot m_3 \\ 01\cdot m_0' \oplus 01\cdot m_1 \oplus 02\cdot m_2 \oplus 03\cdot m_3 \\ 03\cdot m_0' \oplus 01\cdot m_1 \oplus 01\cdot m_2 \oplus 02\cdot m_3 \end{bmatrix} = \underbrace{\begin{bmatrix} n_0' \\ n_1' \\ n_2' \\ n_3' \end{bmatrix}}_{=:N'}$$

Betrachten wir nun die Differenz von N und N'

$$\begin{bmatrix} 02\cdot(m_0 \oplus m_0') \\ 01\cdot(m_0 \oplus m_0') \\ 01\cdot(m_0 \oplus m_0') \\ 03\cdot(m_0 \oplus m_0') \end{bmatrix}.$$

Unter der Voraussetzung, daß $m_0 \neq m_0'$ ist, ist die Differenz in allen 4 Bytes $\neq 0$, N und N' unterscheiden sich also in allen 4 Bytes. □

Diese Eigenschaft nennen wir im folgenden Maximum Distance Separability, kurz MDS und die Matrix $\mathbf{A}$ nennen wir MDS-Matrix. Man kann sich leicht überlegen, daß die MDS-Eigenschaft auch für das inverse MixColumn gilt. In der Codierungstheorie bezeichnet man eine solche Abbildung als Code mit Abstand 5.

Die MDS-Eigenschaft führt in Verbindung mit dem ShiftRow zu einer vollen Diffusion nach 2 Runden, d.h. nach 2 Runden hat jedes Klartextbyte jedes Byte des Zustands beeinflußt.

3.4.4. KeyAdd

Zur Einbringung der geheimen Schlüsselinformation in den Verschlüsselungsprozeß dient die Transformation KeyAdd. In jeder Runde fließt ein eigener Rundenschlüssel ein, der über den KeySchedule (vgl. Abschnitt 3.5) vom Chiffrierschlüssel abgeleitet wird. Ein Rundenschlüssel ist ein Byte-Array von der Größe eines Zustandes, so daß eine byteweise Verknüpfung per **XOR** möglich ist.

Verschlüsselung Als Eingabe erhält die KeyAdd-Transformation den Zustand und einen Rundenschlüssel, der zuvor mittels KeySchedule aus dem Chiffrierschlüssel erzeugt wurde. Der Rundenschlüssel hat wie der Zustand 4 Zeilen und auch N_b Spalten. Zur Verschlüsselung werden nun die Bytes an den jeweiligen Positionen

$$\underbrace{\begin{array}{|c|c|c|c|}\hline Z_{0,0} & Z_{0,1} & Z_{0,2} & Z_{0,3} \\ \hline Z_{1,0} & Z_{1,1} & Z_{1,2} & Z_{1,3} \\ \hline Z_{2,0} & Z_{2,1} & Z_{2,2} & Z_{2,3} \\ \hline Z_{3,0} & Z_{3,1} & Z_{3,2} & Z_{3,3} \\ \hline \end{array}}_{\text{Zustand } Z} \bigoplus \underbrace{\begin{array}{|c|c|c|c|}\hline K_{0,0} & K_{0,1} & K_{0,2} & K_{0,3} \\ \hline K_{1,0} & K_{1,1} & K_{1,2} & K_{1,3} \\ \hline K_{2,0} & K_{2,1} & K_{2,2} & K_{2,3} \\ \hline K_{3,0} & K_{3,1} & K_{3,2} & K_{3,3} \\ \hline \end{array}}_{\text{Rundenschlüssel } K} = \underbrace{\begin{array}{|c|c|c|c|}\hline Z'_{0,0} & Z'_{0,1} & Z'_{0,2} & Z'_{0,3} \\ \hline Z'_{1,0} & Z'_{1,1} & Z'_{1,2} & Z'_{1,3} \\ \hline Z'_{2,0} & Z'_{2,1} & Z'_{2,2} & Z'_{2,3} \\ \hline Z'_{3,0} & Z'_{3,1} & Z'_{3,2} & Z'_{3,3} \\ \hline \end{array}}_{\text{Zustand } Z'}$$

Abbildung 3.7.: KeyAdd

mittels **XOR** verknüpft.

Entschlüsselung Da die **XOR** -Operation selbstinvers ist, folgt daraus, daß auch die KeyAdd-Transformation selbstinvers ist. Es wird lediglich ein modifizierter KeySchedule benötigt, so daß die Rundenschlüssel in umgekehrter Reihenfolge angewendet werden.

3.4.5. Aufbau einer Runde

Aus den oben beschriebenen Transformationen wird nun eine Verschlüsselungsrunde zusammengestellt. Der Ablauf innerhalb einer Runde ist:

1.	**ByteSub**	(nichtlineare Substitution)
2.	**ShiftRow**	(zyklische Verschiebung innerhalb einer Zeile)
3.	**MixColumn**	(Mischen einer Spalte)
4.	**KeyAdd**	(Addieren des Rundenschlüssels)

Abbildung 3.8.: Aufbau einer Runde

Die Reihenfolge der Transformationen läßt sich bedingt abwandeln:

1. ByteSub und ShiftRow können vertauscht werden.
   Die beiden Transformationen sind vertauschbar, weil ByteSub jedes Byte

1.	**ByteSub**	(nichtlineare Substitution)
2.	**ShiftRow**	(zyklische Verschiebung innerhalb einer Zeile)
3.	**KeyAdd'**	(Addieren des modifizierten Rundenschlüssels)
4.	**MixColumn**	(Mischen einer Spalte)

Abbildung 3.9.: Alternativer Aufbau einer Runde

unabhängig von seiner Position ersetzt und ShiftRow die Position eines Bytes unabhängig von dessen Wert verändert.

2. MixColumn und KeyAdd sind vertauschbar.
   Da MixColumn und InvMixColumn lineare Transformationen sind, gilt für jeden Zustand Z:
   InvMixColumn(Z **XOR** Rundenschlüssel) = InvMixColumn(Z) **XOR** InvMixColumn(Rundenschlüssel).
   Eine Vertauschung führt zu einem alternativen Aufbau einer Runde (Abb.3.9) mit InvMixColumn(Rundenschlüssel) als modifiziertem Rundenschlüssel.

3.4.6. Gesamte Verschlüsselung

Die Verschlüsselung eines Textblocks erfolgt nun durch mehrfache Ausführung der Rundenfunktion (Abb. 3.8). Lediglich am Anfang und am Ende werden kleinere Änderungen vorgenommen:

- Vor der ersten Runde wird zunächst ein KeyAdd angewendet. Der Grund hierfür ist, daß sonst in den ersten 3 Rundentransformationen noch keinerlei geheime Information eingeflossen ist und die Transformationen somit keine Sicherheit bieten.

- Die letzte Runde wird ohne MixColumn ausgeführt. Die Autoren von Rijndael entschieden sich, das MixColumn in der letzten Runde auszulassen, da es keinerlei zusätzliche Sicherheit bietet und eine schönere Struktur beim Entschlüsseln verhindern würde (siehe 3.4.7).

Die Verschlüsselungsvorschrift haben wir in Abb. 3.10 dargestellt.

3.4.7. Gesamte Entschlüsselung

Die Entschlüsselung eines chiffrierten Textblocks geschieht durch Anwendung der Transformationen in umgekehrter Reihenfolge.

1.	Initiales KeyAdd
2.	$N_r - 1$ volle Runden
3.	letzte Runde ohne MixColumn

Abbildung 3.10.: Verschlüsselung

Macht man sich jedoch die in Abschnitt 3.4.5 beschriebenen Vertauschungsmöglichkeiten der einzelnen Transformationen zu nutze, so bekommt man exakt die gleiche Reihenfolge der inversen Transformationen wie bei einer Verschlüsselung. Lediglich der KeySchedule muß modifiziert werden, indem er auf jeden Rundenschlüssel, außer dem ersten und dem letzten, ein InvMixColumn anwendet.

In Abb. 3.11 wird beschrieben, wie die Umsetzung des ursprünglichen Ablaufs, bei dem die Transformationen in umgekehrter Reihenfolge durchlaufen werden, in den veränderten Ablauf, der die gleiche Struktur wie ein Verschlüsselungsablauf hat, erfolgt. Eine **1** steht dafür, daß ByteSub und ShiftRow vertauscht werden und die **2** dafür, daß MixColumn und KeyAdd vertauscht werden.

3.4.8. Effiziente Implementierung einer Runde

Betrachtet man die arithmetischen Definitionen der einzelnen Transformationen auf einer Spalte, so erzielt man durch Umformungen eine andere Darstellung einer Runde. Diese Darstellung führt zu einer effizienteren Implementierung, die nur aus S-Box Lookups und KeyAdd's besteht. Wir kürzen die Transformationen ByteSub durch BS, ShiftRow durch SR, MixColumn durch MC und KeyAdd durch KA ab.

Zu Beginn einer Runde werden die 4 Bytes $a_{0,j}, a_{1,j+C1}, a_{2,j+C2}, a_{3,j+C3}$ durch ByteSub und ShiftRow transformiert:

$$\begin{bmatrix} a_{0,j} \\ a_{1,j+C1} \\ a_{2,j+C2} \\ a_{3,j+C3} \end{bmatrix} \rightarrow \begin{bmatrix} \mathbf{S}[a_{0,j}] \\ \mathbf{S}[a_{1,j+C1}] \\ \mathbf{S}[a_{2,j+C2}] \\ \mathbf{S}[a_{3,j+C3}] \end{bmatrix}$$

und bilden dann die j-te Spalte Die Anwendung von MixColumn und KeyAdd führen zu

$$\begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix} \cdot \begin{bmatrix} \mathbf{S}[a_{0,j}] \\ \mathbf{S}[a_{1,j+C_1}] \\ \mathbf{S}[a_{2,j+C_2}] \\ \mathbf{S}[a_{3,j+C_3}] \end{bmatrix} \oplus \begin{bmatrix} k_{0,j} \\ k_{1,j} \\ k_{2,j} \\ k_{3,j} \end{bmatrix}$$

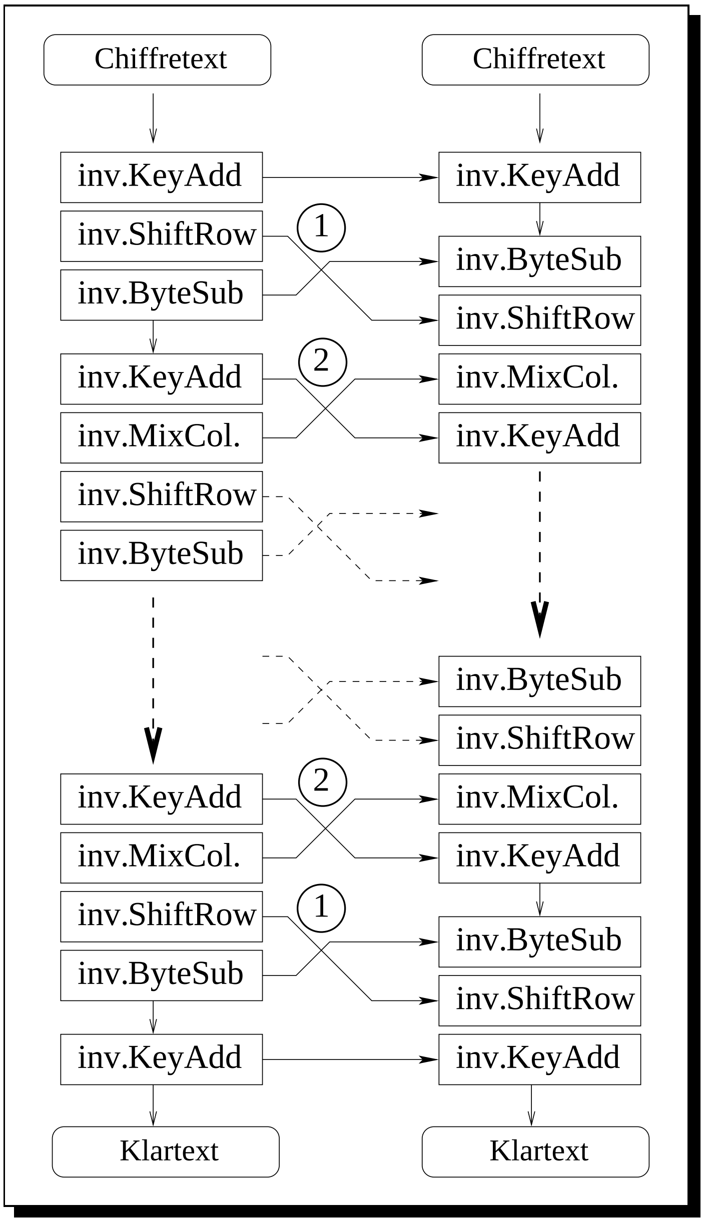

Abbildung 3.11.: Ablauf einer Entschlüsselung

Die Matrixmultiplikation kann man nun als Linearkombination von Vektoren schreiben:

$$\mathbf{S}[a_{0,j}]\begin{bmatrix}02\\01\\01\\03\end{bmatrix}\oplus\mathbf{S}[a_{1,j+C1}]\begin{bmatrix}03\\02\\01\\01\end{bmatrix}\oplus\mathbf{S}[a_{2,j+C2}]\begin{bmatrix}01\\03\\02\\01\end{bmatrix}\oplus\mathbf{S}[a_{3,j+C2}]\begin{bmatrix}01\\01\\03\\02\end{bmatrix}\oplus\begin{bmatrix}k_{0,j}\\k_{1,j}\\k_{2,j}\\k_{3,j}\end{bmatrix}$$

Aus dieser Form der Darstellung lassen sich nun 4 neue S-Boxen $\mathbf{T_0}[a]\ldots\mathbf{T_3}[a]$ erzeugen:

$$\mathbf{T_0}[a]=\begin{bmatrix}\mathbf{S}[a]\cdot 02\\\mathbf{S}[a]\\\mathbf{S}[a]\\\mathbf{S}[a]\cdot 03\end{bmatrix},\mathbf{T_1}[a]=\begin{bmatrix}\mathbf{S}[a]\cdot 03\\\mathbf{S}[a]\cdot 02\\\mathbf{S}[a]\\\mathbf{S}[a]\end{bmatrix},$$

$$\mathbf{T_2}[a]=\begin{bmatrix}\mathbf{S}[a]\\\mathbf{S}[a]\cdot 03\\\mathbf{S}[a]\cdot 02\\\mathbf{S}[a]\end{bmatrix},\mathbf{T_3}[a]=\begin{bmatrix}\mathbf{S}[a]\\\mathbf{S}[a]\\\mathbf{S}[a]\cdot 03\\\mathbf{S}[a]\cdot 02\end{bmatrix}$$

$\mathbf{T_0}[a]\ldots\mathbf{T_3}[a]$ haben jeweils 256 Einträge zu je 32 Bit und damit eine Größe von je 1KByte. 4 Bytes können nun mittels

$$\mathbf{T_0}[a_{0,j}]\oplus\mathbf{T_1}[a_{1,j+C1}]\oplus\mathbf{T_2}[a_{2,j+C2}]\oplus\mathbf{T_3}[a_{3,j+C3}]\oplus k_j$$

eine Runde verschlüsselt werden.

Wie in Abschnitt 3.4.7 gezeigt, besitzt die Entschlüsselung die gleiche Struktur wie die Verschlüsselung. Wir können also ebenfalls für die Entschlüsselung 4 neue S-Boxen $\mathbf{S_0}[a]\ldots\mathbf{S_3}[a]$ erzeugen:

$$\mathbf{S_0}[a]=\begin{bmatrix}\mathbf{S^{-1}}[a]\cdot 0E\\\mathbf{S^{-1}}[a]\cdot 0B\\\mathbf{S^{-1}}[a]\cdot 0D\\\mathbf{S^{-1}}[a]\cdot 09\end{bmatrix},\mathbf{S_1}[a]=\begin{bmatrix}\mathbf{S^{-1}}[a]\cdot 09\\\mathbf{S^{-1}}[a]\cdot 0E\\\mathbf{S^{-1}}[a]\cdot 0B\\\mathbf{S^{-1}}[a]\cdot 0D\end{bmatrix},$$

$$\mathbf{S_2}[a]=\begin{bmatrix}\mathbf{S^{-1}}[a]\cdot 0D\\\mathbf{S^{-1}}[a]\cdot 09\\\mathbf{S^{-1}}[a]\cdot 0E\\\mathbf{S^{-1}}[a]\cdot 0B\end{bmatrix},\mathbf{S_3}[a]=\begin{bmatrix}\mathbf{S^{-1}}[a]\cdot 0B\\\mathbf{S^{-1}}[a]\cdot 0D\\\mathbf{S^{-1}}[a]\cdot 09\\\mathbf{S^{-1}}[a]\cdot 0E\end{bmatrix}.$$

4 Bytes können also mit

$$\mathbf{S_0}[a_{0,j}]\oplus\mathbf{S_1}[a_{1,j-C1}]\oplus\mathbf{S_2}[a_{2,j-C2}]\oplus\mathbf{S_3}[a_{3,j-C3}]\oplus k_j$$

eine Runde entschlüsselt werden.

3.5. Der Keyschedule

Bei einer Ver- bzw. Entschlüsselung wird für jedes KeyAdd ein eigener Rundenschlüssel verwendet, so daß insgesamt $N_r + 1$ Rundenschlüssel [3] nötig sind. Jeder dieser Rundenschlüssel wird mittels des KeySchedules aus dem Chiffrierschlüssel K abgeleitet, d.h. der KeySchedule ist der "Schlüssel(fahr)plan" der angibt, wie jedes Byte des Chiffrierschlüssels K in den jeweiligen Rundenschlüssel einfließt.

Zunächst wird der sogenannte erweiterte Schlüssel W berechnet. Er besteht aus $N_b \cdot (N_r + 1)$ 32 Bit Worten $W_0 \ldots W_{(N_b \cdot (N_r+1)-1)}$, von denen die ersten N_k Worte mit den N_k Spalten des Chiffrierschlüssels K initialisiert werden. Die übrigen Worte werden sukzessive mittels Verknüpfung von 2 bereits berechneten Worten gebildet. Für $N_k \leq 6$ bzw. für $N_k > 6$ existieren 2 leicht unterschiedliche Berechnungsvorschriften.

Für $N_k \leq 6$ wird ein Wort W_i, dessen Position i ein Vielfaches von N_k ist, mit

$$W_i = \underbrace{\mathbf{SubByte}(\mathbf{RotByte}(W_{i-1})) \oplus \ \mathbf{Rcon}[i/N_k]}_{=:M} \oplus W_{i-N_k}$$

berechnet. **RotByte** rotiert die Bytes a, b, c, d eines 32 Bit Wortes (a, b, c, d) zyklisch nach links:

$$\mathbf{RotByte}(a, b, c, d) \to (b, c, d, a).$$

Die Funktion **SubByte** wendet auf jedes Byte a, b, c, d die S-Box an. **Rcon** ist ein Array von fest definierten 32 Bit Rundenkonstanten (vgl. Anhang A.3).

Die Berechnung eines Wortes W_i, dessen Position kein Vielfaches von N_k ist, erfolgt durch

$$W_i = W_{i-1} \oplus W_{i-N_k}$$

Die gesamte Schlüsselerweiterung in Pseudocode für $N_b \leq 6$ ist in Abb. 3.12 und für $N_b = 8$ in Abb. 3.13 dargestellt.

In Abbildung 3.14 ist der Beginn der Berechnung des erweiterten Schlüssels W skizziert. Dabei bezeichnet M die Modifikation $\mathbf{SubByte}(\mathbf{RotByte}(W_{i-1})) \oplus \mathbf{Rcon}[i/N_k]$.

Aus dem erweiterten Schlüssel W werden nun die Rundenschlüssel extrahiert, indem man einfach die ersten N_b Worte für Rundenschlüssel 0, die folgenden N_b Worte für Rundenschlüssel 1 usw. verwendet. Der Rundenschlüssel für Runde i besteht also aus den Worten $W_{N_b \cdot i}$ bis $W_{N_b \cdot (i+1)-1}$ (siehe Abb. 3.14).

Wie man dieser Spezifikation des KeySchedules entnehmen kann, ist es bei Kenntnis von N_k aufeinanderfolgenden Spalten des erweiterten Schlüssels W möglich, den KeySchedule rückwärts anzuwenden. Wir können also mit Kenntnis von genügend Rundenschlüsselbytes den Chiffrierschlüssel allein durch Umkehrung des KeySchedules bestimmen.

[3] zusätzlich zu jeder der N_r Runden wird vor der ersten Runde ein KeyAdd angewendet

```
for (i = 0; i < N_k; i++)
{
        W[i] = (K[4·i], K[4·i+1], K[4·i+2], K[4·i+3]);
}
for (i = N_k; i < N_b·(N_r+1); i++)
{
        temp = W[i−1]
        if(i ≡ 0 mod N_k)
            temp = Subbyte(RotByte(temp)) ⊕ Rcon[i/N_k];
        W[i] = W[i−N_k] ⊕ temp;
}
```

Abbildung 3.12.: Schlüsselerweiterung für $N_b \leq 6$ in Pseudocode

```
for (i = 0; i < N_k; i++)
{
        W[i] = (K[4·i], K[4·i+1], K[4·i+2], K[4·i+3]);
}
for (i = N_k; i < N_b·(N_r+1); i++)
{
        temp = W[i−1]
        if(i ≡ 0 mod N_k)
            temp = Subbyte(RotByte(temp)) ⊕ Rcon[i/N_k];
        else if (i ≡ 4 mod N_k)
            temp = SubByte(temp);
        W[i] = W[i−N_k] ⊕ temp;
}
```

Abbildung 3.13.: Schlüsselerweiterung für $N_b = 8$ in Pseudocode

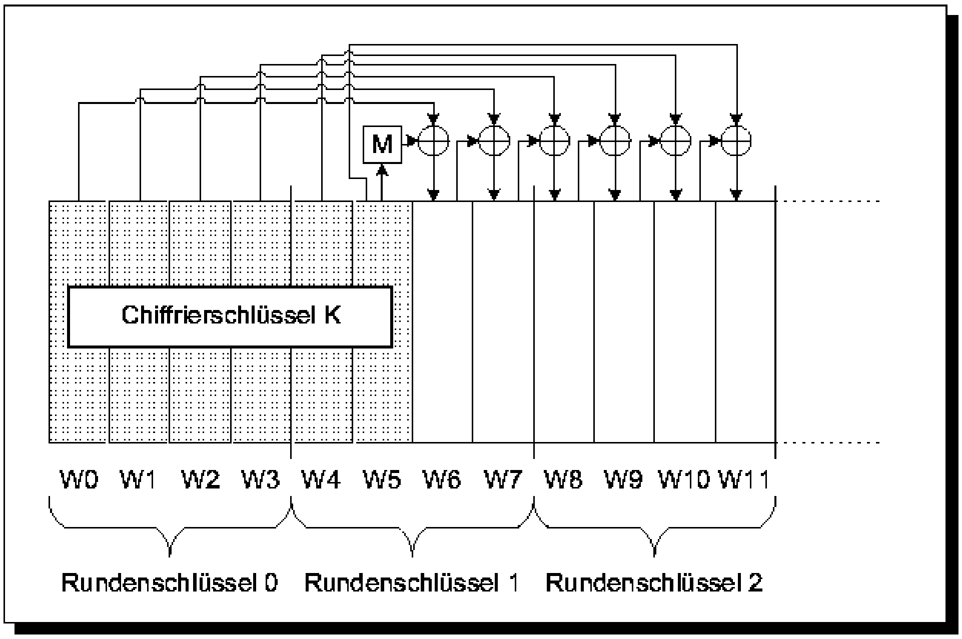

Abbildung 3.14.: Berechnung des erweiterten Schlüssels W

4. Kryptoanalyse von Rijndael

« Cipher Work will have little permanent attraction for one who expects results at once, without labor ... »

Col. Parker Hitt [1]

Die Kryptoanalyse, also die "unbefugte" Entzifferung verschlüsselter Nachrichten, ist so alt wie die Kryptographie selbst. Schon zu Beginn der Menschheitsgeschichte gab es die ersten Versuche, Nachrichten vertraulich zu übermitteln und im Gegenzug Bemühungen, diese Verschlüsselungen zu knacken.

Der Wettstreit zwischen Kryptographen d.h. Entwicklern von Verschlüsselungen und Kryptoanalytikern, die versuchen die Verschlüsselungssysteme zu brechen, wurde bis Mitte des 20. Jahrhunderts meistens über kurz oder lang von den Kryptoanalytikern gewonnen. Die Geschichte der Kryptographie ist voll von Beispielen "unknackbarer" Chiffren, die letztendlich doch von den Kryptoanalytikern gebrochen wurden. Ein berühmtes Beispiel war das Brechen der Enigma im zweiten Weltkrieg. In vielen Fällen beruhte die Sicherheitsanalyse allein auf der großen Anzahl von möglichen Schlüsseln, die eine vollständige Suche nach dem richtigen Schlüssel unmöglich machen sollte. Tatsächlich aber liefert die Größe des Schlüsselraums nur eine obere Schranke für die Sicherheit, wie Kerckhoffs bereits 1883 feststellte. Sie kann nur erreicht werden, wenn es dem Angreifer nicht gelingt, Schwächen im Verfahren selbst zu finden. Außer dem "perfekt sicheren" und sehr ineffizienten One Time Pad, gibt es bis heute keine Chiffre, die beweisbar sicher ist. Man kann lediglich die Aussage treffen, daß ein Verfahren resistent gegenüber den bisher bekannten Angriffen ist.

4.1. Grundlagen zur Kryptoanalyse und Begriffsbildung

Wir gehen von folgendem Szenario aus: Alice und Bob wollen über einen unsicheren Kanal kommunizieren. Zu diesem Zweck haben sie sich auf ein Verschlüsselungsverfahren geeinigt und einen geheimen Schlüssel K vereinbart. K wurde zufällig

[1] Manual for the Solution of Military Ciphers

gleichverteilt aus der Menge der möglichen Schlüssel $\mathcal{K}$ ausgewählt. Die Wahrscheinlichkeit einen bestimmten Schlüssel K zu wählen ist also $1/|\mathcal{K}|$. Wenn Alice nun Bob eine Nachricht P zusenden möchte, sendet sie stattdessen den Chiffretext $C = E_K(P)$. Bob kann durch Kenntnis von K den Klartext P aus C rekonstruieren.

Eve möchte die Kommunikation von Alice und Bob abhören. Sie ist in der Lage, die Chiffretexte mitzulesen und sie kennt alle Verschlüsselungsdetails wie z.B. das Verfahren, die verwendete Sprache usw. Das einzige, was Eve nicht kennt, ist der Schlüssel K (Kerckhoffs Prinzip).

Klassifikation der Angriffe Es gibt verschiedene Klassen von Angriffen, die ein Angreifer verwenden kann:

- **Known Ciphertext:** Der Angreifer kennt eine oder mehrere verschlüsselte Nachrichten, z.B. durch Abhören des Funkverkehrs.
- **Known Plaintext:** Der Angreifer kennt einen oder mehrere Chiffretexte und die dazugehörigen Klartexte.
- **Chosen Plaintext:** Der Angreifer kann beliebige Klartexte wählen und die dazu gehörigen Chiffretexte ausgeben lassen. Der Angreifer hat z.B. das Verschlüsselungsverfahren als eine Black-Box gegeben, die er mit den Eingaben (Klartexte) füttern kann und die ihm dann die entsprechenden Chiffretexte liefert. Andere Operationen, insbesondere das Ablesen von Zwischenergebnissen, sind nicht möglich.
- **Adaptive Chosen Plaintext:** Der Angreifer kann beliebige Klartexte in Abhängigkeit der zuvor berechneten Chiffretexte wählen.
- **Chosen Ciphertext:** Der Angreifer kann beliebige Chiffretexte wählen und die dazu gehörigen entschlüsselten Klartexte berechnen.

Theoretische Sicherheit Shannon hat in [Sha49] den Begriff der “perfekten Sicherheit” definiert. Ein Verfahren ist perfekt sicher, falls Eve durch das Abhören der Kommunikation keinerlei Wissen erhält [2].

Er wies nach, daß es perfekt sichere Verfahren gibt (z.B. One Time Pad), zeigte aber auch gleich den gravierenden Nachteil: Der Schlüssel muß mindestens so lang sein wie die Nachricht und darf nur einmal verwendet werden. Diese Bedingungen schränken den Einsatz auf Situationen allerhöchster Sicherheitsstufe ein. In der alltäglichen Praxis ist ein solches Verfahren zu aufwendig.

[2] außer vielleicht, daß eine Kommunikation stattgefunden hat

Praktische Sicherheit Der Begriff praktische Sicherheit bezeichnet die Resistenz eines Verfahrens gegenüber "praktisch möglichen Angriffen". Hier gibt es verschiedene Bedrohungen für eine Chiffre:

- Der Angreifer kann den geheimen Schlüssel K berechnen.
- Der Angreifer findet einen Algorithmus, der äquivalent zu $E_K()$ oder $D_K()$ ist.
- Der Angreifer ist in der Lage, zu einem Chiffretext den Klartext zu berechnen.
- Der Angreifer erhält Informationen über den Schlüssel, den Klartext oder Chiffretext, die er vor dem Angriff nicht besaß.
- Der Angreifer kann einen Chiffretext von einer zufälligen Permutation unterscheiden.

Im folgenden beschäftigen wir uns mit der Kryptoanalyse von Rijndael mit reduzierter Rundenzahl und einer Blocklänge von 128 Bit sowie einer Schlüssellänge von 128 Bit.

Zur besseren Orientierung führen wir Bezeichnungen für jeden Zustand ein, der im Verlauf der Ver- bzw. Entschlüsselung auftritt. Dabei bezeichnen wir mit B den Zustand nach einem ByteSub, mit R den Zustand nach einem ShiftRow, mit M den Zustand nach einem MixColumn und mit A den Zustand nach einem KeyAdd. Mit dem Index ^{r} spezifizieren wir die Runde r, in der der Zustand auftritt, so daß der Zustand damit eindeutig festgelegt ist. Z.B. ist M^4 der Zustand nach dem MixColumn der 4ten Runde. Zusätzlich bezeichnen wir den Startzustand mit P und den Endzustand mit C $(= A^{N_r})$.
Den Rundenschlüssel in Runde r bezeichnen wir mit K^r und mit $\widehat{K}^r$ bezeichnen wir einen äquivalenten Rundenschlüssel, der vor dem MixColumn eingebracht wird (vgl. 3.4.5). Ein Byte eines Zustands wird durch seine Koordinaten (i,j), $i \in \{0,\ldots,3\}$,$j \in \{0,\ldots,N_b-1\}$ eindeutig definiert, z.B. ist $A^5_{1,2}$ das Byte in Zeile 1 Spalte 2 des Zustands nach dem KeyAdd von Runde 5.

Als Kostenmaß für den zeitlichen Aufwand wählen wir die Zeit, die für eine Verschlüsselung benötigt wird. Den Aufwand an Speicherplatz geben wir in Blöcken an.

4.2. Der SQUARE-Angriff

Der SQUARE-Angriff ([DKR97], [DR99]) ist ein Chosen Plaintext Angriff, bei dem der Angreifer für 256 ausgewählte Klartexte die entsprechenden Chiffretexte kennen muß. Die Klartexte haben die Eigenschaft, daß 15 der 16 Bytes in allen Klartexten gleich sind. In dem restlichen Byte aber unterscheiden sich jeweils zwei

Klartexte aus dieser Menge. Wie wir zeigen werden, ergeben diese Texte nach 3 Runden Verschlüsselung die Summe 0. Dieses "Zwischenergebnis" wird dann in einem Algorithmus für eine effiziente Suche nach dem Schlüssel genutzt.

Wir beginnen mit der Beschreibung der Idee, die zu einem Angriff auf 4 Runden Rijndael führt, und erläutern dann, wie dieser Basis-Angriff auf bis zu 6 Runden erweitert werden kann.

4.2.1. Idee

Für den SQUARE-Angriff benötigen wir eine Menge von 256 Klartexten. Zunächst definieren wir einige Begriffe:

Definition 4.2.1 (aktives Byte) *Ein Byte an Position* (i, j) *heißt aktives Byte, falls sich in einer Menge* L *von* 256 *Zuständen* $Z^0, \ldots, Z^{255}$ *jeweils* 2 *Zustände in dem Byte an Position* (i, j) *unterscheiden.*

$$\forall Z^k, Z^l \in L, k \neq l : Z^k_{i,j} \neq Z^l_{i,j} \Leftrightarrow \text{ Byte } (i,j) \text{ist aktiv}$$

Ein aktives Byte nimmt also über einer Menge von 256 Zuständen jeden möglichen Wert genau einmal an.

Definition 4.2.2 (passives Byte) *Ein Byte an Position* (i, j) *heißt passives Byte, falls in einer Menge* L *von* 256 *Zuständen* $Z^0, \ldots, Z^{255}$ *jeder Zustand in dem Byte an Position* (i, j) *den gleichen Wert enthält.*

$$\forall Z^k, Z^l \in L : Z^k_{i,j} = Z^l_{i,j} \Leftrightarrow \text{ Byte } (i,j) \text{ist passiv}$$

Definition 4.2.3 (ausgeglichenes Byte) *Ein Byte an Position* (i, j) *heißt ausgeglichen, falls die Summe über einer Menge* L *von* 256 *Zuständen* $Z^0, \ldots, Z^{255}$ *an Position* (i, j) 0 *ergibt:*

$$\sum_{0 \leq k \leq 255} Z^k_{i,j} = 0$$

Lemma 4.2.1 *Jedes aktive Byte und jedes passive Byte ist ausgeglichen.*

Beweis.

- Das Byte an Position (i, j) ist passiv.
  Dann ist $\sum_{0 \leq l \leq 255} Z^l_{i,j}$ das **XOR** von 256 gleichen Werten. Da das **XOR** von jeweils zwei gleichen Werten 0 ergibt, so ist auch die Gesamtsumme 0.

- Das Byte an Position (i, j) ist aktiv.
  Dann ist $\sum_{0 \leq l \leq 255} Z^l_{i,j} = \sum_{0 \leq l \leq 255} l$ das **XOR** aller 256 Werte. Wir betrachten die Binärdarstellung aller möglichen Werte $(0 \ldots 255)$. Das **XOR** an jeder Bitposition enthält genau 128 Nullen und 128 Einsen. Also ergibt die Summe jedes Bits 0 und damit ist die Gesamtsumme 0.

$\square$

Definition 4.2.4 (Λ-Menge) *Eine Menge L von 256 Zuständen L^l, mit $l = 0, \ldots, 255$, deren Bytes entweder aktiv oder passiv sind, heißt Λ-Menge.*

Damit hat eine Λ-Menge insbesondere die Eigenschaft, daß alle Bytes ausgeglichen sind (Lemma 4.2.1).

Wie betrachten nun eine Λ-Menge $L = \{L^0, \ldots, L^{255}\}$, die genau ein aktives Byte hat und betrachten die Veränderungen, die die einzelnen Transformationen bewirken:

KeyAdd Da an einer Byte-Position ein festes Schlüsselbyte addiert wird, verändert ein KeyAdd die Eigenschaft, aktiv, passiv oder ausgeglichen zu sein, nicht.

ByteSub ByteSub ist eine bijektive Abbildung, d.h. 2 gleiche Bytes werden auf die gleichen Werte abgebildet, 2 unterschiedliche Bytes werden auf 2 unterschiedliche Werte abgebildet.
Im Falle eines passiven Bytes, wird in jedem der 256 Zustände lediglich ein konstanter Wert angenommen. ByteSub wirkt also auf alle Zustände an dieser Position in gleicher Weise und erzeugt damit wieder ein passives Byte lediglich mit einem veränderten Wert.
Im Falle eines aktiven Bytes, wird in jedem der 256 Zustände ein anderer Wert angenommen. D.h. ByteSub bildet jedes Byte auf einen anderen Wert ab. Da es aber nur 256 mögliche Werte gibt, werden nach einem ByteSub wieder alle Werte angenommen; das Byte bleibt also aktiv.
Ein ausgeglichenes Byte, das weder aktiv noch passiv ist, bleibt im allgemeinen nach einem ByteSub nicht ausgeglichen. Die Nichtlinearität der S-Box verhindert, daß sich diese Eigenschaft in der Transformation erhält.

ShiftRow Ein ShiftRow verändert nur die Positionen der Bytes, d.h. es wird gegebenfalls die Positionen der aktiven bzw. passiven Bytes verändert. Dies hat aber keinen Einfluß auf die Eigenschaften der Bytes.

MixColumn MixColumn bildet eine Linearkombination aus den 4 Bytes einer Spalte. Die MDS-Eigenschaft führt dazu, daß bei einer Eingabe von drei passiven und einem aktiven Byte sich jeweils zwei Texte in allen 4 Bytes unterscheiden müssen (siehe Lemma 3.4.1). Dies ist gleichbedeutend damit, daß jedes Byte

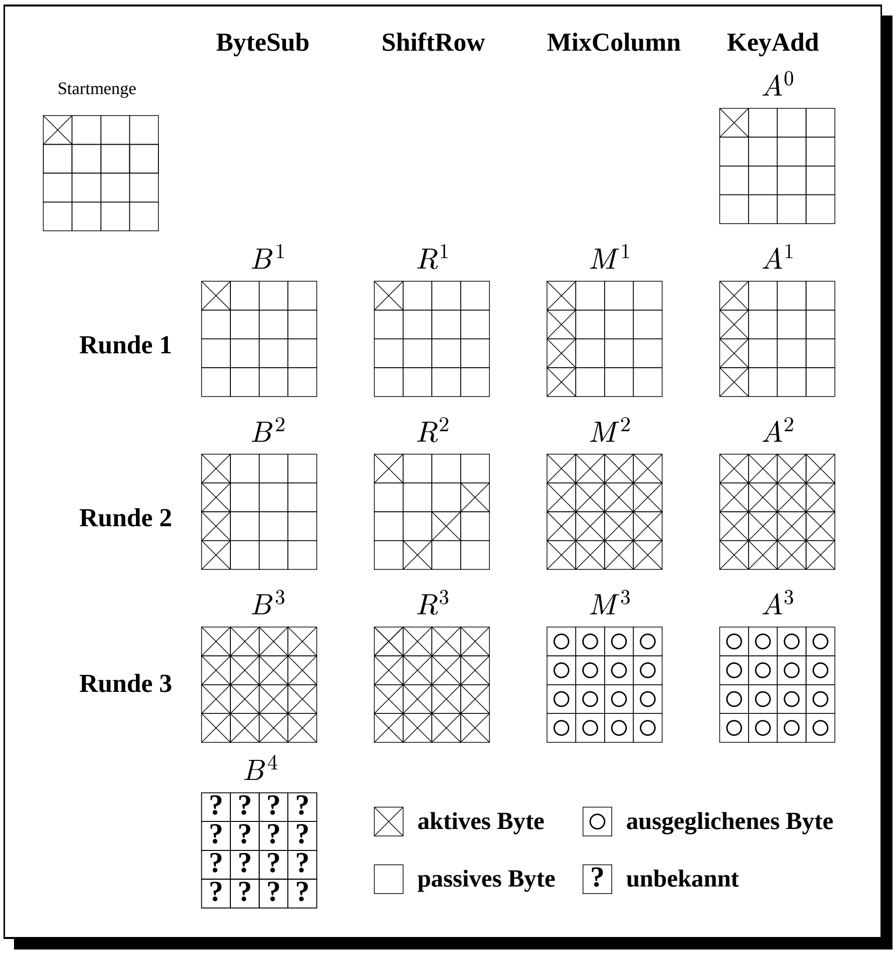

Abbildung 4.1.: Verlauf der Verschlüsselung einer Λ-Menge

der Spalte jeden Wert annehmen muß, also aktiv ist.
Gehen mehr als ein aktives Byte ein, so entstehen Bytes, die zwar noch ausgeglichen sind, aber weder aktiv noch passiv sind. L ist damit keine Λ-Menge mehr.
Bei Eingabe von 4 passiven Bytes, werden alle Bytes auf die gleiche Weise geändert, bleiben also passiv.

Der Verlauf, mit $P_{0,0}$ als aktivem Byte der Startmenge, ist in Abbildung 4.1 skizziert. Ein aktives Byte ist mit einem Kreuz markiert. Man erkennt, daß am Ende der zweiten Runde alle Bytes aktiv sind (A^2). Diese Eigenschaft wird auch von dem nachfolgenden ByteSub und ShiftRow erhalten (R^3). Das folgende MixColumn zerstört die Aktivheit aller Bytes einer Spalte, weil hier zum ersten Mal ein MixColumn auf mehr als ein aktives Byte angewendet wird. Die MDS-Eigenschaft läßt hier keine Aussage mehr über die Eigenschaft der Bytes zu. Da aber MixColumn eine lineare Transformation ist, wird die Summe der Bytes in M^3 nicht verändert (bleibt also 0). Allgemein gilt:

Lemma 4.2.2 *Bei Eingabe von ausschließlich ausgeglichenen Bytes (also z.B. aktiven oder passiven), erhält MixColumn die Eigenschaft der Ausgeglichenheit.*

Beweis. Sei $a \in L$ ein Zustand und seien $a_0 \dots a_3$ die ausgeglichenen Bytes einer Spalte. Es gilt also

$$\bigoplus_{a\in\Lambda} a_0 = \bigoplus_{a\in\Lambda} a_1 = \bigoplus_{a\in\Lambda} a_2 = \bigoplus_{a\in\Lambda} a_3 = 0$$

Da MixColumn eine lineare Transformation ist, muß die Eigenschaft der Ausgeglichenheit erhalten bleiben. Seien $b_0, \dots, b_3$ die entsprechenden Einträge aus der MDS-Matrix, so führt MixColumn zu einer Linearkombination der Form

$$\bigoplus_{a\in\Lambda} b_0 \cdot a_0 \oplus b_1 \cdot a_1 \oplus b_2 \cdot a_2 \oplus b_3 \cdot a_3 = b_0 \underbrace{\bigoplus_{a\in\Lambda} a_0}_{=0} \oplus b_1 \underbrace{\bigoplus_{a\in\Lambda} a_1}_{=0} \oplus b_2 \underbrace{\bigoplus_{a\in\Lambda} a_2}_{=0} \oplus b_3 \underbrace{\bigoplus_{a\in\Lambda} a_3}_{=0} = 0$$

□

Das darauffolgende KeyAdd erhält die Ausgeglichenheit (A^3) und erst das (nichtlineare) ByteSub der Runde 4 verändert die Summe (B^4).

Bei der Verschlüsselung einer Λ-Menge haben wir also einen "Ankerpunkt" am Ende der dritten Runde (A^3), an dem alle Bytes ausgeglichen sind. Diese Eigenschaft liefert uns einen Ansatzpunkt für einen effizienten Angriff.

4.2.2. Angriff auf 4 Runden (Basis-Angriff)

Wir betrachten in diesem Abschnitt ein auf 4 Runden reduziertes Rijndael (ein initiales KeyAdd, 3 volle Runden und eine verkürzte letzte Runde ohne MixColumn). Die Verschlüsselung einer Λ-Menge führt zu einem ausgeglichenen Zustand am Ende von Runde 3.

Ausgehend von den Chiffretexten hängt ein Byte $C_{i,j}$ genau von einem Byte [3] $A^3_{i',j'}$ des Zustands am Ende von Runde 3 und von einem Byte $K^4_{i,j}$ des vierten Rundenschlüssels ab:

$$C_{i,j} = \mathbf{S}[A^3_{i',j'}] \oplus K^4_{i,j} \tag{4.1}$$

[3] die Position wird eventuell vom ShiftRow verändert

Wir können also ein Byte des 4. Rundenschlüssel raten ($K^4_{i,j}$), bis zum Ende der 3. Runde ($A^3_{i',j'}$) zurückrechnen und dann testen, ob das Byte ausgeglichen ist. Wenn dies nicht der Fall ist, dann haben wir ein falsches Schlüsselbyte geraten. Falls wir aber Ausgeglichenheit feststellen, so haben wir wahrscheinlich das richtige Schlüsselbyte geraten.

Lemma 4.2.3 *Unter der Annahme, daß sich die Rückrechnung von A^4 nach A^3 wie eine zufällige Abbildung verhält, führt ein zufällig gleichverteilt gewähltes falsches Schlüsselbyte mit Wahrscheinlichkeit* $1/256$ *zur Ausgeglichenheit.*

Beweis. Die Summe der ersten 255 der insgesamt 256 Werte ist eine zufällig gleichverteilte Zwischensumme. Das Byte ist genau dann ausgeglichen, wenn das **XOR** mit dem letzten noch zu addierenden zufällig gleichverteilten Wert, 0 ergibt. Dieser Fall tritt genau mit Wahrscheinlichkeit $1/256$ auf. $\square$

Dies bedeutet, daß die erwartete Anzahl von Schlüsselbytes, die zur Ausgeglichenheit führen $1 + \frac{255}{256} \approx 2$ ist. Das richtige Schlüsselbyte führt mit Wahrscheinlichkeit 1 zur Ausgeglichenheit und die übrigen 255 falschen Schlüsselbytes gleichen jeweils mit Wahrscheinlichkeit $1/256$ aus, d.h. wir erwarten, daß jeweils das richtige und ein falsches Schlüsselbyte übrig bleiben.

Um das Schlüsselbyte eindeutig zu bestimmen, kann man entweder den Test mit einer weiteren Λ-Menge wiederholen, oder man testet am Ende alle Kombinationen möglicher Schlüsselkandidaten.

Aufwand Die erwartete Textmenge, die benötigt wird, beschränkt sich hier auf $2 \cdot 2^8$ Klartext-Chiffretextpaare, da wir erwarten nach 2 Λ-Mengen ein Schlüsselbyte eindeutig bestimmt zu haben.

Der zeitliche Aufwand wird bestimmt durch die 2^8 Schlüsselmöglichkeiten, die wir testen müssen. Der Einfachheit halber schätzen wir das Rückrechnen von 256 Texten grob mit einer Verschlüsselung ab. Wir erhalten also einen zeitlichen Aufwand von 2^8 Verschlüsselungen.

4.2.3. Erweiterung am Ende (4+1 Angriff)

Wie in Abb. 4.2 illustriert, kann der SQUARE-Angriff auf 5 Runden Rijndael ausgeweitet werden, indem wir einfach am Ende eine weitere Runde anhängen. Nun hängt ein Byte c_i der Chiffretexte von 4 Bytes des 5ten Rundenschlüssels K^5 und von einem Byte des 4ten Rundenschlüssels K^4 ab, weil nun die 4te Runde ein MixColumn enthält. Die Bytes, die im Verlauf der Verschlüsselung von $A^3_{0,0}$ abhängen, sind mit einem schwarzen Punkt markiert.

Wir können also zunächst die 4 Bytes $(k_0, \ldots, k_3)$ von K^5 raten (2^{32} Möglichkeiten) und damit die 5te Runde zurückrechnen (A^4). Die Positionen der 4 Bytes

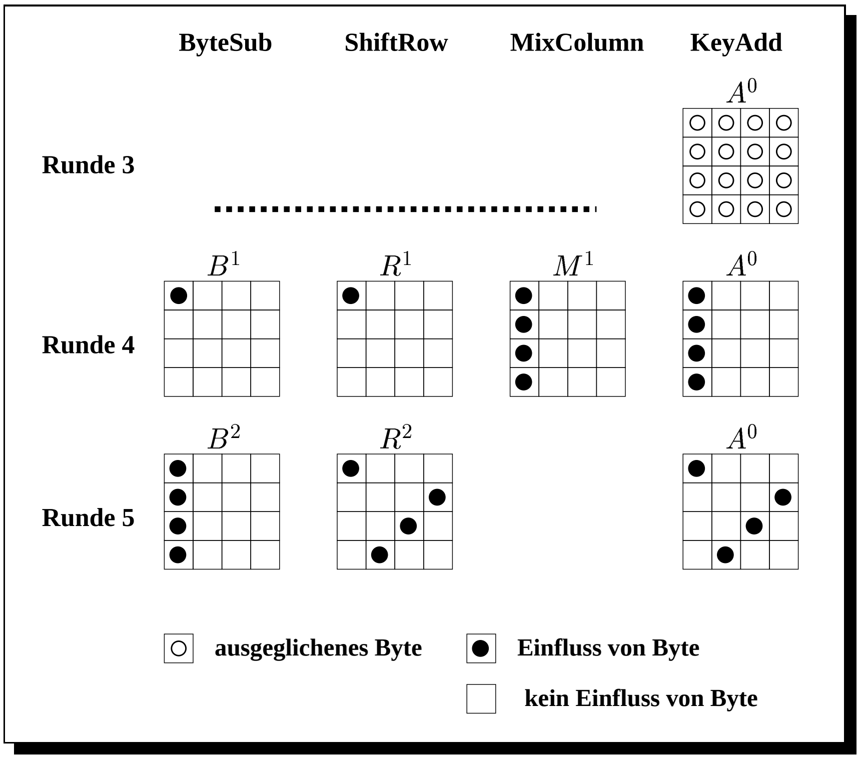

Abbildung 4.2.: Einfluß des Bytes $A^3_{0,0}$ im Verlauf der Verschlüsselung

müssen so gewählt sein, daß sie nach dem ShiftRow in B^5 eine Spalte ergeben.
Im normalen Verlauf müssten wir jetzt wieder 4 Bytes von K^4 raten, um im nächsten Schritt das MixColumn zu invertieren und R_4 zu berechnen. Wie aber bereits in Abschnitt 3.4.5 über den Aufbau einer Runde beschrieben, ist es möglich, die Reihenfolge von MixColumn und KeyAdd zu vertauschen (siehe Abb. 4.3), so daß wir nur ein Schlüsselbyte k_4 raten müssen.

So können wir zunächst das MixColumn invertieren und dann durch Raten eines Schlüsselbytes von $\widehat{K}^4$ ein Byte $R^4_{i,j}$ bestimmen (vgl. Abb 4.3). Die Invertierung von ShiftRow und ByteSub liefert endlich das Byte $A^3_{i,j}$ und wir können prüfen, ob Ausgeglichenheit vorliegt. Betrachten wir die effiziente Implementierung von Rijndael, so lösen wir die Gleichung

$$A^3_{i,j} = \mathbf{S^{-1}}[\mathbf{S_0}[c_0 \oplus k_0] \oplus \mathbf{S_1}[c_1 \oplus k_1] \oplus \mathbf{S_2}[c_2 \oplus k_2] \oplus \mathbf{S_3}[c_3 \oplus k_3] \oplus k_4] \qquad (4.2)$$

Falls das Byte nicht ausgeglichen ist, so sind wir sicher, daß die Kombination der geratenen Schlüsselbytes von K^5 und $\widehat{K}^4$ falsch war.

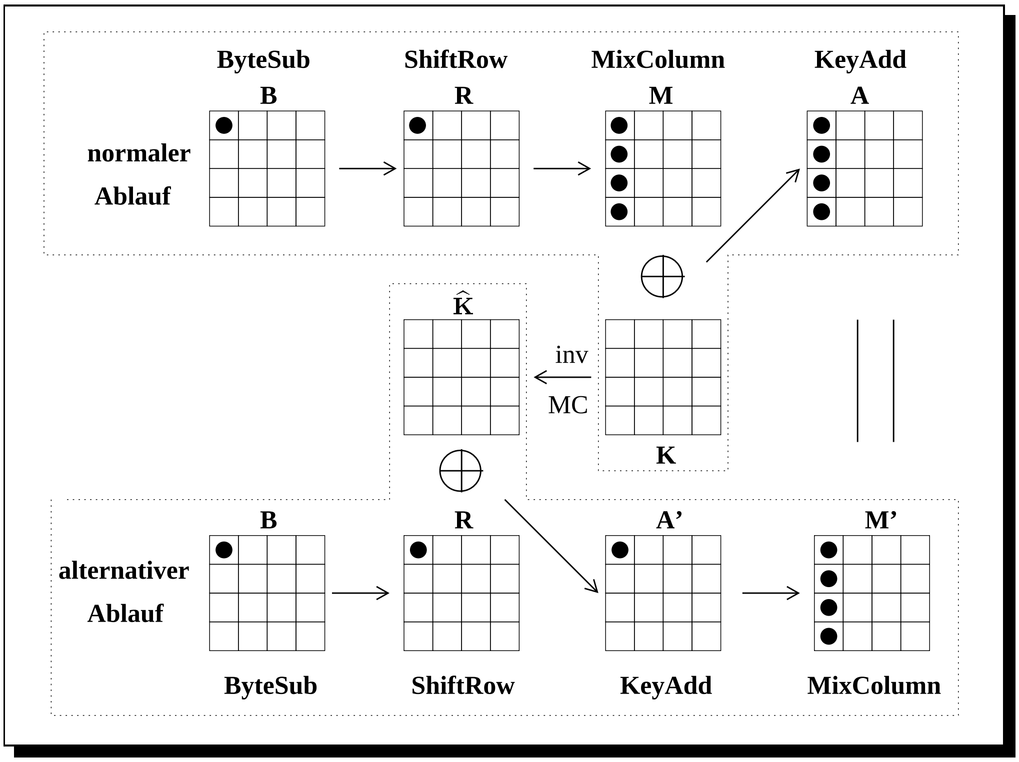

Abbildung 4.3.: Alternativer Ablauf einer Runde

Falls das Byte jedoch ausgeglichen ist, so sind die geratenen Schlüsselbytes eine mögliche Kombination. Wir haben für die 4 geratenen Schlüsselbytes aus K^5 ein Schlüsselbyte aus $\widehat{K}^4$ gefunden, das Ausgeglichenheit erzeugt. Wie in Lemma 4.2.3 gezeigt, erzeugt ein zufällig gleichverteilt gewähltes Byte von $\widehat{K}^4$ mit Wahrscheinlichkeit 1/256 Ausgeglichenheit, so daß nach einem Durchlauf die erwartete Anzahl von möglichen Kombinationen der Schlüsselbytes $\frac{1}{256} \cdot 2^{40} = 2^{32}$ beträgt. Pro Λ-Menge werden die Möglichkeiten für den Schlüssel um den Faktor 256 verringert, so daß wir erwarten, nach Auswertung von 5 Λ-Mengen einen Teil des Schlüssels, nämlich 4 Bytes von K^5 und 1 Byte von $\widehat{K}^4$ eindeutig berechnet zu haben.

Aufwand Der erwartete Bedarf an Textmaterial ist auf $5 \cdot 2^8$ Klartext-Chiffretextpaare gestiegen, da wir erwarten, nach Auswertung von 5 Λ-Mengen eine eindeutige Information über den Schlüssel zu erhalten.

Wir raten 5 Schlüsselbytes und müssen daher für jede der 2^{40} Möglichkeiten Gleichung 4.2 mit je 2^8 Texten berechnen. Wir approximieren die Zeit für das Testen einer Λ-Menge mit einer Schlüsselkombination mit der Zeit, die wir für eine

Verschlüsselung eines Textblocks benötigen und erhalten also einen Aufwand von ungefähr 2^{40} Verschlüsselungen.

4.2.4. Erweiterung am Anfang (1+4 Angriff)

Eine weitere Möglichkeit den Basis SQUARE-Angriff mit 4 Runden zu erweitern, ist es, eine Runde vorzuschalten. Das heißt, wir verschlüsseln keine Λ-Menge, sondern eine Menge von 256 Texten, die nach einer Runde (A^1) zu einer Λ-Menge wird. Wenn wir eine solche Textmenge gefunden haben, können wir nach einer 5 Runden Verschlüsselung den Basis SQUARE-Angriff verwenden. Wir haben also den Ankerpunkt, der am Ende der dritten Runde lag, auf das Ende der vierten Runde verschoben.

Eine Möglichkeit, eine solche Textmenge zu finden, ist, daß wir zunächst eine Λ-Menge wählen. Diese betrachten wir als Zwischenergebnis M^1 und verfolgen die Spalte mit dem aktiven Byte beim Entschlüsseln. D.h. wir invertieren das MixColumn, ShiftRow und ByteSub und erhalten A^0. Nun raten wir die 4 relevanten Schlüsselbytes von K^0 und erhalten 4 mögliche Klartextbytes. Die restlichen 12 Bytes des Zustands füllen wir mit konstanten Werten und erhalten so eine Textmenge, die am Ende der ersten Runde eine Λ-Menge ergibt.

Eine andere Möglichkeit, eine geeignete Textmenge zu finden ist die folgende: Wir nehmen eine Menge von 2^{32} Texten, die in einer Spalte von R^1 alle möglichen Werte annehmen und in den restlichen Bytes konstant sind. Diese Auswahl der Texte ist unabhängig von den entsprechenden Bytes von K^0, weil ein KeyAdd die Eigenschaft alle Werte anzunehmen, nicht zerstört. Nun raten wir die 4 relevanten Schlüsselbytes $(k_0, \ldots k_3)$ von K^0 und berechnen mit Hilfe der entsprechenden Klartextbytes $p_0, \ldots, p_3$ die Spalte in M^1 mittels

$$\mathbf{T_0}[p_0 \oplus k_0] \oplus \mathbf{T_1}[p_1 \oplus k_1] \oplus \mathbf{T_2}[p_2 \oplus k_2] \oplus \mathbf{T_3}[p_3 \oplus k_3] \tag{4.3}$$

Zu beachten ist hier, daß MixColumn auf alle 2^{32} möglichen Spalten angewendet wird. Da MixColumn eine bijektive Abbildung ist, werden in der Spalte wieder alle 2^{32} Möglichkeiten angenommen. Diejenigen Mengen von Texten, die in einer Spalte von M^1 alle Werte annehmen und in den restlichen Bytes konstant sind, fassen wir als Menge von 2^{24} Λ-Mengen auf. Aus den 2^{32} Möglichkeiten suchen wir jetzt 256 heraus, die in einem Byte alle Werte annehmen (aktives Byte) und in den restlichen Bytes konstant sind (passive Bytes).

Diese Möglichkeit werden wir in späteren Angriffen verwenden, da hier der Aufwand für das Testen eines Schlüsselkandidaten geringer ist.

Nachdem wir eine geeignete Textmenge ermittelt haben, wenden wir den Basis-Angriff an. Falls wir zu Beginn eine falsche Kombination der K^0 Bytes raten, so wird beim Verschlüsseln in M^1 keine Λ-Menge erzeugt und ein zufällig gleichverteilt gewähltes K^5 Byte führt wieder mit Wahrscheinlichkeit $1/256$ zur Ausgeglichenheit. Wir erwarten also wie beim $4 + 1$ Angriff nach Auswertung von 5 Λ-Mengen eine eindeutige Schlüsselinformation zu erhalten.

Aufwand Die benötigte Menge an Klartext-Chiffretextpaaren beträgt 2^{32}. Der zeitliche Aufwand wird wieder von der Anzahl der zu ratenden Schlüsselbytes bestimmt und beträgt wie beim $4 + 1$-Angriff 2^{40} Verschlüsselungen.

4.2.5. Angriff auf 6 Runden

Ausgehend von dem Basis-SQUARE-Angriff für 4 Runden, bei dem man 1 Schlüsselbyte raten muß, kann man beide Erweiterungen gleichzeitig anbringen. Man kann eine Runde vorschalten und muß dafür 4 Schlüsselbytes von K^0 raten; man kann auch eine Runde anhängen, wobei man dann zusätzlich 4 Schlüsselbytes von K^6 raten muß.

Aufwand Der Aufwand an Textpaaren wird bestimmt durch das Maximum der Aufwände der beiden Erweiterungen, also 2^{32} Paare.

Da wir in dem 6-Runden Angriff 9 Schlüsselbytes raten, müssen wir für 2^{72} mögliche Kombinationen jeweils 256 Texte auswerten. Schätzen wir die Auswertung von 256 Texten mit der Zeit für eine Verschlüsselung ab, so erhalten wir einen zeitlichen Aufwand von 2^{72} Verschlüsselungen.

4.2.6. Umgedrehter Angriff

Zu Beginn des Kapitels über den SQUARE-Angriff sind wir davon ausgegangen, daß wir beliebige Klartexte verschlüsseln können (chosen Plaintext Angriff). Der SQUARE-Angriff funktioniert aber auch als chosen Ciphertext Angriff, d.h. wir sind in der Lage, beliebige Chiffretexte zu entschlüsseln.

Dies kann man sich leicht überlegen:
Damit der Angriff 1:1 übertragbar ist, müssen zwei Eigenschaften gelten:
1. Rijndael verwendet zum Verschlüsseln die gleiche Struktur wie zum Entschlüsseln.
2. Die Eigenschaften der Transformationen insbesondere von ByteSub und MixColumn müssen auch für ihre Inversen gelten.
Teil 1 wurde in 3.4.7 gezeigt. Rijndael verwendet zum Verschlüsseln den gleichen Ablauf der Transformationen wie zum Entschlüsseln, es werden lediglich die inversen Transformationen verwendet. Für den zweiten Teil müssen wir die Eigenschaften der Nichtlinearität des inversen ByteSub und die MDS-Eigenschaft des inversen MixColumn zeigen. Das inverse ByteSub, führt zunächst eine feste affine Abbildung und dann eine Invertierung durch, die eine nichtlineare Abbildung liefert. Daß das

inverse MixColumn ebenfalls die MDS-Eigenschaft hat, haben wir schon in 3.4.3 erwähnt. Der Beweis ist analog zum Beweis zu Lemma 3.4.1.

Wir können für den SQUARE-Angriff also auch eine Λ-Menge von Chiffretexten wählen, diese entschlüsseln lassen und den Angriff "umgekehrt" starten. Diese Form des Angriffs als Chosen Ciphertext, liefert zunächst keinerlei Verbesserung gegenüber der Chosen Plaintext Variante. Wie wir aber in Abschnitt 4.5 zeigen werden, können wir den SQUARE-Angriff bei leichter Modifikation des Verschlüsselungsverfahrens verbessern.

4.3. Eine Verbesserung mittels Teilsummen

Ende 1999 verbesserte das "extended Twofish Team" bestehend aus Niels Ferguson, John Kelsey, Stefan Lucks, Bruce Schneier, Mike Stay, David Wagner und Doug Whiting den oben beschriebenen SQUARE-Angriff für 6 Runden erheblich [FKL+00]. Es gelang ihnen den Zeitaufwand von 2^{72} auf 2^{44} zu reduzieren. Dabei nutzten sie die Struktur des Angriffs geschickt aus, um die Menge von Zwischenergebnissen möglichst klein zu halten.

4.3.1. Idee

Zunächst einmal kann man den Aufwand durch eine simple Veränderung um den Faktor 2^8 reduzieren. Es besteht nämlich die Möglichkeit, zu Beginn des Angriffs nicht unbedingt eine Λ-Menge aus den 2^{32} Texten zu extrahieren, sondern man kann den Angriff auch mit der kompletten Textmenge durchführen.
Dies kann man sich leicht überlegen: Wir haben die 2^{32} Texte so gewählt, daß die 4 Bytes einer Spalte von A^1 genau alle Werte annehmen. Dies ist aber nichts anderes, als ob wir eine Menge von 2^{24} Λ-Mengen mit je einem aktiven Byte erzeugen, deren aktive Bytes genau in der Spalte liegen. Wir sind zwar nicht in der Lage, die Λ-Mengen exakt anzugeben, wir wissen aber, daß wir eine Menge von Λ-Mengen erzeugen. Jede Λ-Menge für sich betrachtet durchläuft also den SQUARE-Angriff und liefert in A^4 ausgeglichene Bytes. Da aber die Summe in einer Λ-Menge 0 ergibt, so ergibt auch die Gesamtsumme über alle Λ-Mengen 0. Diese kleine Modifikation erspart uns einen Aufwand von 2^{32}, weil wir die 4 Schlüsselbytes von K^0 nicht mehr raten müssen. Wir reduzieren also die zu testenden Schlüsselbytes durch die Eingabe von mehr Textpaaren. Die Erhöhung der Textmenge führt aber zu einem zusätzlichen Aufwand von 2^{24}, so daß im Endeffekt ein Gewinn von 2^8 übrig bleibt.

Die Verwendung aller 2^{32} Textpaare macht aber auch eine geschickte Berechnung von Zwischenergebnissen möglich.

Dabei wird die Eigenschaft der **XOR** Operation ausgenutzt, daß sich doppelt vorkommende Werte rauskürzen. D.h. es kommen nur Werte in die Endsumme die in ungerader Anzahl auftreten. Wenn wir die Zwischenergebnisse des Angriffs stufenweise berechnen, so können wir bereits nach jeder Stufe gleiche Zwischenergebnisse löschen.

4.3.2. Detaillierte Beschreibung

Grundlage für die Verbesserung ist die effiziente Implementierung einer Runde Rijndael aus Abschnitt 3.4.8, aus dem wir auch die Notation der S-Boxen verwenden.
Von den 2^{32} Chiffretexten verwenden wir die jeweils 4 relevanten Bytes $c_0, \ldots, c_3$, raten die entsprechenden Schlüsselbytes $k_0, \ldots, k_3$ von K^6 und k_4 von K^5 und berechnen damit

$$\bigoplus \underbrace{\mathbf{S^{-1}}[\underbrace{\underbrace{\underbrace{\mathbf{S_0}[c_0 \oplus k_0] \oplus \mathbf{S_1}[c_1 \oplus k_1]}_{\text{Stufe1}} \oplus \mathbf{S_2}[c_2 \oplus k_2]}_{\text{Stufe2}} \oplus \mathbf{S_3}[c_3 \oplus k_3]}_{\text{Stufe3}} \oplus k_4]}_{\text{Stufe4}} \tag{4.4}$$

über alle 2^{32} Texte.

In Stufe 1 berechnen wir aus den 4-Tupeln (c_0, c_1, c_2, c_3) die Tripel (z_1, c_2, c_3) mit der Teilsumme

$$z_1 = \mathbf{S_0}[c_0 \oplus k_0] \oplus \mathbf{S_1}[c_1 \oplus k_1]$$

Von diesen Tripeln kann es aber nur 2^{24} verschiedene geben. Jeweils zwei gleiche Tripel führen aber zu demselben Ergebnis in Stufe 4, bilden also dieselben Werte in der **XOR** -Operation über alle Texte. Da sich aber beim **XOR** jeweils zwei gleiche Werte zu 0 summieren, gehen letztendlich nur verschiedene Tripel in die Summe ein. Wir müssen uns also zu jedem möglichen Tripel nur die wirklich vorkommende Anzahl modulo 2 bestimmen. Ist die Anzahl 0, so kam das Tripel in gerader Zahl oder gar nicht vor und hat somit keine Auswirkung; ist die Anzahl dagegen 1, so geht das Tripel einmal in die Summe ein.

In Stufe 2 berechnen wir dann aus den Tripeln (z_1, c_2, c_3) die Tupel (z_2, c_3) mit

$$z_2 = z_1 \oplus \mathbf{S_2}[c_2 \oplus k_2]$$

Die Anzahl der verschiedenen Tupel ist auf 2^{16} begrenzt. Auch hier kürzen sich jeweils zwei gleiche Tupel heraus, so daß nur jedes Tupel maximal einmal in die Summe eingeht.

In Stufe 3 werden schließlich aus den Tupeln (z_2, c_3) die Werte z_3 mit

$$z_3 = z_2 \oplus \mathbf{S_3}[c_3 \oplus k_3]$$

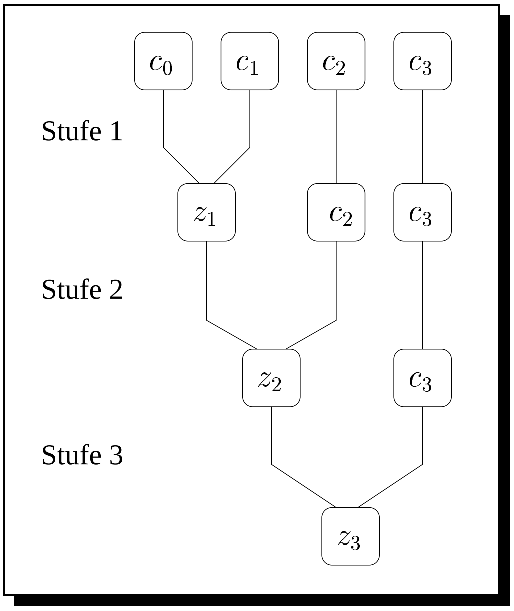

Abbildung 4.4.: Berechnung der Teilsummen

berechnet, von denen schließlich auch nur die Werte in die Summe einfließen, die in ungerader Anzahl auftreten. In Abb. 4.4 ist der Ablauf bis hierhin schematisch dargestellt.

In Stufe 4 wird schließlich

$$z_4 = \mathbf{S^{-1}}[z_3 \oplus k_4]$$

berechnet und das **XOR** über alle z_3 Werte gebildet. Ist diese Summe 0, so haben wir eine mögliche Schlüsselbytekombination gefunden, andernfalls war die Schlüsselbytekombination definitiv falsch.

4.3.3. Aufwand

Kostenmaß In diesem Kapitel betrachten wir vornehmlich die effiziente Implementierung von Rijndael aus 3.4.8, in der nur Table-Lookups und Verknüpfungen per **XOR** vorkommen. Wir geben den Zeitaufwand in Anzahl notwendiger Operationen (**Op**) an, wobei 1 **Op** $\widehat{=}$ 1 **XOR** und 6 **Op** $\widehat{=}$ 1 Table-Lookup. Diese Abschätzung wird ebenfalls in [Oua97] verwendet.
Zur Berechnung des Aufwands zählen wir alle Table Lookups, also S-Box Lookups und Zugriffe auf Elemente eines Zustands bzw. das Speichern eines Ergebnisses in einem Zustand, und alle XOR-Operationen. Wir ermitteln den Aufwand, um eine

Spalte eine Runde zu verschlüsseln anhand der Gleichung

$$e_j = \mathbf{T_0}[a_{0,j}] \oplus \mathbf{T_1}[a_{1,j+C1}] \oplus \mathbf{T_2}[a_{2,j+C2}] \oplus \mathbf{T_3}[a_{3,j+C3}] \oplus k_j.$$

Dazu benötigen wir einen Table Lookup, um auf die Elemente des Zustands zuzugreifen und 4 Table Lookups, um die S-Box Auswertungen auszuführen. Weiterhin benötigen wir einen Table Lookup für den Zugriff auf den Rundenschlüssel und einen Table Lookup für das Speichern des Ergebnisses. Zusammen mit den 4 **XOR** Operationen ergibt sich so ein Aufwand von 46 **Op**. Um den gesamten Zustand eine Runde zu verschlüsseln, benötigen wir die 4fache Anzahl Operationen, nämlich 184 **Op**. Für volle 6 Runden Verschlüsselung erhalten wir $6 \cdot 184 = 1104$ **Op**

Berechnung des Aufwands Der Zeitaufwand dieses Angriffs hängt von zwei Grössen ab: zum einen von der Anzahl von Schlüsselkombinationen, die in eine Stufe einfließen und zum anderen von der Größe der Menge der Teilsummen. Erstere gibt an, wie oft eine Stufe durchlaufen wird und letztere wieviele Operationen in dieser Stufe ausgeführt werden müssen. In Tabelle 4.1 sind die einzelnen Stufen mit den jeweiligen Angaben aufgelistet.

Stufe	# Texte	# Schlüssel	Gesamtaufwand
1	2^{32}	2^{16}	2^{48}
2	2^{24}	2^{24}	2^{48}
3	2^{16}	2^{32}	2^{48}
4	2^{8}	2^{40}	2^{48}

Tabelle 4.1.: Aufwand des SQUARE-Angriffs mit Teilsummen

Wir müssen Gleichung 4.4 also 2^{48} mal berechnen. Für eine Berechnung der Gleichung benötigen wir in unserem Kostenmaß 93 **Op**. (14 Table Lookups = 84 **Op** für Zugriffe auf Elemente und S-Box Anwendungen und 9 **XOR** Verknüpfungen.) Die Gesamtanzahl an Operationen beträgt also $2^{48} \cdot 93 \approx 2^{54.5}$ **Op**. Umgerechnet in Verschlüsselungszeiten erhalten wir einen zeitlichen Aufwand, der ca. $2^{44.5}$ Verschlüsselungen entspricht.

Unter der Annahme, daß die Chiffretexte zufällig gleichverteilt sind, ist die Wahrscheinlichkeit für ein n-Tupel mit gerader Anzahl vorzukommen gleich 1/2. Wir erwarten also, daß in jeder Stufe nur die Hälfte aller möglichen $n-$Tupel vorkommen. Dies gilt insbesondere für die 4 relevanten Bytes der Chiffretexte als Eingabe für Stufe 1. Dieses Verhalten konnten wir experimentell bestätigen. In Tabelle 4.2 ist die erwartete Anzahl von Texten in jeder Stufe aufgelistet. Daraus ergibt sich ein erwarteter Aufwand von $1/2 \cdot 2^{44.5} = 2^{43.5}$ Verschlüsselungen.

Stufe	max. # Texte	erw. # Texte
1	2^{32}	2^{31}
2	2^{24}	2^{23}
3	2^{16}	2^{15}
4	2^{8}	2^{7}

Tabelle 4.2.: Erwartete Anzahl n-Tupel in jeder Stufe

4.3.4. Details der Implementierung

In diesem Abschnitt betrachten wir den SQUARE-Angriff mit Teilsummen unter dem Aspekt einer effizienten Implementierung. Ziel ist es herauszufinden, inwieweit der Angriff praktisch durchführbar ist. Hierbei spielt nicht nur die Aufwandsabschätzung von $2^{44.5}$ Verschlüsselungen eine Rolle, sondern z.B. auch der Aufwand für die Verwaltung der Texte.

Zunächst müssen wir uns überlegen, in welcher Datenstruktur wir die jeweils 4 relevanten Bytes der 2^{32} Chiffretexte ablegen. Eine direkte Speicherung der 4-Tupel in einer Liste würde einen Platz von 16 Gigabyte benötigen und damit einen laufenden Zugriff auf die Festplatte erfordern. Ein Vergleich mit Zeiten für $2^{44.5}$ Verschlüsselungen wäre damit nicht möglich, weil die meiste Zeit für Festplattenzugriffe benötigt würde.

Wir haben uns deshalb für eine andere Möglichkeit entschieden. Wir interpretieren jedes 4-Tupel als Zahl a zwischen 0 und $2^{32}-1$, so daß ein 4-Tupel durch ein Bit an Position a in einem Bit-Array der Größe 2^{32} dargestellt werden kann. Diese Vorgehensweise hat 2 Vorteile: Zum ersten reduziert sich der benötigte Speicherplatz für die Chiffretexte auf 2^{32} Bits, also 512 Megabyte. Wir können die Textmenge also im Hauptspeicher ablegen. Zweitens benötigen wir zu jedem 4-Tupel nur die Information, ob es in gerader oder ungerader Anzahl vorkommt. 4-Tupel die in gerader Anzahl vorkommen haben keinen Einfluß auf die Berechnungen, weil sie sich in der **XOR** Operation am Ende herauskürzen. Wir flippen also bei jedem Vorkommen eines 4-Tupels das korrespondierende Bit.

Der Nachteil dieser Form der Speicherung besteht darin, daß wir für die weitere Verarbeitung der 4-Tupel jedes der 2^{32} Bits testen müssen und, falls es gesetzt ist, das 4-Tupel aus der Position des Bits rekonstruieren müssen. Heutige Prozessoren, wie der hier verwendete Pentium III, sind aber auf 32 Bit Wortzugriffe optimiert, nicht auf Zugriffe auf einzelne Bits oder Bytes, so daß wir hier die Prozessorgeschwindigkeit nicht optimal nutzen können. Hinzu kommt, daß wir erwarten, daß nur die Hälfte der Bits gesetzt ist, wir aber trotzdem jedes Bit testen müssen.

Die Ausgaben von Stufe 1, also die 2^{24} Tripel (z_1, c_2, c_3), legen wir nach dem gleichen Prinzip in einem Byte-Array der Größe 64 Megabyte ab. So sparen wir im weiteren Verlauf zumindest die Operationen, um auf einzelne Bits zuzugreifen. Diese Struktur verwenden wir auch in den übrigen Stufen. Tabelle 4.3 zeigt jeweils die Eingabe- und Ausgabegrößen der einzelnen Stufen. Abb. 4.5 zeigt die

Stufe	Eingabe	Ausgabe
1	512 MB	16 MB
2	16 MB	64 KB
3	64 KB	256 B
4	256 B	1 B

Tabelle 4.3.: Ein- und Ausgabegrößen der einzelnen Stufen

Implementierung des Angriffs in Pseudo-Code. Für die konkrete Implementierung wählen wir die Programmiersprache C (siehe Anhang B), weil sie den Komfort einer Hochsprache mit schnellem, optimiertem Code verbindet. Unser Testsystem ist ein Pentium III mit 933 MHz, 2 Gigabyte Hauptspeicher und Linux (Kernel 2.4.6).

Für die Zeitmessung haben wir jede Stufe mit der jeweils vollen Textmenge, aber nur mit einer geeigneten Anzahl Schlüssel gestartet und die resultierenden Zeiten entsprechend hochgerechnet. In Tabelle 4.4 haben wir den Zeitaufwand in Tagen aufgelistet, der in jeder Stufe benötigt wird. Wie man sieht, ist der zeitliche

Stufe	#Tupel	# Schlüssel	benötigte Zeit
1	2^{32} (512 MB)	2^{16}	166 Tage
2	2^{24} (16 MB)	2^{24}	160 Tage
3	2^{16} (64 KB)	2^{32}	70 Tage
4	2^{8} (256 B)	2^{40}	40 Tage
Gesamt			436 Tage

Tabelle 4.4.: Zeiten für SQUARE-Angriff mit Teilsummen

Aufwand für die einzelnen Stufen recht unterschiedlich. Stufe 1 benötigt ein wenig länger als Stufe 2, weil die Berechnung der Teilsumme z_1 einige Operationen mehr erfordert. Der Geschwindigkeitszuwachs in Stufe 3 und 4, ist durch die immer kleiner werdende Textmenge zu erklären. Der hier verwendete Pentium III Prozessor verfügt über einen 512 KB großen Second Level Cache und einen 32 KB großen

$$
\begin{array}{l}
\textbf{for } (k_0 = 0; k_0 < 256; k_0 ++) \\
\{ \\
\quad \textbf{for } (k_1 = 0; k_1 < 256; k_1 ++) \\
\quad \{ \\
\quad\quad \forall 2^{32} \text{ 4-Tupel } (c_0, c_1, c_2, c_3)\text{: } z_1 = \mathbf{S_0}[c_0 \oplus k_0] \oplus \mathbf{S_1}[c_1 \oplus k_1]; \\
\quad\quad \textbf{for } (k_2 = 0; k_2 < 256; k_2 ++) \\
\quad\quad \{ \\
\quad\quad\quad \forall 2^{24} \text{ Tripel } (z_1, c_2, c_3)\text{: } z_2 = z_1 \oplus \mathbf{S_2}[c_2 \oplus k_2]; \\
\quad\quad\quad \textbf{for } (k_3 = 0; k_3 < 256; k_3 ++) \\
\quad\quad\quad \{ \\
\quad\quad\quad\quad \forall 2^{16} \text{ Tupel } (z_2, c_3)\text{: } z_3 = z_2 \oplus \mathbf{S_3}[c_3 \oplus k_3]; \\
\quad\quad\quad\quad \textbf{for } (k_4 = 0; k_4 < 256; k_4 ++) \\
\quad\quad\quad\quad \{ \\
\quad\quad\quad\quad\quad \forall 2^{8} \ (z_3) : z_4 = z_4 \oplus \mathbf{S^{-1}}[z_3 \oplus k_4]; \\
\quad\quad\quad\quad\quad \textbf{falls } z_4 == 0 \\
\quad\quad\quad\quad\quad \{ \\
\quad\quad\quad\quad\quad\quad (k_0, \ldots, k_4) \text{ ist möglicher Kandidat} \\
\quad\quad\quad\quad\quad \} \\
\quad\quad\quad\quad \} \\
\quad\quad\quad \} \\
\quad\quad \} \\
\quad \} \\
\}
\end{array}
$$

Abbildung 4.5.: Square Angriff mit Teilsummen im Pseudo-Code

First Level Cache, die einen schnelleren Zugriff auf Daten erlauben als der Hauptspeicher. Da die Textmenge in Stufe 3 in den Second Level Cache und in Stufe 4 in den First Level Cache paßt, können wir hier eine deutliche Geschwindigkeitssteigerung feststellen. Die Gesamtzeit in der untersten Zeile ist die Zeit, die wir für die Auswertung einer Menge von 2^{32} Chiffretexten benötigen. Danach haben wir also die Menge der möglichen Schlüssel von 2^{40} auf 2^{32} reduziert und erwarten, daß wir 4 weitere Textmengen auswerten müssen, um eine eindeutige Informationen über die Rundenschlüssel $\widehat{K}^5$ und K^6 zu erhalten.

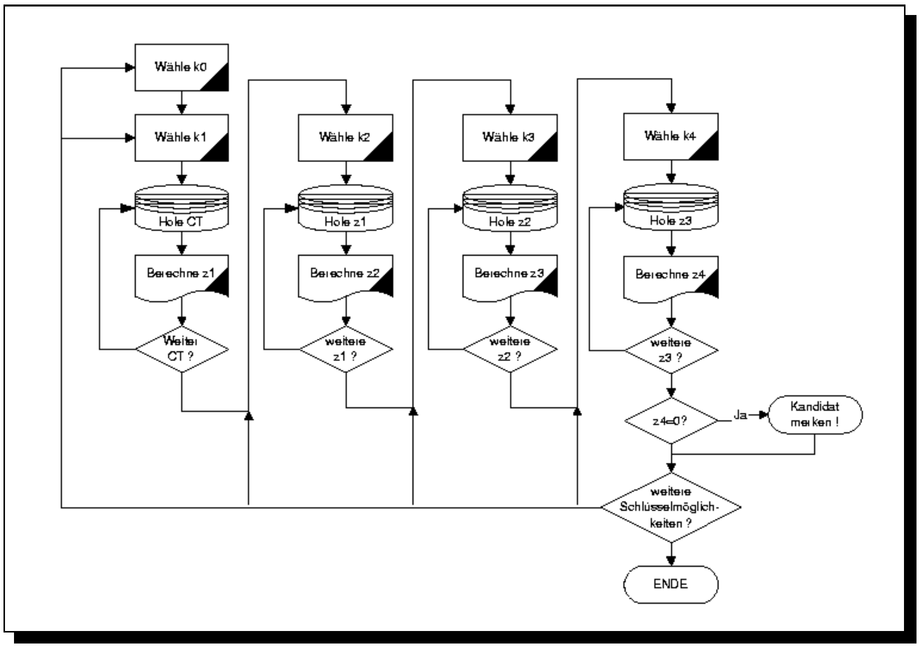

Abbildung 4.6.: Angriff mit Teilsummenverbesserung als Diagramm

4.4. Mögliche Verbesserung

4.4.1. Idee

Ein möglicher Ansatz zur weiteren Verbesserung des Teilsummen-Angriffs aus 4.3 besteht darin, die zusätzlichen Informationen, die bei den Table-Lookups anfallen, mitzunutzen. Ein Table-Lookup der S-Boxen $\mathbf{S_0}[a], \ldots, \mathbf{S_3}[a]$ liefert einen 32 Bit Wert zurück, von dem im Original-Angriff aber nur 8 Bit genutzt werden. Die restlichen 24 Bit werden verworfen, weil diese nicht nötig sind, um den 1-Byte Test auf Ausgeglichenheit in Runde 5 durchzuführen. Die Idee besteht nun darin, die zusätzlichen Informationen mitzuverarbeiten, um so schneller eine eindeutige Schlüsselinformation zu erhalten. Hierzu wird angenommen, daß ein **XOR** über 8 Bit genauso teuer ist wie ein **XOR** über 32 Bit. Gemäß dieser Annahme ist die Verarbeitung der zusätzlichen Informationen in Stufe 1,2,3 ohne Mehraufwand gegenüber dem Original-Angriff durchzuführen. In Stufe 4 kann man nun die Ausgeglichenheit an allen 4 Positionen testen, was zu einer Reduzierung der möglichen Schlüsselkandidaten führt.

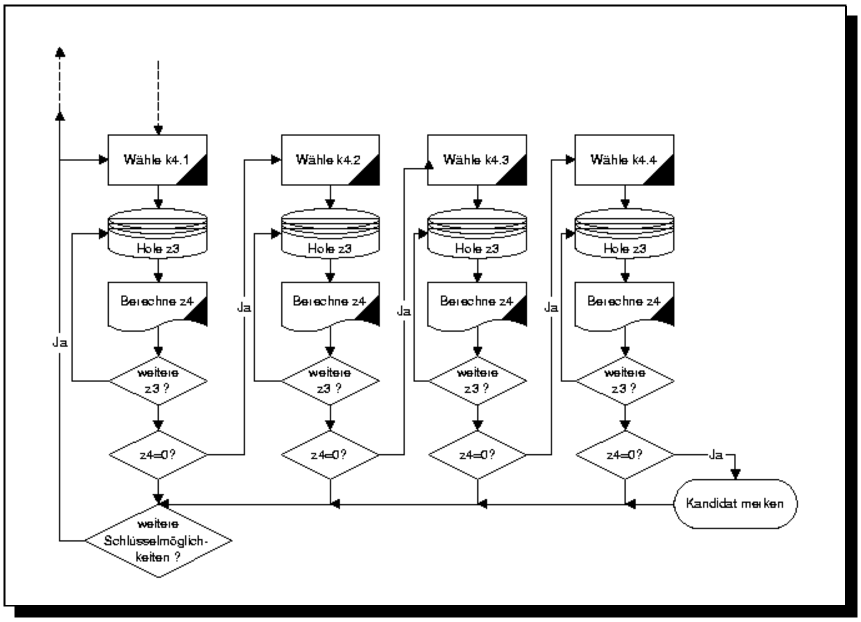

Abbildung 4.7.: Stufe 4 der Verbesserung

4.4.2. Reduzierung der Schlüsselkandidaten

Um festzustellen, inwieweit diese Vorgehensweise eine Verbesserung darstellt, ist es nötig zu wissen, um welchen Faktor die Anzahl möglicher Kandidaten reduziert wird.

Bei dem Original-Angriff reduziert sich diese Anzahl jeweils um den Faktor 2^8, vorausgesetzt, daß die Eingaben für die 4.Stufe zufällig gleichverteilt sind. Wenn wir die 4 Schlüsselbytes $k_0, \ldots, k_3$ von K^6 gewählt haben, finden wir im Durchschnitt jeweils ein Schlüsselbyte k_4, das zur Ausgeglichenheit führt.

$$\Pr(k_4 \text{ gleicht aus } | k_0 \ldots k_3 \text{ fest}) = 1/256$$

Bei der hier untersuchten Veränderung hängt die Akzeptanz eines Kandidaten aber von 4 Bytes aus $\widehat{K}^5$ ab, d.h. es muß für jede der 4 Positionen einen Schlüsselwert geben, der an dieser Position zur Ausgeglichenheit führt.

Wir betrachten zunächst einmal die Reduzierung der Schlüsselkandidaten, falls nur eine weitere Position getestet wird. Wir durchlaufen die Stufen 1 bis 3, fixieren also die Schlüsselbytes (k_0, k_1, k_2, k_3) und suchen nun $k_{4,0}$ und $k_{4,1}$, die an verschiedenen Positionen zur Ausgeglichenheit führen. Die Schlüsselbytekombination wird nur akzeptiert, falls wir an beiden Positionen Ausgeglichenheit feststellen können. Ist dies nicht der Fall, so können (k_0, k_1, k_2, k_3) nicht Teil des Chiffrierschlüssels

# Positionen	Reduzierungsfaktor	mögl. Kand.
1	2^8	2^{32}
2	$0,63 \cdot 2^8$	$0,63 \cdot 2^{32}$
3	$0,63^2 \cdot 2^8$	$0,63^2 \cdot 2^{32}$
4	$0,63^3 \cdot 2^8$	$\sim 2^{30}$

Tabelle 4.5.: Reduzierung der Schlüsselkandidaten

sein. Die Wahrscheinlichkeit, daß für feste (k_0, k_1, k_2, k_3) keine Ausgeglichenheit möglich ist, berechnen wir wie folgt:

$$\begin{aligned}
& \text{Pr (keines der } k_4 \text{ balanciert } |k_0 \ldots k_3 \text{ fest)} \\
= \; & \text{Pr (ein } k_4 \text{ balanciert nicht } |k_0 \ldots k_3 \text{ fest)}^{256} \\
= \; & (1 - \text{Pr (ein } k_4 \text{ balanciert} |k_0 \ldots k_3 \text{ fest))}^{256} \\
= \; & (1 - \frac{1}{256})^{256} \\
= \; & \left(\frac{255}{256}\right)^{256} \approx 0,367
\end{aligned}$$

Also ist die Wahrscheinlichkeit, daß eines der 256 möglichen k_4 zur Ausgeglichenheit führt:

$$1 - \left(\frac{255}{256}\right)^{256} \approx 0,63$$

d.h. der Test eines weiteren Bytes reduziert die Anzahl der möglichen Kandidaten auf das $0,63$ fache. Die Einbeziehung weiterer Positionen bewirkt jeweils eine weitere Reduzierung um diesen Faktor bis hin zu einem max. Faktor von $0,63^3 \approx \frac{1}{4}$. Ausgehend von anfangs 2^{40} Schlüsselkandidaten zeigt Tabelle 4.5 den Reduzierungsfaktor in Abhängigkeit von der Anzahl der getesteten Positionen.

4.4.3. Aufwand für die Auswertung einer Textmenge

Unter den oben genannte Voraussetzungen, daß ein 8-Bit **XOR** genauso teuer ist wie ein 32-Bit **XOR** , gibt es keinen Mehraufwand in den Stufen 1-3. In Stufe 4 wird die erste Position immer getestet, d.h. man hat immer mindestens den Aufwand des Original-Angriffs. Bei Erfolg wird die jeweils nächste Position getestet, d.h. man hat einen maximalen Mehraufwand von 4 mal dem Original-Angriff. Unter der Annahme, daß die Eingaben zufällig gleichverteilt sind, ist die erwartete Anzahl von Positionen, die wirklich getestet werden müssen, in Tabelle 4.6 dargestellt. D.h der

Pos	erw. Tests	$\approx$
1	256	256
2	$0,63 \cdot 256$	162
3	$0,63^2 \cdot 256$	$102,5$
4	$0,63^3 \cdot 256$	$64,9$
		$585,4$

Tabelle 4.6.: Erwartete Anzahl von Tests

erwartete Aufwand in Stufe 4 beträgt pro Schlüsselkandidat ca. $585,4/256 = 2,28$ Berechnungen der Teilsumme

$$z_4 = \mathbf{S^{-1}}[z_3 \oplus k_4].$$

Da der Aufwand zum Testen der ersten Position der gleiche ist, wie in der originalen Teilsummenverbesserung, müssen wir für den Gesamtaufwand nur den Aufwand von $2,28 - 1 = 1,28$ Berechnungen der Teilsumme z_4 addieren. Hierfür benötigen wir in unserem Kostenmaß jeweils 20 **Op**, so daß wir zu dem Aufwand der Teilsummenverbesserung von $93 \cdot 2^{48}$ **Op** einen Aufwand von $1,3 \cdot 20 \cdot 2^{48}$ **Op** addieren müssen. Insgesamt erhalten wir also einen Aufwand von $119 \cdot 2^{48}$ **Op** bzw. $2^{44.8}$ Verschlüsselungen.

4.4.4. Bewertung der Verbesserung

In diesem Abschnitt wollen wir unsere Verbesserung mit dem originalen Teilsummenangriff aus 4.3 vergleichen. Wir gehen davon aus, daß wir genügend Textpaare generieren können, um den originalen Teilsummenagriff ausführen zu können [4] und daß wir für eine Auswertung von Textpaaren nur die Schlüsselkandidaten heranziehen, die bei einer vorherigen Auswertung übrig geblieben sind. Betrachten wir zunächst den originalen Teilsummenangriff:

Nach der Auswertung einer Textmenge erwarten wir, daß zu jedem Tupel (k_0, k_1, k_2, k_3) ein Schlüsselbyte k_4 übrig bleibt. Bei der Auswertung einer zweiten Textmenge prüfen wir, ob zu jedem (k_0, k_1, k_2, k_3) wieder dasselbe k_4 aus der ersten Auswertung zu Ausgeglichenheit führt. Ist dies der Fall, so bleibt diese Schlüsselbytekombination ein möglicher Kandidat, andernfalls können wir diese Kombination definitv ausschließen. Gehen wir wieder davon aus, daß die Zwischenergebnisse in A^5 zufällig gleichverteilt sind, so besteht ein k_4 aus der ersten Auswertung den

[4] Wir vergleichen hier nur die reine Laufzeit, nicht die Ansprüche an die Anzahl der Textpaare.

Test mit Wahrscheinlichkeit 1/256, die Kandidatenmenge wird also wieder um den Faktor 2^8 reduziert.

In unserer Verbesserung berechnen wir bei der Auswertung der ersten Textmenge zu jedem Tupel (k_0, k_1, k_2, k_3) ein Tupel $(k_{4,0}, k_{4,1}, k_{4,2}, k_{4,3})$ von Schlüsselbytes aus $\widehat{K}^5$, die an verschiedenen Positionen in A^4 Ausgeglichenheit erzeugen. Hier erwarten wir, daß im Durchschnitt für ca. 1/4 der 2^{32} Tupel (k_0, k_1, k_2, k_3) ein Tupel $(k_{4,0}, k_{4,1}, k_{4,2}, k_{4,3})$ übrig bleibt.
Werten wir nun eine zweite Textmenge aus, so muß auch hier zu jedem Tupel (k_0, k_1, k_2, k_3) exakt das gleiche Tupel $(k_{4,0}, k_{4,1}, k_{4,2}, k_{4,3})$ aus der vorherigen Auswertung zur Ausgeglichenheit führen. Ist dies nicht der Fall, so verwerfen wir diese Schlüsselbytekombination. Andernfalls haben wir einen Kandidaten gefunden. Die Wahrscheinlichkeit für jedes der $(k_{4,0}, k_{4,1}, k_{4,2}, k_{4,3})$ wieder zur Ausgeglichenheit zu führen beträgt unabhängig voneinander 1/256. Das gesamte Tupel führt also nur mit Wahrscheinlichkeit $1/2^{32}$ zur Ausgeglichenheit. Da wir aber erwarten, nur noch 2^{30} Schlüsselbytekombinationen testen zu müssen, bleibt mit Wahrscheinlichkeit 3/4 nur die richtige Schlüsselbytekombination übrig.

Im Gegensatz zum originalen Teilsummenangriff, der nach Auswertung von 5 Textmengen mit Wahrscheinlichkeit 1/2 eine eindeutige Lösung liefert, benötigen wir nur 2 Textmengen und sind dann mit Wahrscheinlichkeit 3/4 eindeutig. Beziehen wir den zusätzlichen Aufwand von ca. einem Faktor 1.3 für unsere Verbesserung mit ein, so erzielen wir insgesamt in etwa eine Verbesserung vom Faktor $(1 \cdot 5)/(1.3 \cdot 2) \approx 2$.

4.5. Verbesserung bei Shift nach rechts

In der ersten Version des Rijndael Proposals von 1998 wurde die ShiftRow Transformation leicht anders beschrieben. Hier wurden die zyklischen Verschiebungen innerhalb einer Zeile beim Verschlüsseln nach rechts statt nach links vorgenommen. In der zweiten Version des Proposals wurden dann die Richtungen der Verschiebungen vertauscht. Wir werden in diesem Abschnitt zeigen, daß der in 4.2.6 vorgestellte Chosen Ciphertext Angriff in diesem Fall um einen Faktor von 2^8 schneller ist. Dazu verwenden wir die äquivalente Struktur einer Entschlüsselung aus Abschnitt 3.4.7, die in Abb. 3.11 dargestellt ist.

Der Angriff Wir betrachten hier den 6 Runden-Angriff aus Abschnitt 4.2.6 als Chosen Ciphertext Angriff. Wir können also beliebige Chiffretexte wählen und diese entschlüsseln lassen. Analog zum Chosen Plaintext Angriff wählen wir nun eine Menge von 256 Texten, die nach Entschlüsselung einer Runde eine Λ-Menge ergeben und nach 4 Runden Entschlüsseln ist jedes Byte ausgeglichen. Um diese Textmenge zu finden, müssen wir wie beim Chosen Plaintext Angriff auch, 4

Bytes des ersten Rundenschlüssels, in diesem Fall K^6, raten. Zu dieser Textmenge wählen wir die korrespondierenden Klartexte, die wir 2 Runden verschlüsseln. Hierzu müssen wir 4 Schlüsselbytes von K^0 raten und im Anschluss daran ein Byte von K^1 raten. Der Angriff verläuft also bis hierhin exakt genauso wie der Chosen Plaintext Angriff. Die Verbesserung ist nun dadurch möglich, daß wir den KeySchedule mit in unseren Angriff einbinden. In Abb. 4.8 ist der Beginn des KeySchedules mit unserer Wahl der zu ratenden Schlüsselbytes dargestellt. Wie

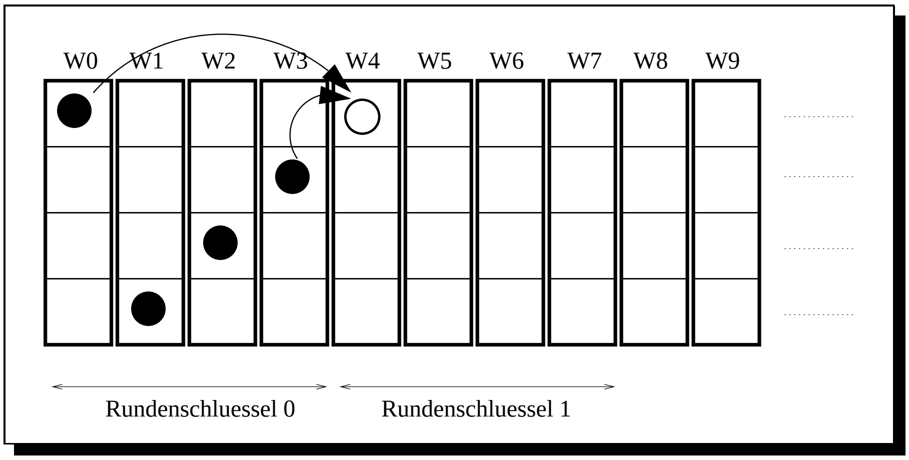

Abbildung 4.8.: Wahl der Schlüsselbytes von K^0

man hier sieht, ermöglicht es uns unsere Wahl der 4 Schlüsselbytes von K^0, das Byte von K^1 mittels des Keyschedules zu berechnen. Dadurch ersparen wir uns den Aufwand 2^8 Schlüsselkombinationen zu testen und müssen insgesamt nur 8 Schlüssselbytes raten, was in dem vereinfachten Kostenmaß einen Aufwand von 2^{64} statt 2^{72} Verschlüsselungen bedeutet.

Wie wir an diesem Beispiel gesehen haben, ist die Richtung der zyklischen Verschiebung im ShiftRow nicht unerheblich. So ermöglicht die Verschiebung bei der Verschlüsselung nach rechts zwar nicht, den Angriff auf mehr als 6 Runden zu erweitern, aber läßt zumindest eine einfache Reduzierung des Aufwands zu.

4.6. Berechnung des Chiffrierschlüssels mit dem SQUARE-Angriff

Bis jetzt haben wir immer nur Teilinformationen von Rundenschlüsseln berechnet. In diesem Abschnitt erläutern wir, wie man mit dem SQUARE-Angriff in Kombination mit dem KeySchedule den Chiffrierschlüssel für 6-Runden Rijndael mit

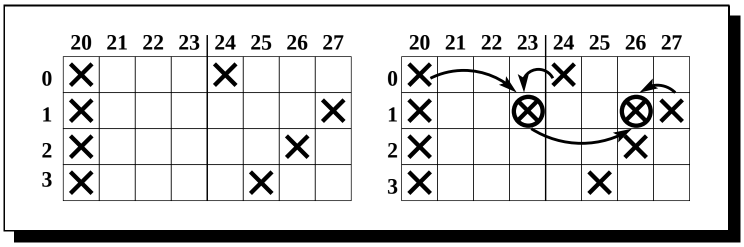

Abbildung 4.9.: Bekannte Schlüsselbytes nach erstem Angriff (links) und Ausnutzung des Key-Schedules (rechts)

Blocklänge und Schlüssellänge von 128 Bit berechnen kann. Grundlage hierfür ist, daß aus dem Wissen von N_k aufeinanderfolgenden Spalten des erweiterten Schlüssels W, der Chiffrierschlüssel allein mittels des Keyschedules ermittelbar ist. In unserem Fall reicht also die Kenntnis eines Rundenschlüssels aus, um den Key-Schedule rückwärts anwenden zu können. Wir konzentrieren uns hier auf den 5. und 6. Rundenschlüssel.

Im folgenden beziehen wir uns auf einzelne Bytes des Erweiterten Schlüssels W durch Angabe der Zeile und der Spalte z.B. bezeichnet $W_{0,24}$ das obere Byte in Spalte 24. Für den SQUARE-Angriff werden ca. 5 Mengen zu je 2^{32} Chosen Plaintexts benötigt, um eine eindeutige Schlüsselinformation über 4 Schlüsselbytes von K^6 und 1 Schlüsselbyte von K^5 zu gewinnen. Nach einem solchen Angriff kennen wir 4 korrekte Bytes von K^6 und können mit diesem Wissen die restlichen 3 Bytes der Spalte von K^5 einfach durch Testen jeweils aller 256 Möglichkeiten berechnen.

Im Prinzip ist die Auswahl der Spalte, die in dem Angriff verwendet werden soll, beliebig. Für das weitere Vorgehen ist aber die Wahl der ersten Spalte des 6. Rundenschlüssels (Spalte 24) die günstigste Möglichkeit, da hier die Eigenschaften des Key-Schedule optimal in weiteren Angriffen verwendet werden können. Nach dem Angriff auf die Spalte 24 verfügt man über 8 Schlüsselbytes, wie in Abb. 4.9 (links) dargestellt. Durch Ausnutzung des Key-Schedules kann zunächst das Byte

$$W_{1,23} = \mathbf{S^{-1}}[W_{0,24} \oplus W_{0,20} \oplus \mathbf{Rcon}[24/4]]$$

und dann

$$W_{1,26} = W_{1,27} \oplus W_{1,23}$$

berechnet werden (siehe Abb 4.9 rechts).

Diese beiden neu bestimmten Bytes finden nun Verwendung in einem neuen Angriff, bei dem wir die Schlüsselbytes $W_{0,27}$, $W_{1,26}$,$W_{2,25}$,$W_{3,24}$ von K^6 und $W_{1,23}$ von K^5 berechnen wollen. Die Bytes $W_{1,26}$ und $W_{1,23}$ konnten wir aber schon

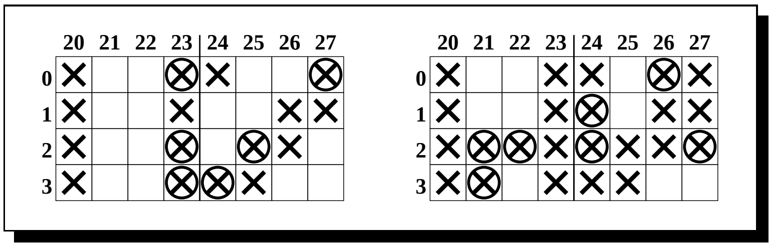

Abbildung 4.10.: Bekannte Schlüsselbytes nach dem zweiten Angriff (links) und Ausnutzung des Key-Schedules (rechts)

mittels des KeySchedules berechnen, so daß nur noch 2^{24} Schlüsselmöglichkeiten geprüft werden müssen. Nach diesem Angriff und der Bestimmung der restlichen 3 Bytes von K^5 sind also insgesamt 16 Schlüsselbytes bekannt (vgl. Abb 4.10(links)). Daraus können folgende neue Bytes berechnet werden:

$$\begin{aligned}
W_{2,22} &= W_{2,26} \oplus W_{2,25} \\
W_{3,21} &= W_{3,25} \oplus W_{3,24} \\
W_{2,27} &= W_{2,26} \oplus W_{2,23} \\
W_{1,24} &= W_{1,20} \oplus \mathbf{S}[W_{2,23}] \\
W_{2,24} &= W_{2,20} \oplus \mathbf{S}[W_{3,23}] \\
W_{0,26} &= W_{0,27} \oplus W_{0,23} \\
W_{2,21} &= W_{2,25} \oplus W_{2,24}
\end{aligned}$$

Die restlichen Schlüsselbytes werden mit einem letzten Angriff ermittelt, bei dem wir die Bytes $W_{0,25}, W_{1,24}, W_{2,27}, W_{3,26}$ von K^6 und das Byte $W_{3,21}$ von K^5 bestimmen wollen. Die Bytes $W_{1,24}$,$W_{2,27}$ und $W_{3,21}$ haben wir aber bereits berechnet, so daß der Angriff nur nur noch 2^{16} Möglichkeiten testen muß. Dieser liefert 4 neue Schlüsselbytes (vgl. Abb. 4.11 links) und ermöglicht damit die Berechnung der restlichen Schlüsselbytes:

$$\begin{aligned}
W_{0,22} &= W_{0,26} \oplus W_{0,25} \\
W_{1,25} &= W_{1,24} \oplus W_{1,21} \\
W_{1,22} &= W_{1,26} \oplus W_{1,25} \\
W_{3,27} &= W_{3,26} \oplus W_{3,23} \\
W_{3,22} &= W_{3,27} \oplus W_{3,26}
\end{aligned}$$

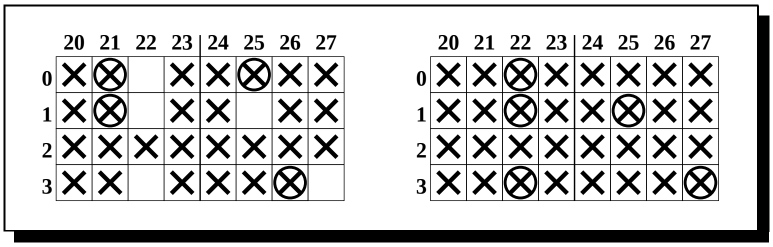

Abbildung 4.11.: Bekannte Schlüsselbytes nach dem dritten Angriff (links) und Ausnutzung des Key-Schedules (rechts)

Für die Berechnung des Chiffrierschlüssels kann nun der Key-Schedule in der umgekehrten Richtung durchlaufen werden. Der Gesamtaufwand wird von dem ersten Angriff dominiert, der mittels Teilsummenverbesserung einen Aufwand von ca. $5 \cdot 2^{45}$ Verschlüsselungen benötigt. Die nachgeschalteten Angriffe sind mit 2^{24} bzw. 2^{16} Schlüsselkombinationen deutlich weniger aufwendig und können die Mengen verschlüsselter Texte des ersten Angriffs "wiederverwenden". Wir verwenden hier den normalen SQUARE-Angriff aus 4.2.5, weil dieser mit weniger Texten operiert, was bei diesen "kleinen" Mengen an Schlüsselkombinationen zu einem Aufwand von nur 2^{24} bzw. 2^{16} Verschlüsselungen führt.

Zusammenfassend ist zu sagen, daß der Aufwand zur Berechnung des Chiffrierschlüssels mittels des SQUARE-Angriffs von dem Aufwand für den ersten Angriff dominiert wird. Im Anschluß daran kann man den KeySchedule nutzen, um die restlichen Bytes mit deutlich weniger Aufwand zu bestimmen.

4.7. Übersicht über die Angriffe

Um dieses Kapitel abzuschließen, möchten wir einen kurzen Überblick über die in dieser Arbeit vorgestellten Angriffe geben. In Tabelle 4.7 haben wir alle Angriffe mit Rundenzahl, benötigter Textmenge und zeitlichem Aufwand in Verschlüsselungen aufgelistet.

4.8. Beobachtung zur S-Box

Im folgenden betrachten wir die S-Box der ByteSub Transformation. Wir untersuchen die Länge der Zyklen, die durch wiederholtes Anwenden der Substitution auf ein Element entstehen. Für ein Element a bezeichnen wir die n-fache Hintereinan-

Angriff	# Textpaare	# Verschlüsselungen
Basis Angriff (4 Runden)	$2 \cdot 2^8$	$2 \cdot 2^8$
Erweiterung am Ende (4+1 Angriff)	$5 \cdot 2^8$	$5 \cdot 2^{40}$
Erweiterung am Anfang (1+4 Angriff)	2^{32}	$5 \cdot 2^{40}$
6 Runden SQUARE-Angriff	$5 \cdot 2^{32}$	$5 \cdot 2^{72}$
Umgedrehter SQUARE-Angriff	$4 \cdot 2^{32}$	$5 \cdot 2^{64}$
SQUARE-Angriff mit Teilsummen	$5 \cdot 2^{32}$	$5 \cdot 2^{45}$
Unsere Verbesserung	$2 \cdot 2^{32}$	$2 \cdot 2^{45}$

Tabelle 4.7.: Übersicht über die Angriffe

derausführung mit

$$\underbrace{\mathbf{S}[\ldots\mathbf{S}[a]]}_{n\ \text{mal}} = \mathbf{S}[a]^n$$

Da ByteSub eine invertierbare Abbildung ist, muß zu jedem Element a ein $0 \leq n \leq 255$ existieren, mit $\mathbf{S}[a]^n = a$. Wir interessieren uns nun für die Länge n dieser Zyklen. Bei der festgelegten S-Box von Rijndael können wir diese Zyklen leicht per vollständiger Suche ermitteln. Insgesamt treten 5 disjunkte Zyklen auf, die wir mit ihrem kleinsten enthaltenen Element charakterisieren. In Tabelle 4.8 haben wir die Zyklen mit ihrer Länge aufgelistet. Der Zyklus 115 fällt wegen seiner kurzen

Zyklus	Länge
0	59
1	81
4	87
11	27
115	2

Tabelle 4.8.: Zyklen der S-Box

Länge 2 auf. Es gilt $\mathbf{S}[\mathbf{S}[115]] = 115$ und für das zweite Element 143 gilt ebenfalls $\mathbf{S}[\mathbf{S}[143]] = 143$.

Für die Verschlüsselung bedeutet dies, daß eine mehrfache Hintereinanderausführung von ByteSub die Sicherheit nicht erhöht, denn z.B. bei den Elementen 115 und 143 würden die jeweiligen Ergebnisse immer einem der beiden Elemente entsprechen. Wenn man sich nun die Verschlüsselungsvorschrift von Rijndael ansieht, so stellt man fest, daß nach jeder Substitution eine lineare Modifikation erfolgt. Sowohl das MixColumn als auch das **XOR** mit dem Rundenschlüssel verändern das Element. Es kommt also während einer Verschlüsselung im allgemeinen nicht zu einer doppelten Substitution, so daß ein kurzer Zyklus kein Problem darstellen sollte.

Die S-Box wird aber auch im KeySchedule verwendet. Hier wird ein Element nach der Substitution nur durch **XOR** mit anderen Elementen des vorherigen Rundenschlüssels verändert. Treten hier nur Nullwerte auf, so kann es zum Durchlaufen des Zyklus führen. Wir können also Chiffrierschlüssel angeben, die in einer begrenzten Anzahl von Rundenschlüsseln eine gewisse Struktur erhalten. Betrachten wir folgendes Beispiel:

K^0				K^1				K^2				K^3				K^4			
108	105	103	115	14	103	0	115	111	8	8	123	8	0	8	115	115	115	123	8
0	99	0	0	99	0	0	0	0	0	0	0	143	143	143	143	0	143	0	143
99	0	0	0	0	0	0	0	115	115	115	115	0	115	0	115	253	142	142	253
0	0	0	0	143	143	143	143	0	143	0	143	33	174	174	33	174	0	174	143

Hier sieht man, wie im ersten Rundenschlüssel die vierte Zeile nur aus den Elementen 143 besteht. Diese Zeile wird im zweiten Rundenschlüssel um eins nach oben geschoben und alle Elemente nehmen den Wert 115 an. Im Rundenschlüssel 3 wird die Zeile wieder um eins nach oben geschoben und die Elemente wechseln wieder zur 143. Man sieht auch, wie sich jeweils unter dieser Zeile eine andere Zeile mit ähnlicher Struktur gebildet hat. Hier sind nicht alle 4 Elemente gleich, sondern jeweils das erste und dritte Element der Zeile bleibt 0. In den weiteren Rundenschlüsseln verschwindet diese einfache Struktur.

Die Bedeutung dieser Eigenschaft scheint für die Verschlüsselung mit der vollen Rundenzahl keinerlei Bedeutung zu haben. Nicht nur, daß die Struktur bereits nach 4 Rundenschlüsseln zu verschwinden scheint, sondern es ist auch zu beachten, daß sich die Struktur zeilenweise fortpflanzt. In der Verschlüsselung würde ein solcher Rundenschlüssel auf einen Zustand addiert, der dann wiederum spaltenweise vom MixColumn modifiziert wird. Uns ist nicht klar, ob und wie man diese Eigenschaft effektiv zur Kryptoanalyse von Rijndael einsetzen kann.

5. Anwendung in Hashfunktionen

« Somehow the verb “to hash” magically became standard terminology for key transformation during the mid-1960s, yet nobody was rash enough to use such an undignified word in print until 1967! »

Donald E. Knuth [1]

In diesem Kapitel beschäftigen wir uns mit einem Grundbaustein der modernen Kryptographie, den Hashfunktionen. Eine Hashfunktion bildet aus einem Text beliebiger Länge einen Wert fester Länge, den sogenannten Hashwert. Andere Bezeichnungen sind engl. Message Digest oder Fingerabdruck in Anlehnung an den Fingerabdruck eines Menschen. Die Bezeichnung “Hash” stammt von dem altfranzösischen Wort “hachis” für das Gehackte beim Metzger [von99]. Der Text wird also bildlich gesprochen zerhackt, durch den Fleischwolf gedreht und am Ende wird nur ein kleiner Teil verwendet, in den aber die gesamte Information eingeflossen ist. Eine Hashfunktion kann zur Überprüfung der Datenintegrität verwendet werden, indem zusätzlich zur Nachricht der Hashwert übermittelt wird. Nach der Übertragung wird der Hashwert erneut berechnet und mit dem übertragenen Hashwert verglichen.

Zu beachten ist hier, daß eine Hashfunktion keinerlei geheime Information verarbeitet; der Hashwert wird nach einem öffentlich bekannten Verfahren berechnet, schützt also allein nicht gegen mutwillige Änderungen. Ein Angreifer kann den Inhalt des Textes ändern und den passenden Hashwert selbst berechnen. Soll die Nachricht auch gegen mutwillige Änderungen geschützt werden, so müssen sogenannte kryptographische Hashfunktionen verwendet werden und der Hashwert muß mit anderen kryptographischen Verfahren z.B. Verschlüsselung geschützt werden.

An kryptographische Hashfunktionen werden zusätzliche Anforderungen gestellt; so soll es schwierig sein, zu einem gegebenen Hashwert eine passende Nachricht zu finden. Eine Hashfunktion mit dieser Eigenschaft liefert uns also einen Hashwert, den wir in kryptographischen Verfahren als Stellvertreter für den originalen Texte verwenden können. Das klassische Einsatzgebiet in der Kryptographie stellt die digitale Signatur dar. Für die Unterzeichnung eines Textes wer-

[1] The Art of Computer Programming Vol.3

den rechenintensive Verfahren eingesetzt und die berechnete digitale Signatur hat üblicherweise die Länge des unterzeichneten Textes. Hier ermöglicht der Einsatz von Hash-Funktionen einen immensen Zeit- und Speicherplatzvorteil. Unterzeichnet wird nämlich nicht der Text selber, sondern stellvertretend der berechnete Hashwert, der im allgemeinen wesentlich kürzer ist als der Text.

Ein anderes Einsatzgebiet ist der Schutz von Zugangspassworten z.B. für Unix. Für jeden Benutzer wird nur der Hashwert seines Passwortes gespeichert. Bei einer Anmeldung wird der Hashwert des eingegebenen Passwortes mit dem abgespeicherten Hashwert verglichen und bei Gleichheit wird die Anmeldung akzeptiert. Diese Vorgehensweise verhindert, daß die Passworte der Benutzer direkt gespeichert werden müssen und mißbraucht werden können.

Daraus ergibt sich die Anforderung an den Hashwert, daß er lang genug sein muß, um eine erschöpfende Suche zu aufwendig zu machen.

Betrachten wir das Szenario, daß ein Angreifer eine gültige digitale Unterschrift zu dem Hashwert einer Nachricht abgefangen hat. Er kann nun versuchen, zu diesem Hashwert eine andere Nachricht zu finden, so daß die Unterschrift auch für diesen Text gültig ist. In Abb. 5.1 zeigen wir ein Beispiel aus [DP84], wie man leicht 2^{37} verschiedene Variationen eines betrügerischen Briefes allein durch Verwendung synonymer Begriffe erzeugen kann. Der Angreifer berechnet zu jeder Variante den Hashwert und prüft, ob er zu der Unterschrift paßt. Je kürzer der Hashwert ist, umso größer ist die Wahrscheinlichkeit, mit diesem Verfahren eine Unterschrift zu fälschen.

In diesem Kapitel stellen wir zunächst die Grundlagen für kryptographische Hashfunktionen zusammen. Danach stellen wir unter anderem Methoden/Verfahren vor, wie man aus Blockchiffren Hashfunktionen generiert und im letzten Teil untersuchen wir die Sicherheit von Hashfunktionen, wenn wir Rijndael mit reduzierter Anzahl von Runden als Grundlage verwenden.

5.1. Grundlagen kryptographischer Hashfunktionen

Definition 5.1.1 (Hashfunktion) *Eine Hashfunktion h berechnet effizient zu einer Nachricht M beliebiger Länge einen Hashwert $H = h(M)$ mit fester Länge n (Kompression).*

Die Eigenschaft der Kompression führt dazu, daß h nicht injektiv ist. Es gibt also sogenannte Kollisionen, d.h. es existieren mehrere Nachrichten, die denselben Hashwert haben. Für den Einsatz innerhalb der Kryptographie ist es aber notwendig, daß diese Kollisionen nicht effizient berechnet werden können.

Eigenschaften kryptographischer Hashfunktionen Je nach Einsatzgebiet werden folgende Bedingungen an kryptographische Hashfunktionen gestellt:

Dear Anthony,

$\left\{\begin{array}{c}\text{This letter is}\\ \text{I am writing}\end{array}\right\}$ to introduce $\left\{\begin{array}{c}\text{you to}\\ \text{to you}\end{array}\right\}$ $\left\{\begin{array}{c}\text{Mr.}\\ \text{—}\end{array}\right\}$ Alfred $\left\{\begin{array}{c}\text{P.}\\ \text{—}\end{array}\right\}$

Barton, the $\left\{\begin{array}{c}\text{new}\\ \text{newly appointed}\end{array}\right\}$ $\left\{\begin{array}{c}\text{chief}\\ \text{senior}\end{array}\right\}$ jewellery buyer for

$\left\{\begin{array}{c}\text{our}\\ \text{the}\end{array}\right\}$ Northern $\left\{\begin{array}{c}\text{European}\\ \text{Europe}\end{array}\right\}$ $\left\{\begin{array}{c}\text{area}\\ \text{Division}\end{array}\right\}$. He $\left\{\begin{array}{c}\text{will take}\\ \text{has taken}\end{array}\right\}$ over

$\left\{\begin{array}{c}\text{the}\\ \text{—}\end{array}\right\}$ responsibility for $\left\{\begin{array}{c}\text{all}\\ \text{the whole of}\end{array}\right\}$ our interests in

$\left\{\begin{array}{c}\text{watches and jewellery}\\ \text{jewellery and watches}\end{array}\right\}$ in the $\left\{\begin{array}{c}\text{area}\\ \text{region}\end{array}\right\}$. Please $\left\{\begin{array}{c}\text{afford}\\ \text{give}\end{array}\right\}$

him $\left\{\begin{array}{c}\text{every}\\ \text{all the}\end{array}\right\}$ help he $\left\{\begin{array}{c}\text{may need}\\ \text{needs}\end{array}\right\}$ to $\left\{\begin{array}{c}\text{seek out}\\ \text{find}\end{array}\right\}$ the most

$\left\{\begin{array}{c}\text{modern}\\ \text{up to date}\end{array}\right\}$ lines for the $\left\{\begin{array}{c}\text{top}\\ \text{high}\end{array}\right\}$ end of the market. He is

$\left\{\begin{array}{c}\text{empowered}\\ \text{authorized}\end{array}\right\}$ to receive on our behalf $\left\{\begin{array}{c}\text{samples}\\ \text{specimens}\end{array}\right\}$ of the

$\left\{\begin{array}{c}\text{latest}\\ \text{newest}\end{array}\right\}$ $\left\{\begin{array}{c}\text{watch and jewellery}\\ \text{jewellery and watch}\end{array}\right\}$ products, $\left\{\begin{array}{c}\text{up}\\ \text{subject}\end{array}\right\}$ to a

$\left\{\begin{array}{c}\text{limit}\\ \text{maximum}\end{array}\right\}$ of ten thousand pounds sterling. He will $\left\{\begin{array}{c}\text{carry}\\ \text{hold}\end{array}\right\}$

a signed copy of this $\left\{\begin{array}{c}\text{letter}\\ \text{document}\end{array}\right\}$ as proof of identity. An order

with this signature, which is $\left\{\begin{array}{c}\text{appended}\\ \text{attached}\end{array}\right\}$ $\left\{\begin{array}{c}\text{authorizes}\\ \text{allows}\end{array}\right\}$ you to

charge the cost to this company at the $\left\{\begin{array}{c}\text{above}\\ \text{head office}\end{array}\right\}$ address. We

$\left\{\begin{array}{c}\text{fully}\\ \text{—}\end{array}\right\}$ expect that our $\left\{\begin{array}{c}\text{level}\\ \text{volume}\end{array}\right\}$ of orders will increase in the

$\left\{\begin{array}{c}\text{following}\\ \text{next}\end{array}\right\}$ year and $\left\{\begin{array}{c}\text{trust}\\ \text{hope}\end{array}\right\}$ that the new appointment will $\left\{\begin{array}{c}\text{be}\\ \text{prove}\end{array}\right\}$

$\left\{\begin{array}{c}\text{advantageous}\\ \text{an advantage}\end{array}\right\}$ to both our companies.

Abbildung 5.1.: 2^{37} Variationen eines betrügerischen Briefes

- **Einwegfunktion:** Gegeben einen Hashwert H soll es nicht effizient möglich sein, eine Nachricht M zu finden, mit $h(M) = H$. Diese Eigenschaft wird auch **preimage resistance** genannt.

- **schwach kollisionsresistent:** Gegeben eine Nachricht M mit Hashwert $H = h(M)$ soll es nicht effizient möglich sein, eine zweite Nachricht $M' \neq M$ zu finden mit $h(M') = h(M) = H$. Diese Eigenschaft wird auch **2nd preimage resistance** genannt.

- **stark kollisionsresistent:** Es soll nicht effizient möglich sein, zwei beliebige Nachrichten M, M' mit $M \neq M'$ zu finden, die denselben Hashwert haben ($h(M) = h(M')$). Diese Eigenschaft wird auch **collision resistance** genannt.

Eine obere Schranke für die Sicherheit einer Hashfunktion h ist durch eine Funktion in der Länge n des Hashwertes H gegeben. Ein zu kurzer Hashwert ermöglicht eine erschöpfende Suche und bietet somit keine Sicherheit.

Im folgenden betrachten wir jede dieser 3 Eigenschaften unter dem Aspekt, wie groß der Aufwand für eine erschöpfende Suche in Abhängigkeit von n ist. Der Aufwand zum Brechen einer guten kryptographischen Hashfunktion sollte im Bereich dieser oberen Schranken liegen.

Sicherheit von kryptographischen Hashfunktionen im Random Oracle Modell

Wir gehen von dem Szenario aus, daß wir die Hashfunktion h als eine "Black-Box" gegeben haben. Wir können uns also für beliebige Eingaben die Ausgaben von h berechnen lassen, haben aber keinerlei Zugriff auf Interna von h.

Für die Bewertung der folgenden Angriffe nehmen wir an, daß die Hashfunktion h eine Nachricht M auf einen zufällig gleichverteilten Hashwert H abbildet. Falls diese Annahme nicht zutreffen sollte, so wird die Erfolgswahrscheinlichkeit der Angriffe erhöht.

- **Einwegfunktion:** Ist eine Hashfunktion h eine Einwegfunktion, so soll es für einen Angreifer schwierig sein, zu einem gegebenen Hashwert H eine Nachricht M zu finden, mit $h(M) = H$. Wählt der Angreifer eine Nachricht M, so ist $h(M)$ mit Wahrscheinlichkeit $1/2^n$ gleich dem gesuchten H. Wir erwarten also, daß er $\Theta(2^n)$ Nachrichten testen muß.

- **schwach kollisionsresistent:** Eine Hashfunktion h mit dieser Eigenschaft soll es dem Angreifer schwierig machen, zu einer Nachricht M und einem Hashwert $H = h(M)$ eine andere Nachricht M' zu finden, mit $h(M) = H = h(M')$. Auch hier kann der Angreifer im allgemeinen nur Nachrichten wählen und testen, ob sie den gesuchten Hashwert H haben. Wieder muß der Angreifer $\Theta(2^n)$ Nachrichten prüfen.

- **stark kollisionsresistent** Diese Eigenschaft schränkt einen Angreifer nicht auf einen bestimmten Hashwert ein, sondern er kann beliebige verschiedene Nachrichten M, M' suchen, die denselben Hashwert $h(M) = h(M')$ haben. In diesem Fall ist ein sogenannter Geburtstagsangriff möglich. Der Name rührt von dem sogenannten Geburtstagsparadoxon [2] her, bei dem eine Lösung zu folgendem Problem gesucht wird:
  Wieviel Personen sind nötig, so daß mit Wahrscheinlichkeit $\geq 1/2$ zwei von ihnen am gleichen Tag Geburtstag haben ?
  Vorausgesetzt die Geburtstage der Personen sind zufällig gleichverteilt, genügen 23 Personen. Übertragen auf unser Problem der Kollisionssuche bedeutet dies, daß wir nur ca. $\sqrt{2^n}$ Nachrichten prüfen müssen. Der Angriff auf die starke Kollisionsresistenz ist also mit einem erwarteten Aufwand von $\Theta(2^{n/2})$ zu prüfenden Nachrichten deutlich schneller als Angriffe auf die anderen Eigenschaften und wurde erstmals 1979 von Yuval beschrieben [Yuv79].

Nach heutigem Stand der Dinge bilden 2^{64} Operationen die Grenze des gerade noch Machbaren. Das bedeutet, daß der Hashwert einer stark kollisionsresistenten Hashfunktion h mindestens die Länge 128 Bit haben sollte. Dem Mooreschen Gesetz zufolge erhöht sich die Geschwindigkeit der Rechner alle 3 Jahre um den Faktor 4. Diese Entwicklung bedeutet, daß ein Hashwert von 160 Bit ungefähr für die nächsten 24 Jahre Sicherheit bietet. Gerade wenn Hashfunktionen innerhalb von digitalen Signaturen eingesetzt werden, ist die Dauer der Gültigkeit der Unterschrift ein wichtiges Kriterium für die Länge des Hashwertes.

Aufbau praktischer Hashfunktionen Eine Hashfunktion h bildet eine Nachricht beliebiger Länge auf einen Hashwert H fester Länge n ab. Das gängige Verfahren in der Praxis ist, daß die Nachricht M zunächst in l Blöcke M_i der Länge m unterteilt wird und jeder Block mittels einer sogenannten Kompressionsfunktion f verarbeitet wird. Die Kompressionsfunktion f bildet Textblöcke der Länge m auf Textblöcke der Länge n ab, mit $m > n$. Sollte die Länge der Nachricht M kein Vielfaches von m sein, so werden am Ende eine entsprechende Anzahl Bits angehängt.[3] Eine Hashfunktion h wird nun nach folgendem Schema konstruiert:

$$\begin{aligned} H_0 &= IV \\ H_i &= f(M_i, H_{i-1}) \\ H &= H_l \end{aligned}$$

[2] Der Begriff "Paradoxon" soll nur andeuten, daß die Lösung dieses Problems zunächst nicht intuitiv ist.

[3] Hierbei ist einige Sorgfalt nötig, auf die wir hier aber nicht eingehen wollen.

Der Anfangswert H_0 bzw. IV ist ein öffentlich bekannter, fest gewählter Wert. Merkle und Damgård haben unabhängig voneinander in [Mer89] [Dam90] ein effi-

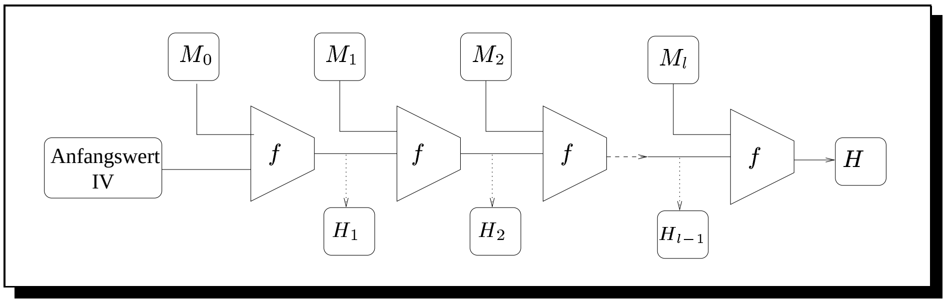

Abbildung 5.2.: Konstruktion einer Hashfunktion aus einer Kompressionsfunktion

zientes Verfahren vorgestellt, das es ermöglicht, aus einer stark kollisionsresistenten Kompressionsfunktion eine stark kollisionsresistente Hashfunktion zu konstruieren. Das Problem, eine sichere Hashfunktion zu generieren, ist also auf das Finden einer sicheren Kompressionsfunktion reduziert worden. Ähnlich wie bei den Verschlüsselungsverfahren, bei denen nicht bekannt ist, ob es überhaupt sichere effiziente Verfahren gibt, verhält es sich bei den Kompressionsfunktionen. Es ist nicht bekannt, ob es tatsächlich effiziente kollisionsresistente Kompressionsfunktionen und damit auch Hashfunktionen gibt. In der Praxis verfährt man deswegen genauso wie bei den Verschlüsselungsverfahren: Eine Hashfunktion gilt solange als kollisionsresistent bis das Gegenteil bzw. Indizien für das Gegenteil gefunden wurden.

5.2. Hashfunktionen basierend auf Blockchiffren

Im folgenden beschreiben wir einige Verfahren, wie man eine Blockchiffre einsetzen kann, um Hashfunktionen zu generieren.

Ein Vorteil, Hashfunktionen auf der Grundlage von Blockchiffren zu verwenden ist, daß oft die Blockchiffre bereits vorhanden ist und man die Implementierung wiederverwenden kann. Leider ist nicht bekannt, welche Eigenschaften eine Blockchiffre haben sollte, um einen guten Baustein einer Hashfunktion darzustellen. Es besteht nur die Hoffnung, daß sich die Sicherheit der Chiffre für die Verschlüsselung auch auf die Sicherheit der Hashfunktion überträgt. Der Vorteil, den die Verwendung des AES bedeutet, ist, daß der AES im Mittelpunkt der Forschung vieler Kryptoanalytiker steht und somit vermutlich besser untersucht wird als andere Blockchiffren.

Nachteilig wirkt sich die Verwendung von Blockchiffren auf die Effizienz der Hashfunktion im Vergleich zu speziell entwickelten Hashfunktionen aus. Ein Grund hierfür ist unter anderem, daß für jeden Block einer Nachricht ein Schlüsselwechsel vorgenommen werden muß.

Ein Maß für die Effizienz einer Hashfunktion auf Basis einer Blockchiffre ist die sogenannte **Hash-Rate**. Sie bezeichnet wieviele n Bit "Klartextblöcke" einer Nachricht M mit einer Verschlüsselung verarbeitet werden können. Je größer die Hash-Rate ist, um so effizienter ist die Hashfunktion.

In dieser Arbeit beschränken wir uns auf Schemata für Hashfunktionen, die einen Hashwert liefern, dessen Länge mit der Blocklänge der Blockchiffre übereinstimmt. Preneel, Govaerts und Vandewalle haben in [PGV94] alle 64 möglichen Variationen untersucht und kommen zu dem Ergebnis, daß 12 von ihnen keine bekannten Schwachstellen aufweisen. Zwei dieser Schemata, das sogenannte Davies-Meyer Schema und ein Schema von Matyas, Meyer und Oseas [MMO85] wollen wir im Zusammenspiel mit Rijndael untersuchen. Da die Blocklänge des AES aber auf 128 Bit festgelegt wurde, erhalten wir nur einen 128 Bit Hashwert; der Einsatz im Bereich der digitalen Signaturen ist also nicht zu empfehlen.

Im folgenden bezeichnen wir mit n die Blocklänge und mit k die Schlüssellänge der Chiffre. Wie zuvor bezeichnet H_i ein Zwischenergebnis der Hashfunktion, wobei H_0 gleich einem festen Anfangswert IV und $H_l = H$ gleich dem Hashwert ist.

5.2.1. Matyas-Meyer-Oseas Schema (MMO-Schema)

Dieses Schema wurde zuerst von S.M. Matyas, C.H. Meyer und J. Oseas im Jahre 1985 in [MMO85] veröffentlicht. Es ist als Standardschema zur Generierung von Hashfunktionen aus Blockchiffren im ISO Standard 10118-2 festgeschrieben.

Beschreibung des Schemas In diesem Schema wird eine Nachricht M zunächst in l Blöcke M_i, $(1 \leq i \leq l)$ der Länge n geteilt. Ein Block M_i geht als Klartextblock in die Chiffre ein. Die Funktion g bildet das vorherige Zwischenergebnis H_{i-1} der Länge n auf einen Wert der Länge k ab, der als Schlüssel in die Chiffre eingeht. Der resultierende Chiffretextblock wird mit dem Klartextblock per **XOR** verknüpft, um das neue Zwischenergebnis H_i zu berechnen:

$$H_i = \mathbf{E}_{g(H_{i-1})}(M_i) \oplus M_i$$

In Abbildung 5.3 ist der Ablauf schematisch dargestellt.

Dieses Schema hat eine Hash-Rate von 1, d.h. zum Hashen eines Nachrichtenblocks ist genau eine Verschlüsselung nötig.

Eine intuitive Begründung für die Sicherheit dieses Verfahrens ist folgende: Der Angreifer muß zu einem Schlüssel ein Klartext-Chiffretextpaar mit einer bestimmten Differenz (dem Hashwert) finden bzw. er muß 2 unterschiedliche Klartext-Chiffretextpaare finden, die dieselbe Differenz haben. Unter der Annahme, daß die

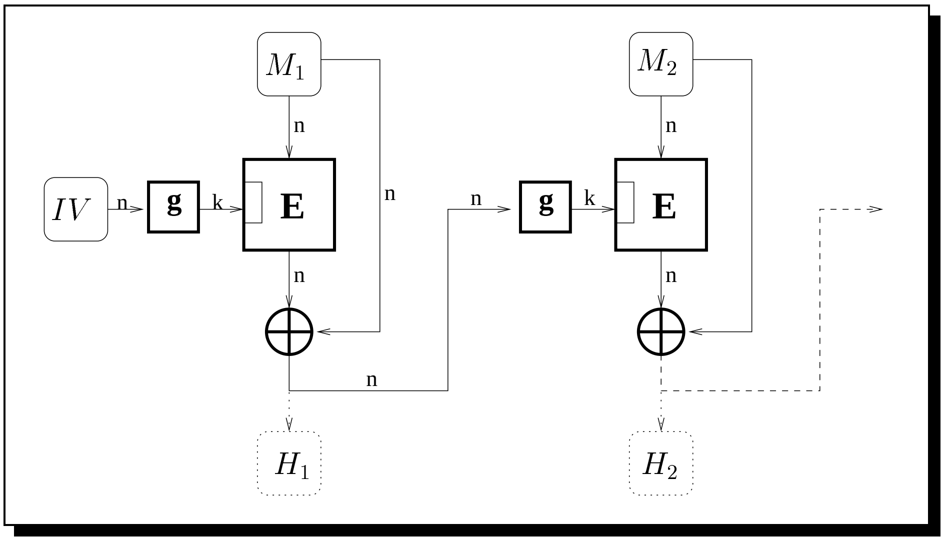

Abbildung 5.3.: Das Matyas-Meyer-Oseas Schema

Blockchiffre einen Klartextblock auf einen zufällig gleichverteilt gewählten Chiffretextblock abbildet, sollte dieses Vorgehen für einen Angreifer schwierig sein.

5.2.2. Davies-Meyer Schema (DM-Schema)

Das Davies-Meyer Schema [4] wird häufig auch als duales Schema zum MMO-Schema bezeichnet, denn hier tauschen Nachrichtenblock M_i und Zwischenergebnis H_{i-1} die Rollen.

Beschreibung des Schemas Die Nachricht M wird hier also in l Blöcke der Länge k aufgeteilt, die als Schlüssel in die Chiffre eingehen:

$$H_i = \mathbf{E}_{M_i}(H_{i-1}) \oplus H_{i-1}$$

Schematisch haben wir den Ablauf in Abbildung 5.4 dargestellt.

Die Hash-Rate dieses Schemas beträgt k/n. Eine Chiffre, deren Schlüssellänge k kleiner ist als die Blocklänge n führt also zu einer etwas langsameren Hashfunktion als beim MMO-Schema. Zum Beispiel hat DES eine Schlüssellänge von 56 Bit und eine Blocklänge von 64 Bit; eine Hashfunktion nach dem DM-Schema hat also eine Hash-Rate von 7/8. In unserem Fall allerdings hat der AES eine feste Blocklänge

[4]Das Schema wird zwar in der Literatur als Davies-Meyer bezeichnet, aber Davies hat es allein Meyer zugeschrieben

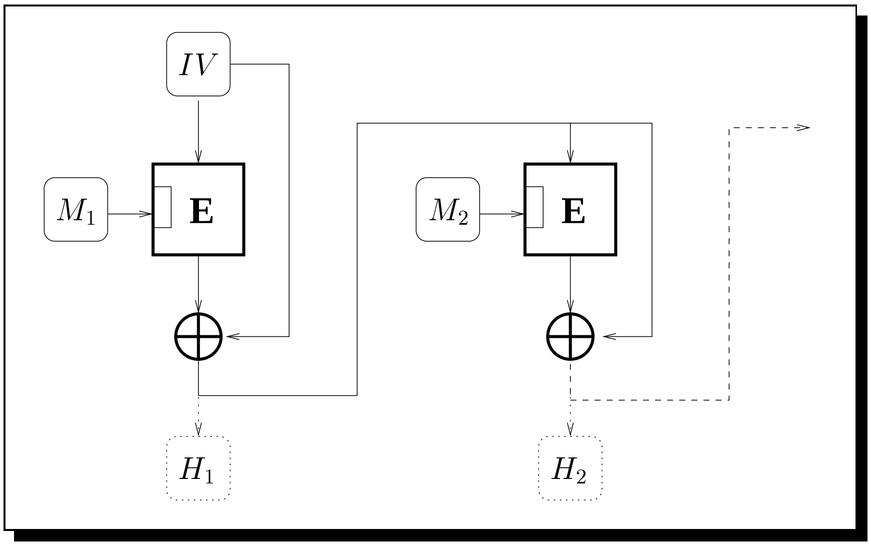

Abbildung 5.4.: Das Davies-Meyer Schema

von 128 Bit und mögliche Schlüssellängen von 128, 192 und 256 Bit, so daß Hash-Raten von 1, 1.5 und 2 erreicht werden. Ein weiterer Vorteil des DM-Schemas in unserem Fall ist, daß wir keine Funktion g benötigen, um die Schlüsseleingaben der Chiffre auf die richtige Länge zu bringen.

Die Sicherheit können wir auch hier nur intuitiv beschreiben. Ein Angreifer muß zu einem Klartext einen Schlüssel finden, so daß das berechnete Klartext-Chiffretextpaar eine bestimmte Differenz, nämlich den Hashwert hat. Setzen wir wieder voraus, daß die Blockchiffre einen Klartextblock auf einen zufällig gleichverteilt gewählten Chiffretextblock abbildet, so sollte dieses Problem für einen Angreifer schwierig zu lösen sein.

5.3. Untersuchungen zur starken Kollisionsresistenz

In diesem Abschnitt betrachten wir die beiden vorgestellten Schemata mit einer reduzierten Version von Rijndael. Dabei skizzieren wir zunächst nur einfache Ansätze zur Kollisionssuche bei Nachrichten, die nur aus einem Block bestehen, Im letzten Teil betrachten wir dann einen Unterscheider, der 4 Runden von einer Zufallsfunktion unterscheiden kann [GM00]. Ein Einsatz von 4 Runden Rijndael in der Kompressionsfunktion ist also nicht empfehlenswert, weil hier unsere Anforderung an die Blockchiffre, einen Klartextblock auf einen zufällig gleichverteilt gewählten Chiffretextblock abzubilden, nicht mehr zutrifft.

5.3.1. Das MMO-Schema

Betrachten wir zunächst das MMO-Schema mit einer Blockchiffre, die nur aus einer Runde Rijndael besteht. Da diese Runde auch gleichzeitig die letzte Runde ist, wird kein MixColumn ausgeführt. Dies führt dazu, daß keinerlei Diffusion in der Verschlüsselung auftritt, jedes Byte des Texblocks beeinflußt also nur ein Byte des Hashwertes. Seien M, M' zwei unterschiedliche Nachrichten, die jeweils nur aus einem Block bestehen. Der Anfangswert IV geht in diesem Schema als Schlüssel in die Chiffre ein. Der erste Rundenschlüssel ist IV selbst und mit IV' bezeichnen wir den zweiten Rundenschlüssel, den der KeySchedule erzeugt.

Der Hashwert H der Nachricht M berechnet sich also wie folgt:

$$H = \mathbf{S}[M \oplus IV] \oplus IV' \oplus M.$$

Es reicht in diesem einfachen Fall aus, nur eine Kollision in einem Byte zu suchen. Wir suchen also der Einfachheit halber zwei Nachrichten M, M', die sich nur in einem Byte unterscheiden und beziehen die folgende Rechnung auf dieses Byte:

$$\begin{aligned} \mathbf{S}[M \oplus IV] \oplus IV' \oplus M &= \mathbf{S}[M' \oplus IV] \oplus IV' \oplus M' \\ \Leftrightarrow \mathbf{S}[M \oplus IV] \oplus M &= \mathbf{S}[M' \oplus IV] \oplus M' \end{aligned}$$

Vereinfachen wir dieses Beispiel weiter, indem wir den $IV = 0$ setzen, so erhalten wir folgende Gleichung:

$$\mathbf{S}[M] \oplus M = H = \mathbf{S}[M'] \oplus M' \tag{5.1}$$

Es ist zunächst einmal nicht klar, ob diese Gleichung eine Lösung hat bzw. wie viele Lösungen existieren. Da wir aber nur ein Byte prüfen, können wir alle Lösungen leicht per vollständiger Suche finden. In Tabelle 5.1 haben wir alle Lösungen aufgelistet. Für 72 der insgesamt 256 möglichen Werte eines Bytes existieren Lösungen von Gleichung 5.1. Wir können also durch einfaches Probieren relativ schnell eine Kollision erzeugen.

Betrachten wir nun analog das MMO-Schema mit 2 Runden Rijndael, einer vollen ersten Runde und einer verkürzten zweiten Runde ohne MixColumn. Wegen des MixColumns beeinflussen 4 Bytes des Nachrichtenblocks ein Byte des Hashwertes, so daß wir zum Finden einer Kollision je 2^{32} Möglichkeiten von M bzw. M' prüfen müssen. Bereits beim Einsatz von 3 Runden Rijndael beeinflußt jedes Byte des Nachrichtenblocks jedes Byte des Hashwertes, so daß dieser Ansatz zur Kollisionssuche schon den maximalen Aufwand von ca. $\sqrt{2^{128}}$ Überprüfungen nötig macht.

H	M, M'				H	M, M'			
1	199	224			144	95	105	135	
6	150	170	94		147	17	161		
16	145	190			151	32	142		
17	50	93			152	99	162		
18	36	133			156	74	212		
20	107	201			160	122	208		
26	37	61	72		163	53	165		
28	40	70	221		169	85	163	253	
31	46	151			173	127	55		
32	11	200	214		180	27	246		
34	154	236			181	24	211		
43	57	69			182	139	148		
50	88	114			184	26	237		
56	8	173			185	29	193	216	248
60	108	229			190	83	202		
69	206	207	254		194	7	65	134	
74	63	174			195	157	222		
76	21	136	252		199	111	189		
78	118	186			207	42	121		
81	22	109	187		212	184	203		
90	123	220			214	175	213		
96	181	249			215	60	250		
98	91	219			217	59	188	215	
99	0	197			218	13	16	43	
104	106	230			219	18	132		
108	30	124			222	152	179		
109	10	234			231	23	71	86	194
110	5	19	66		232	117	140		
114	73	198			237	164	195		
115	231	232			239	185	209		
121	15	177			240	51	144		
122	169	192	226		242	12	227		
128	28	81			244	138	251		
130	119	125	166		246	4	49		
140	41	62			248	75	182		
141	126	129	218	228	252	115	143		

Tabelle 5.1.: Lösungen M, M' der Gleichung $\mathbf{S}[M] \oplus M = H = \mathbf{S}[M'] \oplus M'$

5.3.2. Das DM-Schema

Beim DM-Schema und Nachrichten, die nur aus einem Block bestehen, müssen wir Lösungen für folgende Gleichung finden

$$\begin{aligned} E_M(IV) \oplus IV = \quad & H & = E_{M'}(IV) \oplus IV \\ \Leftrightarrow E_M(IV) \quad & = & E_{M'}(IV). \end{aligned}$$

Das heißt, wir müssen zu einem festen Klartext IV zwei Schlüssel M, M' finden, so daß der Klartext unter beiden Schlüsseln auf den gleichen Chiffretext abgebildet wird. Dies ist das sogenannte Problem der Schlüsselkollisionen. Bei einer Blockchiffre kann es passieren, daß ein Text unter verschiedenen Schlüsseln auf den gleichen Chiffretext abgebildet wird. Bei einer guten Chiffre sollte die Anzahl dieser Kollisionen aber gering sein und kein effizientes Verfahren sollte es einem Angreifer ermöglichen, diese zu berechnen.

Da wir in unseren ersten Versuchen, konkrete Kollisionen zu finden, nicht über 2 Runden Rijndael hinauskamen, verfolgen wir nun einen anderen Ansatz. Wir hatten die Anforderungen an die Blockchiffre gestellt, daß sie einen Klartext auf einen zufällig gleichverteilt gewählten Chiffretext abbilden muß. Wir zeigen nun, daß Rijndael mit 3 bzw. 4 Runden diese Eigenschaft definitiv nicht erfüllt. Dazu geben wir sogenannte Unterscheider an, die mit Wahrscheinlichkeit $> 1/2$ die Ausgaben von Rijndael von zufällig gleichverteilt gewählten Ausgaben unterscheiden können.

Da unsere Anforderung an die Blockchiffre nicht erfüllt sind, so können wir in diesem Fall über die Sicherheit der Hashfunktionen, die auf den beiden Schemata beruhen, keinerlei Aussage treffen.

5.3.3. Pseudozufall und das Konzept der Unterscheider

In der Kryptographie benötigt man an vielen Stellen Zufallszahlen, wie z.B. bei der Generierung von Schlüsseln. Da "echte" Zufallszahlen nur sehr aufwendig zu erzeugen sind, setzt man in der Praxis sogenannte Pseudozufallsbitgeneratoren (PZBG) ein. Ein PZBG ist ein Algorithmus, der als Eingabe einen kurzen zufälligen Bitstring als Startwert (engl. seed) bekommt und als Ausgabe einen langen "zufällig aussehenden" Bitstring erzeugt. Der PZGB reduziert also die benötigte Menge an "echten" zufälligen Bits. Der Grundgedanke hinter dem Prinzip der PZBG's ist, daß man die Ausgabe nicht effizient von "echtem" Zufall unterscheiden kann. Das Maß für "echten" Zufall ist die uniforme Wahrscheinlichkeitsverteilung u_n, bei der jeder Bitstring der Länge n mit der gleichen Wahrscheinlichkeit $1/(2^n)$ gewählt wird. Es soll also keinen Algorithmus geben, der in Polynomialzeit einen pseudozufälligen Bitstring von einem nach der uniformen Wahrscheinlichkeitsverteilung u_n, gleichmäßig zufällig ausgewählten Bitstring unterscheidet.

Definition 5.3.1 (Unterscheider) *Seien p_0 und p_1 zwei Wahrscheinlichkeitsverteilungen. Ein Unterscheider ist ein probabilistischer Polynomialzeit Algorithmus $\mathcal{A}$, der als Eingabe einen Bitstring erhält und als Ausgabe die Entscheidung liefert, ob die Eingabe von der Wahrscheinlichkeitsverteilung p_0 oder von p_1 kommt.*

Der Erwartungswert von $\mathcal{A}$ bzgl. der Wahrscheinlichkeitsverteilung p ist definiert als

$$E_{\mathcal{A}}(p) = \sum_{x \in \mathbb{Z}_2^l} p(x) \cdot Pr(\mathcal{A}(x) = 1).$$

Die Wahrscheinlichkeit $Pr(\mathcal{A}(x) = 1)$ hängt von den internen Münzwürfen des probabilistischen Algorithmus $\mathcal{A}$ ab. $\mathcal{A}$ heißt ϵ-Unterscheider von p_0 und p_1, falls für ein $\epsilon > 0$ gilt:

$$|E_{\mathcal{A}}(p_0) - E_{\mathcal{A}}(p_1)| \geq \epsilon$$

p_0 und p_1 heißen in diesem Fall ϵ-unterscheidbar.

Ein PZBG definieren wir also wie folgt:

Definition 5.3.2 (Pseudozufallsbitgenerator) *Seien $k, l \in \mathbb{N}$ mit $l > k$. Ein (k, l) Pseudozufallsbitgenerator ist ist eine Abbildung von $\mathbb{Z}_2^k$ nach $\mathbb{Z}_2^l$, die in polynomieller Zeit berechnet werden kann und für die es keinen Unterscheider $\mathcal{A}$ gibt, der die Ausgabe von der uniformen Wahrscheinlichkeitsverteilung mit Wahrscheinlichkeit $> 1/2$ unterscheiden kann. Die Eingabe $s \in \mathbb{Z}_2^k$ nennen wir Seed und die Ausgabe $z \in \mathbb{Z}_2^l$ nennen wir pseudozufälliger Bitstring.*

Im folgenden werden wir Unterscheider für Rijndael mit 3 bzw. 4 Runden angeben. Damit zeigen wir, daß sich Rijndael mit nur 3 bzw. 4 Runden nicht zur Generierung einer sicheren Hashfunktion anhand der beiden Schemata eignet.

5.3.4. Ein Unterscheider für 3 Runden Rijndael

Die Idee für einen Unterscheider zwischen der Normalverteilung und 3 Runden Rijndael beruht auf der gleichen Idee, die auch zum SQUARE-Angriff geführt hat. Verschlüsseln wir eine Λ-Menge 3 Runden, ist ihre Summe an jeder Byteposition 0.

In unserem Modell existiert ein Generator, der als Eingabe eine Λ-Menge erhält und der dann jeweils mit Wahrscheinlichkeit 1/2 Rijndael mit 3 Runden anwendet oder die Ausgabe zufällig gleichverteilt auswählt. Unser Unterscheider muß nur anhand der Ausgabe des Generators entscheiden, welche Wahl der Generator getroffen hatte. Mit einer Wahrscheinlichkeit $> 1/2$ sollte die Antwort korrekt sein.

Der Unterscheider erhält als Eingabe die Ausgabe des Generators ohne jedoch seine interne Entscheidung zu kennen. Er berechnet das **XOR** an allen Bytepositionen und antwortet "Ausgabe wahrscheinlich zufällig gleichverteilt" falls nicht alle Ergebnisse 0 waren und "Ausgabe wahrscheinlich von 3 Runden Rijndael" falls alle Ergebnisse 0 waren.

Analyse des Unterscheiders Im folgenden wollen wir die Güte des Unterscheiders bestimmen, d.h. mit welcher Wahrscheinlichkeit der Unterscheider die korrekte Antwort liefert.

Wählt der Generator Rijndael mit 3 Runden, so liefert unser Unterscheider immer die richtige Antwort.(Das **XOR** einer Λ-Menge nach 3 Runden Verschlüsselung ist immer 0.) Wählt der Generator aber die Gleichverteilung, so erzeugt dieser mit Wahrscheinlichkeit $1/256$ an einer Position bei einer Verknüpfung per **XOR** das Ergebnis 0. Da unser Unterscheider aber alle 16 Positionen prüft, beträgt die Wahrscheinlichkeit für eine falsche Antwort nur $(1/256)^{16}$ bzw. $1-(1/256)^{16}$ für die richtige Antwort.

Der Unterscheider liefert also insgesamt eine richtige Antwort mit Wahrscheinlichkeit

$$\frac{1}{2}\cdot 1+\frac{1}{2}\cdot\left(1-\left(\frac{1}{256}\right)^{16}\right)=1-\frac{1}{2}\cdot\left(\frac{1}{256}\right)^{16}.$$

Die Güte des Unterscheiders bezüglich der Gleichverteilung ist also

$$\epsilon=\left|\frac{1}{2}-\frac{1}{2}\cdot\left(\frac{1}{256}\right)^{16}\right|.$$

5.3.5. Ein Unterscheider für 4 Runden Rijndael

Bei Eingabe einer geeigneten Menge von Klartexten können wir nach 4 vollen Runden Rijndael mit relativ hoher Wahrscheinlichkeit in der Menge der Chiffretexte Kollisionen erzeugen, die uns ein Kriterium für einen Unterscheider liefern. Hierzu wird die Rijndael-Verschlüsselung als Abbildung von einem Byte im Klartext auf ein Byte im Chiffretext verstanden, die wir durch Wahl von 3 Parameter-Bytes beeinflussen können. Da diese Abbildung von relativ wenigen unbekannten Bytes abhängt, liefert diese Konstruktion einen effizienten Unterscheider [GM00].

Zur Beschreibung der einzelnen Schritte verwenden wir wieder die Notation aus dem Kapitel über die Kryptoanalyse. Dabei bezeichnen wir den Zustand in Runde i nach dem ByteSub mit B^i, nach dem ShiftRow mit R^i, nach dem MixColumn mit M^i und nach dem KeyAdd mit A^i. Den Zustand zu Beginn einer Verschlüsselung bezeichnen wir mit P.

Idee für den Unterscheider Betrachte eine Menge von Klartexten, in denen $P_{0,0} =: y$ verschiedene Werte annimmt und alle anderen Bytes konstant sind. Die

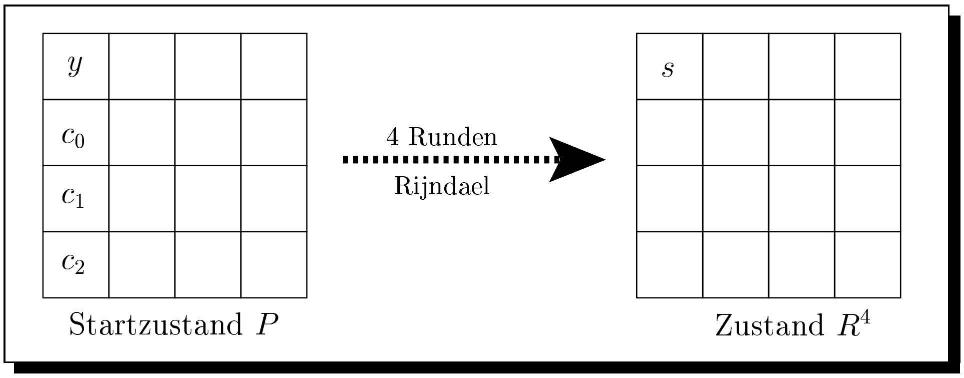

Abbildung 5.5.: Startzustand und Zustand vor dem MixColumn in Runde 4

Konstanten c_0, c_1, c_2 in Spalte 0 können wir passend wählen / variieren, aber alle anderen Bytes bleiben fest (siehe Abb. 5.5). Verfolgt man nun die Änderungen am Startzustand in den einzelnen Transformationen, so erhält das Byte $s := M^4_{0,0}$ eine besondere Bedeutung. Versteht man die Änderungen an den Zuständen nämlich als Abbildung $f(y) = s$, so kann man das Tripel $c := (c_0, c_1, c_2)$ als Parameter von $f(y) = f^c(y)$ interpretieren. D.h. durch Wahl von 3 Parametern c_0, c_1, c_2 können wir die Funktion $f^c(y)$ beeinflussen. Im folgenden Satz zeigen wir, daß die Funktion f^c nur von sehr wenigen unbekannten Bytes abhängt. Diese Tatsache ist entscheidend für die Effizienz des Unterscheiders.

Satz 5.3.1 *Die Funktion f^c hängt lediglich von 5 unbekannten Bytes ab, die nur vom Schlüssel abhängen, und 4 unbekannten Bytes $d_0 \ldots d_3$, die sowohl vom Schlüssel als auch von c abhängen.*

Beweis. Im folgenden entwickeln wir die Funktion $f^c(y)$ aus den jeweiligen Runden-Transformationen und bestimmen in jedem Schritt die Anzahl der unbekannten Bytes von denen die weiteren Transformationen abhängig sind. Man kann diese Bytes danach unterscheiden, ob sie nur vom Schlüssel oder auch von c abhängen. Der Einfachheit halber lassen wir das Key-Add zu Beginn beiseite, weil es auf die Konstruktion keinerlei Einfluß hat. Die Bytes, auf die wir uns um folgenden beziehen, erhalten besondere Bezeichnungen wie in Abbildung 5.6 dargestellt.

Runde 1: Sei P der Startzustand mit den besonders gekennzeichneten Bytes $y := P_{0,0}$, $c_0 := P_{1,0}$, $c_1 := P_{2,0}$, $c_2 := P_{3,0}$ und A^1 der Zustand nach dem KeyAdd in Runde 1 mit den Bytes $z_0 := A^1_{0,0}$, $z_1 := A^1_{1,0}$, $z_2 := A^1_{2,0}$ und $z_3 := A^1_{3,0}$, wie in Abb. 5.6 dargestellt. Die Abbildungen $y \to z_i, i \in \{0, \ldots, 3\}$ hängen von je einem unbekannten Byte ab, welches wiederum von je einem Schlüsselbyte abhängt, nicht jedoch von c. Dies folgt aus der Anwendung

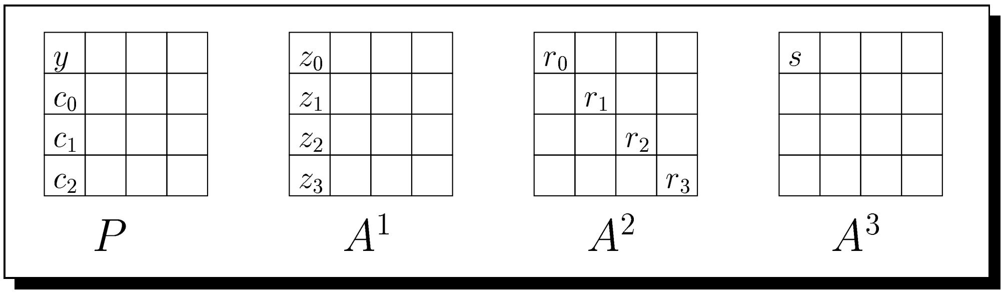

Abbildung 5.6.: Zustände zu Beginn jeder Runde

der ShiftRow Transformation vor dem MixColumn. Durch Anwendung des ShiftRows werden die c_0, c_1, c_2 jeweils auf die 3 anderen Spalten verschoben. Dadurch können sie im nachfolgenden MixColumn keinerlei Einfluß auf die Bytes $z_0, \dots, z_3$ nehmen.

Formal beschrieben: Seien $M_{i,j}$ mit $i, j \in \{0, \dots, 3\}$ die Koeffizienten aus der MDS-Matrix M des MixColumn. Dann lassen sich die z_i schreiben als

$$z_i = A^1_{i,0} = M_{i,0} \cdot \mathbf{S}[y] \oplus \underbrace{b_{i,0}}_{konst}$$

mit Konstanten

$$b_{i,0} = M_{i,1} \cdot \underbrace{\mathbf{S}[P_{1,1}]}_{konst} \oplus M_{i,2} \cdot \underbrace{\mathbf{S}[P_{2,2}]}_{konst} \oplus M_{i,3} \cdot \underbrace{\mathbf{S}[P_{3,3}]}_{konst} \oplus K^1_{i,0}$$

Alle anderen Bytes hängen zusätzlich zum Schlüssel auch noch von c ab. Für $i = 1 \dots 3$ und $j = 0 \dots 3$ gilt:

$$A^1_{i,j} = M_{i,j} \cdot \mathbf{S}[c_{j-1}] \oplus \underbrace{b_{i,j}}_{konst}$$

Runde 2: Betrachten wir nun die Abbildungen von $z_i \to r_i \; \forall i \in \{0, \dots, 3\}$ (vgl. Abb. 5.6). Das ShiftRow in Runde 2 verteilt die z_i auf die 4 Spalten. An den übrigen Positionen befinden sich Bytes, die in Runde 1 in Abhängigkeit von c und dem Rundenschlüssel gebildet wurden. Im nun folgenden MixColumn werden die 4 Bytes einer Spalte linear verknüpft:

$$r_0 = M_{0,0} \cdot \mathbf{S}[z_0] \oplus \underbrace{M_{0,1} \cdot \mathbf{S}[A^1_{1,1}] \oplus M_{0,2} \cdot \mathbf{S}[A^1_{2,2}] \oplus M_{0,3} \cdot \mathbf{S}[A^1_{3,3}] \oplus K^2_{0,0}}_{d_0}$$

$$r_1 = M_{1,3} \cdot \mathbf{S}[z_3] \oplus \underbrace{M_{1,0} \cdot \mathbf{S}[A^1_{0,1}] \oplus M_{1,1} \cdot \mathbf{S}[A^1_{1,2}] \oplus M_{1,2} \cdot \mathbf{S}[A^1_{2,3}] \oplus K^2_{1,1}}_{d_1}$$

$$r_2 = M_{2,2} \cdot \mathbf{S}[z_2] \oplus \underbrace{M_{2,0} \cdot \mathbf{S}[A^1_{0,2}] \oplus M_{2,1} \cdot \mathbf{S}[A^1_{1,3}] \oplus M_{2,3} \cdot \mathbf{S}[A^1_{3,1}] \oplus K^2_{2,2}}_{d_2}$$

$$r_3 = M_{3,1} \cdot \mathbf{S}[z_1] \oplus \underbrace{M_{3,0} \cdot \mathbf{S}[A^1_{0,3}] \oplus M_{3,2} \cdot \mathbf{S}[A^1_{2,1}] \oplus M_{3,3} \cdot \mathbf{S}[A^1_{3,2}] \oplus K^2_{3,3}}_{d_3}$$

Dabei fassen wir jede Linearkombination der 3 Bytes, die von c abhängen zu je einer Unbekannten $d_0, \ldots, d_3$, die von c abhängt, zusammen.

Daraus ergibt sich also, daß jedes der $r_0, \ldots, r_3$ von je einem der $z_0, \ldots, z_3$ und zusätzlich von je einer Unbekannten $d_0, \ldots, d_3$ abhängt. D.h. wir haben Abhängigkeiten von 4 Unbekannten, die nur von dem Rundenschlüssel abhängen und von 4 Unbekannten, die außer vom Rundenschlüssel auch noch von c abhängen.

Runde 3: In Runde 3 betrachten wir nun den Zusammenhang zwischen $r_0 \ldots r_3$ und dem s-Byte. Die Positionen der r-Bytes wurden so gewählt, daß sie nach dem ShiftRow in R^3 in der Spalte des s-Bytes landen. Also ergibt sich das s aus einer einfachen Linearkombination der r-Bytes während des MixColumns:

$$s = M_{0,0} \cdot \mathbf{S}[r_0] \oplus M_{0,1} \cdot \mathbf{S}[r_1] \oplus M_{0,2} \cdot \mathbf{S}[r_2] \oplus M_{0,3} \cdot \mathbf{S}[r_4] \oplus \underbrace{K^3_{0,0}}_{e}$$

Das zusätzliche unbekannte Byte e erhöht die Zahl der Bytes auf 5 schlüsselabhängige und 4 c-abhängige Bytes und der Beweis des Satzes ist damit erbracht.

□

Für das weitere Vorgehen treffen wir die Annahme, daß sich die 4 c-abhängigen Bytes $d_0 \ldots d_3$ wie zufällig gleichverteilt verhalten, d.h. durch eine Wahl von c legen wir uns auf ein f^c aus 2^{32} möglichen “zufällig” fest. Bei einer Stichprobe von 2^{16} c-Werten erzeugen wir wegen des Geburtstagsparadoxons mit einer signifikanten Wahrscheinlichkeit eine Kollision. Eine Kollision bedeutet hier also, daß zwei verschiedene c, c' zwei Funktionen $f^c(y), f^{c'}(y)$ erzeugen, die für alle gewählten y

den gleichen Funktionswert berechnen. Da wir 3 c-Bytes wählen können, können wir also maximal 2^{24} verschiedene $f^c(y)$ wählen, um die Wahrscheinlichkeit für eine Kollision zu erhöhen. Diese Kollisionen bilden nun die Grundlage für einen effizienten Unterscheider, für den wir die Fehlerwahrscheinlichkeit noch durch eine geeignet große Anzahl von y-Werten (z.B. 16) verbessern können.

Der Generator Wir verwenden hier wieder das Modell, daß ein Generator eine Menge von Klartexten als Eingabe erhält und er jeweils mit Wahrscheinlichkeit 1/2 entweder 4 Runden Rijndael darauf anwendet oder das Ergebnis zufällig gleichverteilt auswählt.

Der Unterscheider bekommt als Eingabe die Ausgabe des Generators ohne die Information, mit welchem Verfahren der Generator gearbeitet hat. Er wendet zunächst ein inverses MixColumn auf die Spalte 0 mit den Bytes (t_0, t_1, t_2, t_3) an:

$$s' = 0E \cdot t_0 \oplus 0B \cdot t_1 \oplus 0D \cdot t_2 \oplus 09 \cdot t_3 \tag{5.2}$$

Um aus dem Ergebnis s' nun das s zu berechnen, müssten wir ein Byte des vierten Rundenschlüssels K^4 raten. Da dieses unbekannte Byte aber nicht von c abhängt, ist die Änderung für jedes $f^c(y)$ gleich und beeinträchtigt unseren Test nicht. Anders gesagt kann man die Gleichung 5.2 auch als Abbildung von y verstehen, die ebenfalls von 4 c-abhängigen Bytes und zusätzlich von 6 schlüsselabhängigen Bytes abhängt. In unserem Fall sind aber nur die c-abhängigen Unbekannten relevant.

Der Unterscheider ordnet die Werte in einer Tabelle an: Für jeden c-Wert und jeden y-Wert berechnet er alle Funktionswerte $f^c(y) = s'$ mittels Gleichung 5.2. Dann prüft er, ob es in der Tabelle zwei gleiche Zeilen gibt. Falls ja, so ist dies eine Kollision, die wahrscheinlich aus einer 4 Runden Rijndael-Verschlüsselung stammt. Falls nein, so stammen die Werte vermutlich nicht aus einer Rijndael-Verschlüsselung.

Algorithmus 5.3.1 (Unterscheider)

1. *Wähle eine Menge von c-Werten aus $C \subseteq \{0, \ldots, 255\}^3$ und eine Menge von y-Werten aus $Y \subseteq \{0, \ldots, 255\}$*

2. *Berechne für jedes c und jedes y den Funktionswert $s = f^c(y)$ und speichere ihn in einer Tabelle*

3. *Suche Kollision in der Tabelle*

   a) *Falls Kollision gefunden: Gib Antwort "Wahrscheinlich von 4 Runden Rijndael erzeugt" zurück* *b) Falls keine Kollision gefunden: Gib Antwort "Wahrscheinlich zufällig gleichverteilt" zurück*

Analyse des Unterscheiders Um die Güte des Unterscheiders zu ermitteln, betrachten wir die Fälle, in denen der Unterscheider eine falsche Antwort liefert und berechnen die Wahrscheinlichkeiten für ihr Auftreten. Mit $\#c$ bezeichnen wir die Anzahl der gewählten c-Werte und mit $\#y$ die Anzahl der gewählten y-Werte.

Fall 1: Der Generator wählt Rijndael reduziert auf 4 Runden, aber der Unterscheider findet keine Kollision.
Wie in Satz 5.3.1 gezeigt, kann Rijndael mit 4 Runden nur 2^{32} verschiedene Funktionen $f^c(y)$ erzeugen. werden. Nehmen wir für $\#y$ den maximalen Wert von 256 an, so erhalten wie eine obere Schranke für die Fehlerwahrscheinlichkeit. Die Wahrscheinlichkeit, daß bei $\#c$ vielen Versuchen keine Kollision auftritt, ist also kleiner gleich

$$\frac{2^{32}!}{(2^{32}-\#c)!\cdot(2^{32})^{\#c}}$$

Wählen wir 2^{16} c-Werte, so ergibt sich eine Wahrscheinlichkeit von ca. $0,6$.

Fall 2: Der Generator wählt die Gleichverteilung, erzeugt aber zufällig eine Kollision.
Wählt der Generator die Zeilen zufällig gleichverteilt, so sind $256^{\#y}$ verschiedene Zeilen in der Tabelle möglich. Es werden aber nur $\#c$ viele Zeilen erzeugt, so daß die Wahrscheinlichkeit für eine Kollision gleich

$$1-\frac{256^{\#y}!}{(256^{\#y}-\#c)!\cdot(256^{\#y})^{\#c}}$$

ist. Für unsere Beispielwerte $\#y = 16$ und $\#c = 2^{16}$ liegt die Wahrscheinlichkeit nahe bei 0.

Da der Generator zwischen beiden Algorithmen mit Wahrscheinlichkeit $\frac{1}{2}$ auswählt, liefert der Unterscheider mit Wahrscheinlichkeit

$$\frac{1}{2}\cdot\left(\frac{2^{32}!}{(2^{32}-\#c)!\cdot(2^{32})^{\#c}}+1-\frac{256^{\#y}!}{(256^{\#y}-\#c)!\cdot(256^{\#y})^{\#c}}\right)$$

die richtige Antwort. In unserem Beispiel mit $\#y = 16$ und $\#c = 2^{16}$ liefert der Unterscheider mit Wahrscheinlichkeit von ca. $0,7$ die richtige Antwort. Der Unterscheider benötigt in diesem Fall den Platz für 2^{20} Texte und die Zeit für ca. 2^{20} Rijndael Verschlüsselungen.

Über die Anzahl der verwendeten c-Werte kann man die Fehlerwahrscheinlichkeit im Fall 1 beeinflussen. Je mehr c-Werte (max. 2^{24}) man testet, umso größer wird die Wahrscheinlichkeit, eine Kollision zu erzeugen, aber umso aufwendiger wird der Unterscheider. Ebenso verhält es sich bei der Wahl der Anzahl der y-Werte und der Fehlerwahrscheinlichkeit im Fall 2. Durch das Testen von mehr y-Werten (max. 256) verringert man die Wahrscheinlichkeit, daß der Generator die Gleichverteilung wählt und zufällig eine Kollision erzeugt.

A. Tabellen

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	99	124	119	123	242	107	111	197	48	1	103	43	254	215	171	118
16	202	130	201	125	250	89	71	240	173	212	162	175	156	164	114	192
32	183	253	147	38	54	63	247	204	52	165	229	241	113	216	49	21
48	4	199	35	195	24	150	5	154	7	18	128	226	235	39	178	117
64	9	131	44	26	27	110	90	160	82	59	214	179	41	227	47	132
80	83	209	0	237	32	252	177	91	106	203	190	57	74	76	88	207
96	208	239	170	251	67	77	51	133	69	249	2	127	80	60	159	168
112	81	163	64	143	146	157	56	245	188	182	218	33	16	255	243	210
128	205	12	19	236	95	151	68	23	196	167	126	61	100	93	25	115
144	96	129	79	220	34	42	144	136	70	238	184	20	222	94	11	219
160	224	50	58	10	73	6	36	92	194	211	172	98	145	149	228	121
176	231	200	55	109	141	213	78	169	108	86	244	234	101	122	174	8
192	186	120	37	46	28	166	180	198	232	221	116	31	75	189	139	138
208	112	62	181	102	72	3	246	14	97	53	87	185	134	193	29	158
224	225	248	152	17	105	217	142	148	155	30	135	233	206	85	40	223
240	140	161	137	13	191	230	66	104	65	153	45	15	176	84	187	22

Tabelle A.1.: S-Box

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	82	9	106	213	48	54	165	56	191	64	163	158	129	243	215	251
16	124	227	57	130	155	47	255	135	52	142	67	68	196	222	233	203
32	84	123	148	50	166	194	35	61	238	76	149	11	66	250	195	78
48	8	46	161	102	40	217	36	178	118	91	162	73	109	139	209	37
64	114	248	246	100	134	104	152	22	212	164	92	204	93	101	182	146
80	108	112	72	80	253	237	185	218	94	21	70	87	167	141	157	132
96	144	216	171	0	140	188	211	10	247	228	88	5	184	179	69	6
112	208	44	30	143	202	63	15	2	193	175	189	3	1	19	138	107
128	58	145	17	65	79	103	220	234	151	242	207	206	240	180	230	115
144	150	172	116	34	231	173	53	133	226	249	55	232	28	117	223	110
160	71	241	26	113	29	41	197	137	111	183	98	14	170	24	190	27
176	252	86	62	75	198	210	121	32	154	219	192	254	120	205	90	244
192	31	221	168	51	136	7	199	49	177	18	16	89	39	128	236	95
208	96	81	127	169	25	181	74	13	45	229	122	159	147	201	156	239
224	160	224	59	77	174	42	245	176	200	235	187	60	131	83	153	97
240	23	43	4	126	186	119	214	38	225	105	20	99	85	33	12	125

Tabelle A.2.: inverse S-Box

word32 Rcon[30] = {
0x01, 0x02, 0x04, 0x08, 0x10, 0x20, 0x40, 0x80, 0x1b, 0x36,
0x6c, 0xd8, 0xab, 0x4d, 0x9a, 0x2f, 0x5e, 0xbc, 0x63, 0xc6,
0x97, 0x35, 0x6a, 0xd4, 0xb3, 0x7d, 0xfa, 0xef, 0xc5, 0x91 };

Tabelle A.3.: Rundenkonstante Rcon

B. Sourcecode der Implementierung

```
// Original−Angriff mit Teilsummenverbesserung
// Speicherung der CT,s in Bits ->512 MB
// Stufe1 geteilt (k0,k1 extra)
// Hauptspeicherbedarf ca 1,1 GB !!!!!

typedef unsigned char u8;
typedef unsigned short u16;
typedef unsigned int u32;
typedef long long int  u64;
#define Bitsinu8 8
#define numberplain 4294967296  //2^32 Chiffretexte

#include <stdio.h>
#include <time.h>
#include"msboxes.c" //Einbinden der S-Boxen
#include <sys/time.h>
#include <sys/utsname.h>

//#define DEBUG // Debug-Ausgaben anschalten

static u32 x0[numberplain/32];//speichere X0
static u32 x1[numberplain/256];  //
static u8 x2[numberplain/256/256]; //64KB
static u8 x3[numberplain/256/256/256]; // 256 Bytes
static u8 x4;
int startk0,endk0;
int startk1,endk1;
int startk2,endk2;
int startk3,endk3;
int startk4,endk4;
int k0,k1,k2,k3,k4; //Schluesselbytes
```

```
unsigned long long int c_count=0,x1_count=0; //Zähler für
unsigned long long int x2_count=0,x3_count=0; //Chiffretexte

int readconfig(void) //liest Konfigurationsdatei ein
{
  int i=0;
  char ch;
  FILE *cf;
  char *zeile,*var,*wert;
  cf = fopen("r6attack.cfg","r");
  if(cf==NULL)
    {
      perror("Fehler beim Öffnen der Konfig-Datei");
      return(-1);
    }
  zeile = (char *)malloc(256);
  wert = (char *)malloc(20);
  var =  (char *)malloc(20);
  while ((ch= fgetc(cf))!=EOF)
   {
     // ASCII-Steuerzeichen fuer LF (line feed)
    if (ch!=10) zeile[i++]=ch;
   else
   {
     zeile[i]=,\0,;
     var= (char *)strtok(zeile,"=");
      wert=(char *)strtok(NULL,"=");
      if (var!=0 && wert!=0)
      {
        if (!strcmp(var,"k0_start")) startk0=atoi(wert);
        if (!strcmp(var,"k0_end")) endk0=atoi(wert);
        if (!strcmp(var,"k1_start")) startk1=atoi(wert);
        if (!strcmp(var,"k1_end")) endk1=atoi(wert);
        if (!strcmp(var,"k2_start")) startk2=atoi(wert);
        if (!strcmp(var,"k2_end")) endk2=atoi(wert);
        if (!strcmp(var,"k3_start")) startk3=atoi(wert);
        if (!strcmp(var,"k3_end")) endk3=atoi(wert);
        if (!strcmp(var,"k4_start")) startk4=atoi(wert);
        if (!strcmp(var,"k4_end")) endk4=atoi(wert);
     }
      i=0;
```

```
    }
  }
  fclose(cf);
  return(0);
}

int read_cipher_texts(u32 *ct) //liest Chiffretexte ein
{
  u8 n=0;
  u64 ct_read=0;
  FILE *cipherfile; //Datei mit Ciphertexten
  cipherfile=fopen("r6cipher.bin","r");
  if(cipherfile==NULL)
    {
      perror("Konnte die Chiffre-Datei nicht oeffnen !\n");
      abort();
    }

  for(ct_read=0;ct_read<numberplain/(sizeof(u32)*8);ct_read++)
    {
      if((n=fread(ct,sizeof(u32),1,cipherfile))!=1)
        {
          printf("Fehler beim Einlesen der Chiffren! %d\n",n);
          printf("%llu von %llu Datensätzen gelesen\n",
                 ct_read,(numberplain/8));
          abort();
        }
      ct++;
    }
  fclose(cipherfile);
  return(0);
}

// direkter Zugriff auf einzelne Bits
#define BIT0(x) ((x)&(1))
#define BIT1(x) ((x)&(2))
#define BIT2(x) ((x)&(4))
#define BIT3(x) ((x)&(8))
#define BIT4(x) ((x)&(16))
#define BIT5(x) ((x)&(32))
#define BIT6(x) ((x)&(64))
```

```
#define BIT7(x) ((x)&(128))
#define BIT8(x) ((x)&(256))
#define BIT9(x) ((x)&(512))
#define BIT10(x) ((x)&(1024))
#define BIT11(x) ((x)&(2048))
#define BIT12(x) ((x)&(4096))
#define BIT13(x) ((x)&(8192))
#define BIT14(x) ((x)&(16384))
#define BIT15(x) ((x)&(32768))
#define BIT16(x) ((x)&(65536))
#define BIT17(x) ((x)&(131072))
#define BIT18(x) ((x)&(262144))
#define BIT19(x) ((x)&(524288))
#define BIT20(x) ((x)&(1048576))
#define BIT21(x) ((x)&(2097152))
#define BIT22(x) ((x)&(4194304))
#define BIT23(x) ((x)&(8388608))
#define BIT24(x) ((x)&(16777216))
#define BIT25(x) ((x)&(33554432))
#define BIT26(x) ((x)&(67108864))
#define BIT27(x) ((x)&(134217728))
#define BIT28(x) ((x)&(268435456))
#define BIT29(x) ((x)&(536870912))
#define BIT30(x) ((x)&(1073741824))
#define BIT31(x) ((x)&(2147483648))

//Berechnung von Stufe1 als inline Fkt.
inline int process_stage1(c,i,j) {
 u32 t32;
 u8 c0,c1,c2,c3;
                ++x1_count;
                t32=i;
                t32=t32<<5;
                t32=t32^j;

                c0=t32;
                c3=t32>>8;
                c2=t32>>16;
                c1=t32>>24;
#ifdef DEBUG
                printf("STUFE1 E: c0:%x c1:%x c2:%x c3:%x \n"
```

```
                       ,c0,c1,c2,c3);
#endif
                  c1=S1[c1^k1]>>24;
                  t32=c2;
                  t32=t32<<8;
                  t32=t32^c3;
                  t32=t32<<8;
                  t32=t32^c0^c1; //Reihenfolge: c2,c3,x1

                  if((x1[t32])==0){x1[t32]=1;}else{x1[t32]=0;}
#ifdef DEBUG
                  printf("STUFE1 A:        c2:%x c3:%x x1:%x\n"
                       ,c2,c3,c0^c1);
#endif
                  }

#define bit(x,i) (((x)>>(i))&(1))
#define swap_bit(x,i) ((x)^(1<<(i)))

int main()
{
 clock_t elapsed; //Komplette Laufzeit
 clock_t deltime; // Zeit zum Löschen von X1
 clock_t opentime; // Zeit für Dateizugriff
 clock_t stufe0_zeit,stufe1_zeit,stufe2_zeit,stufe3_zeit;
 clock_t elementary; //Zeit für Berechnung mittels K4
 clock_t mem_time; //Zeit zum Reservieren des Speichers
 long double sec=0;

 unsigned long long int candi_count=0; //Zähler für Kandidaten
 int n;

 FILE *outfile;//Ausgabedatei
 u32 c;
 u8 c0,c1,c2,c3;//Bytes der CT´s
 int j;

 u8 tt,t1,t2; //temporäre Variablen
 u32 t32;
 unsigned long int i,index;
 int dummy;
```

```
u32 *ct_ptr, *k0_ptr;

struct utsname unamebuf;
char zs[255];
char fs[255]="%A %d %B %Y %H:%M:%S";
 time_t startzeit;
struct tm *zeit;

readconfig();
printf("6-Runden-Angriff mit Teilsummen \n");
printf("Version vom "__DATE__" um "__TIME__" Uhr ("
      __FILE__"))");

startzeit= time((time_t *)NULL);;
zeit=localtime(&startzeit);
strftime(zs,255,fs,zeit);
printf("Startzeit: ");
printf("%s \n",zs);
if(uname(&unamebuf)==-1)
 {
 }
else
 {
   printf("Rechner: %s (%s  %s %s) \n"
         ,unamebuf.nodename,unamebuf.machine
         ,unamebuf.sysname,unamebuf.release);
 }

printf("teste %llu Schlüssel \n"
      ,(long long int) (endk0-startk0+1)*(endk1-startk1+1)
      *(endk2-startk2+1)*(endk3-startk3+1)*(endk4-startk4+1));
printf("K0: %3d . . . %3d \n",startk0,endk0);
printf("K1: %3d . . . %3d \n",startk1,endk1);
printf("K2: %3d . . . %3d \n",startk2,endk2);
printf("K3: %3d . . . %3d \n",startk3,endk3);
printf("K4: %3d . . . %3d \n",startk4,endk4);

elapsed=-clock();
printf("Reserviere Speicher . . .");
```

```
mem_time=-clock();

ct_ptr= (u32*) malloc(numberplain/8);
if(ct_ptr==NULL)
  {
    printf("Konnte Speicherplatz nicht reservieren !\n");
    abort();
  }
memset(ct_ptr,0,numberplain/8);
memset(x0,0,numberplain/8);
memset(x2,0,numberplain/256/256);

//Zum Syncen . . .
for(i=0;i<(numberplain/32);i++) dummy=(ct_ptr+i);
for(i=0;i<(numberplain/32);i++) x0[i]=0;
for(i=0;i<(numberplain/256);i++) x1[i]=0;
for(i=0;i<(numberplain/256/256);i++) x2[i]=0;

mem_time+=clock();
printf("fertig\n");
deltime= -clock();
memset(x0,0,numberplain/32); // Lösche CT
deltime+=clock();

printf("Lese Chiffren . . . ");
opentime=-clock();
read_cipher_texts(ct_ptr);
opentime+=clock();
dummy=(ct_ptr+numberplain/32)-1;
printf("fertig \n");

outfile=fopen("candidates.bin","w");
if(outfile==NULL)
  {
    perror("Konnte Ausgabedatei nicht oeffnen");
  }

printf("Starte Angriff . . .\n");

//STUFE 0
//*******************************************
```

```
  for (k0=startk0;k0<=endk0;k0++)
    {
      stufe0_zeit=-clock();
      memset(x0,0,numberplain/8);
      k0_ptr=ct_ptr;
      for(i=0;i<numberplain/32;i++)
        { //alle CT´s
          c=*k0_ptr;//hole ct
          for(j=0;j<32;j++)
            {
              if(bit(c,j)==1)
                {
                  ++c_count;
                  t32=i;
                  t32=t32<<5;
                  t32=t32^j;
                  c3=t32;
                  c2=t32>>8;
                  c1=t32>>16;
                  c0=t32>>24;
#ifdef DEBUG
                  printf("STUFE0 E: c0:%x c1:%x c2:%x c3:%x \n"
                         ,c0,c1,c2,c3);
#endif
                  c0=S0[c0^k0]>>24;
                  t32=c1;
                  t32=t32<<8;
                  t32=t32^c2;
                  t32=t32<<8;
                  t32=t32^c3;
                  t32=t32<<8;
                  t32=t32^c0;
                  x0[(t32>>5)]=swap_bit(x0[(t32>>5)],t32%32);

#ifdef DEBUG
                  printf("STUFE0 A: c0:%x c1:%x c2:%x c3:%x \n"
                         ,c0,c1,c2,c3);
#endif
                }
            }
```

```
      k0_ptr++;
     }
    stufe0_zeit+=clock();

//STUFE 1
//*******************************************

    for(k1=startk1;k1<=endk1;k1++)
     {
      stufe1_zeit=-clock();
      x1_count=0;
      //Stufe 1

      c0=0;c1=0;c2=0;c3=0;
      memset(x1,0,numberplain/256); // Lösche X1
      for(i=0;i<numberplain/32;i++)
       {
        c=x0[i];//hole x0
        if(BIT0(c)>0) process_stage1(c,i,0);
        if(BIT1(c)>0) process_stage1(c,i,1);
        if(BIT2(c)>0) process_stage1(c,i,2);
        if(BIT3(c)>0) process_stage1(c,i,3);
        if(BIT4(c)>0) process_stage1(c,i,4);
        if(BIT5(c)>0) process_stage1(c,i,5);
        if(BIT6(c)>0) process_stage1(c,i,6);
        if(BIT7(c)>0) process_stage1(c,i,7);
        if(BIT8(c)>0) process_stage1(c,i,8);
        if(BIT9(c)>0) process_stage1(c,i,9);
        if(BIT10(c)>0) process_stage1(c,i,10);
        if(BIT11(c)>0) process_stage1(c,i,11);
        if(BIT12(c)>0) process_stage1(c,i,12);
        if(BIT13(c)>0) process_stage1(c,i,13);
        if(BIT14(c)>0) process_stage1(c,i,14);
        if(BIT15(c)>0) process_stage1(c,i,15);
        if(BIT16(c)>0) process_stage1(c,i,16);
        if(BIT17(c)>0) process_stage1(c,i,17);
        if(BIT18(c)>0) process_stage1(c,i,18);
        if(BIT19(c)>0) process_stage1(c,i,19);
        if(BIT20(c)>0) process_stage1(c,i,20);
        if(BIT21(c)>0) process_stage1(c,i,21);
        if(BIT22(c)>0) process_stage1(c,i,22);
```

```
          if(BIT23(c)>0) process_stage1(c,i,23);
          if(BIT24(c)>0) process_stage1(c,i,24);
          if(BIT25(c)>0) process_stage1(c,i,25);
          if(BIT26(c)>0) process_stage1(c,i,26);
          if(BIT27(c)>0) process_stage1(c,i,27);
          if(BIT28(c)>0) process_stage1(c,i,28);
          if(BIT29(c)>0) process_stage1(c,i,29);
          if(BIT30(c)>0) process_stage1(c,i,30);
          if(BIT31(c)>0) process_stage1(c,i,31);

         }
        stufe1_zeit+=clock();
 //STUFE 2***************************************
        //stufe2_zeit=-clock();
        for(k2=startk2;k2<=endk2;k2++)
         {
          stufe2_zeit=-clock();
          memset(x2,0,numberplain/256/256);
          for(i=0;i<numberplain/256;i++)
           {
            if(x1[i]==1)
             {
              ++x2_count;
              index=i;
              t1=index;
              c3=(index>>8);
              c2=(index>>16);
#ifdef DEBUG
              printf("Stufe2 E:      c2:%x c3:%x x1:%x\n"
                     ,c2,c3,t1);
#endif
              tt=S2[c2^k2]>>24;
              t1=t1^tt;
              index=c3;
              index=index<<8;
              index=index+t1;
              if((x2[index])==0){x2[index]=1;}
              else{x2[index]=0;}
#ifdef DEBUG
              printf("Stufe2 A:           c3:%x      x2:%x\n"
                     ,c3,t1);
```

```
#endif
                }
              }

             stufe2_zeit+=clock();

            //Stufe3 ******************************

            for(k3=startk3;k3<=endk3;k3++)
              {
                stufe3_zeit=-clock();
                memset(x3,0,numberplain/256/256/256); //Lösche X3
                for(i=0;i<(numberplain/256)/256;i++)
                  {
                    if(x2[i]==1)
                      {
                        ++x3_count;
                        index=i;
                        t2=index;
                        c3=index>>8;
#ifdef DEBUG
                        printf("Stufe3 E:      c3:%x      x2:%x \n"
                               ,c3,t2);
#endif
                        tt=S3[c3^k3]>>24;
                        t2=t2^tt;
                        if((x3[t2])==0){x3[t2]=1;}else{x3[t2]=0;}
#ifdef DEBUG
                        printf("Stufe3 A:      x3:%x\n",t2);
#endif
                      }
                  }

               stufe3_zeit+=clock();

                //Stufe4********************************

                elementary=-clock();
                for(k4=startk4;k4<=endk4;k4++)
                  {
                    x4=0;
```

```
                for(i=0;i<256;i++) //alle Teilsummen
                  {
                    if(x3[i] ==1)
                      {
                        // printf("Stufe4 E: x3:%x\n",i);
                        x4=x4^Si[i^k4];
                      }
                  }
                if(x4==0)
                  {
#ifdef DEBUG
                    printf("Kandidat gefunden! %d%d%d%d%d \a\n"
                           ,k0,k1,k2,k3,k4);
#endif
                    candi_count++;
                    n=fwrite(&k0,sizeof(u8),1,outfile);
                    n=fwrite(&k1,sizeof(u8),1,outfile);
                    n=fwrite(&k2,sizeof(u8),1,outfile);
                    n=fwrite(&k3,sizeof(u8),1,outfile);
                    n=fwrite(&k4,sizeof(u8),1,outfile);
                  }
                else
                  {
                  }
              }//Stufe4
            elementary+=clock();

          }//Stufe3
        }//Stufe2
      }//Stufe1
    }//Stufe0

  fclose(outfile);
  elapsed+=clock();
  sec=(double)elapsed/(double)CLOCKS_PER_SEC;

  printf("Zeit zum Löschen: %f sec \n"
        ,(double)deltime/(double)CLOCKS_PER_SEC);
  printf("Zeit für Dateizugriff: %f sec \n"
        ,(double)opentime/(double)CLOCKS_PER_SEC);
  printf("Zeit: %Lf s bzw. %Lf min \n"
```

```
      ,(long double)sec,(long double)sec/60);
printf("%llu Schlüssel getestet\n"
      ,(long long int)(endk0-startk0+1)*(endk1-startk1+1)
      *(endk2-startk2+1)*(endk3-startk3+1)*(endk4-startk4+1));
printf("%llu mögliche Kandidaten gefunden!\n"
      ,(long long int)candi_count);
printf("%llu Ciphertexte bearbeitet \n"
      ,(long long int)c_count);
printf("%llu Zwischenergebnisse in Stufe 1\n",x1_count);
printf("%llu Zwischenergebnisse in Stufe 2\n",x2_count);
printf("%llu Zwischenergebnisse in Stufe 3\n",x3_count);
printf("Zeit für Stufe 0: %Lf\n"
      ,(long double)stufe0_zeit/(long double)CLOCKS_PER_SEC);
printf("Zeit für Stufe 1: %Lf\n"
      ,(long double)stufe1_zeit/(long double)CLOCKS_PER_SEC);
printf("Zeit für Stufe 2: %Lf\n"
      ,(long double)stufe2_zeit/(long double)CLOCKS_PER_SEC);
printf("Zeit für Stufe 3: %Lf\n"
      ,(long double)stufe3_zeit/(long double)CLOCKS_PER_SEC);
printf("Zeit für Stufe 4: %Lf\n"
      ,(long double)elementary/(long double)CLOCKS_PER_SEC);
printf("Zeit zum Reservieren des Speichers %Lf\n"
      ,(long double)mem_time/(long double)CLOCKS_PER_SEC);
return(0);
}
```

Literaturverzeichnis

[Dam90] DAMGÅRD, IVAN: *A Design principle for hash functions*. In: BRASSARD, G. (Herausgeber): *Proc. CRYPTO 89*, Seiten 416–427. Springer-Verlag, 1990. Lecture Notes in Computer Science No. 435.

[DH76] DIFFIE, W. und M. E. HELLMAN: *New Directions in Cryptography*. IEEE Trans. Inform. Theory, IT-22:644–654, November 1976.

[DKR97] DAEMEN, J., L. KNUDSEN und V. RIJMEN: *The Block Cipher Square*, 1997.

[DP84] DAVIES, D. W. und W. L. PRICE: *Security for Computer Networks: An Introduction to Data Security in Teleprocessing and Electronic Funds Transfer*. John Wiley and Sons, New York, 1984.

[DR99] DAEMEN, J. und V. RIJMEN: *AES proposal: Rijndael*, 1999.

[FKL+00] FERGUSON, N., J. KELSEY, S. LUCKS, B. SCHNEIER, M. STAY, D. WAGNER und D. WHITING: *Improved Cryptanalysis of Rijndael*. In: *Seventh Fast Software Encryption Workshop*, Seite 19, Berlin, Germany / Heidelberg, Germany / London, UK / etc., 2000. Springer-Verlag.

[GM00] GILBERT, H. und M. MINIER: *A collision attack on 7 rounds of Rijndael*, 2000.

[Knu98] KNUDSEN, L.: *Block Ciphers: A Survey*, 1998.

[LM90] LAI, X. und J. MASSEY: *A Proposal for a New Block Encryption Standard*. In: DAMGÅRD, I.B. (Herausgeber): *EUROCRYPT90*, Seiten 389–404. Springer-Verlag, 1990. Lecture Notes in Computer Science No. 473.

[LN86] LIDL, R. und H. NIEDERREITER: *Introduction to finite fields and their applications*. Cambridge University Press, 1986.

[Mer89] MERKLE, R.C.: *A Fast Software One-Way Hash Function*. Journal of Cryptology, 3(1):43–58, 1989.

[MMO85] MATYAS, S.M., C.H. MEYER und J. OSEAS: *Generating strong one-way functions with cryptographic algorithm.* IBM Technical Disclosure Bulletin, 27:5658–5659, 1985.

[Nyb93] NYBERG, K.: *Differentially uniform mappings for cryptography.* In: HELLESETH, TOR (Herausgeber): *Advances in Cryptology - EuroCrypt '93*, Seiten 55–64, Berlin, 1993. Springer-Verlag. Lecture Notes in Computer Science Volume 765.

[OG94] O'CONNOR und GOLIC: *A Unified Markov Approach to Differential and Linear Cryptanalysis.* In: *ASIACRYPT: Advances in Cryptology – ASIACRYPT: International Conference on the Theory and Application of Cryptology.* LNCS, Springer-Verlag, 1994.

[oS77] STANDARDS, NATIONAL BUREAU OF: *Announcing the Data Encryption Standard.* Technischer BerichtFIPS Publication 46, National Bureau of Standards, Januar 1977.

[Oua97] OUALLINE, STEVE: *Practical C.* O'Reilly & Associates, 1997.

[PGV94] PRENEEL, BART, RENÉ GOVAERTS und JOOS VANDEWALLE: *Hash functions based on block ciphers: a synthetic approach.* In: STINSON, DOUGLAS R. (Herausgeber): *Proc. CRYPTO 93*, Seiten 368–378. Springer, 1994. Lecture Notes in Computer Science No. 773.

[Rob94] ROBSHAW, M.J.B.: *Block Ciphers.* Technischer BerichtTR - 601, RSA Laboratories, July 1994.

[Sha49] SHANNON, C. E.: *Communication theory of secrecy systems.* Bell Sys. Tech. J., 28:657–715, 1949.

[Slo83] SLOANE, N.J.A.: *Encrypting by Random Rotations.* In: T.BETH (Herausgeber): *Cryptography, Proc. Burg Feuerstein 1982*, Band 149, Seiten 71–128, Berlin, 1983. Springer-Verlag.

[vG99] VON ZUR GATHEN, J. und J. GERHARD: *Modern Computer Algebra.* Cambridge University Press, 1999.

[von99] VON ZUR GATHEN, J.: *Lecture Notes on Cryptography*, WS 1998/99.

[Yuv79] YUVAL, G.: *How to Swindle Rabin.* Cryptologia, 3:187–189, Juli 1979.