Informationstheorie, Codierung und Kryptologie

Mirosław Kutyłowski und Willy-B. Strothmann
Uni-GH Paderborn, Skript zur Vorlesung von M. Kutyłowski

WS 1995/96

Version: -0.5

Korrekturen, Verbesserungsvorschläge an: willy@uni-paderborn.de

die aktuellste Version des Skripts kann man auf dem WWW-Server der Uni-GH Paderborn über die Datei:

http://www.uni-paderborn.de/fachbereich/AG/agmadh/WWW/german/LehreKuty/lehre.html

finden. Das Skript basiert auf einer Latex Source (geschrieben durch Guido Haeseler), die auch über diese Adresse zugänglich ist. Die Übungsaufgaben sind ebenso durch WWW zugänglich.

Danksagung

An diesem Skript haben neben den Autoren mitgewirkt:

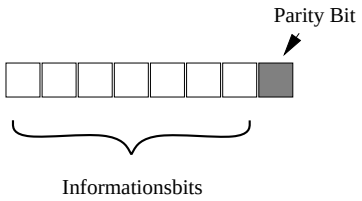
Guido Haeseler	LaTeX-Sourcen der Ur-Version (inklusive Bilder)
Frank Beste	Bilder zum Kerberos-System
Artur Czumaj	Abschnitt über Superstrings

Inhaltsverzeichnis

0.1 Einleitung

0.1.1 Zwecke der Codierung

- **Fehlererkennung**: Fehlerhafte Nachrichten werden erkannt (Beispiel: Geldüberweisungen zwischen Banken, eine eins extra könnte eine Million DM bedeuten). Die Nachrichten werden so codiert, daß zufällige Änderungen erkannt werden können.
  Beispiel: Parity Bit bei ASCII 128-Zeichensatz



- **Fehlerkorrektur**: Fehlerhafte Nachrichten werden automatisch korrigiert. (Beispiel: Modem und Rauschen in der Telefonleitung. Ohne Fehlerkorrektur kann man wilde Zeichen auf den Bildschirm bekommen.) Die Nachrichten werden so codiert, daß zufällige Änderungen rückgängig gemacht werden können.
  Beispiel: Majority Code – jedes Bit wird dreimal geschickt. Beim Empfang: "majority vote", 111, 110, 101, 011 werden als die Nachricht 1 betrachtet; 000, 100, 010, 001 werden als die Nachricht 0 betrachtet.

- **Komprimierung von Dateien**: Die meisten Dateien sind redundant, enthalten mehr Bits als Informationsbits.
  Komprimierung: Nachricht $\rightarrow$ gepackte Version (die die ursprüngliche Informationen enthält)
  Anwendungen:
  – Reduzierung des Speicherbedarfs
  – Übertragung von mehr Informationen pro Zeiteinheit
  – Eliminierung von unwichtigen Informationen (bei Bildern, ...),
  Beispiele: gzip, jpeg, ...

- **Kryptographische Codes**: Dateien sollen nur für berechtigte Personen (u.a. auch elektronische Personen in Netzen) lesbar sein, aber auch für verschiedene andere Sicherheitszwecke (authentification, ...). Beispiele: DES, PGP Programme

Die Ziele sind widersprüchlich:
Bei Fehlererkennung und Fehlerkorrektur wird die Nachrichtenlänge vergrössert. Fehler sind (im gewissen Rahmen) tolerierbar.
Bei Komprimierung wird die Nachrichtenlänge kürzer. Fehelr wirken sich dadurch jedoch stärker aus.
Bei Kryptographische Codes kann die Länge konstant bleiben, denn die Codierung macht die Nachricht unlesbar!

0.1.2 Kanalmodel

Das "Rauschen" kann elektomagnetische Störung bedeuten aber auch ein Eingriff gegen kryptographische Codes.

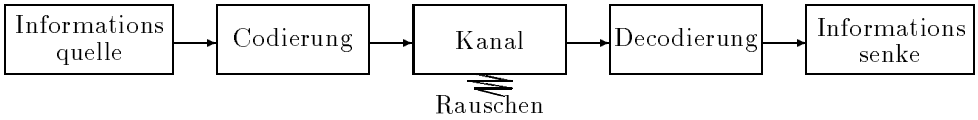


Abbildung 0.1: Beispiel eines Übertragungsweges

0.1.3 Codierung - allgemeine Definition

Fehlertolerante oder fehlerkorrigierende Codierung:
Quellalphabet Σ, Codealphabet Γ

$K : \Sigma \to \Gamma^*$ ist eine Codierung falls K total ist, d.h. $\forall a \in \Sigma \quad \exists K(a) \in \Gamma^*$.
$K(a)$ ist das Codewort von a. $K(\Sigma)$ oder auch kurz K ist der (induzierte) Code.

Beispiel: 2-out-of-5 Code

$x \in \Sigma$	1	2	3	4	5	6	7	8	9	0
$K(x) \in \Gamma^*$	11000	10100	01100	10010	01010	00110	10001	01001	00101	00011

Decodierungs-Verfahren ($K(x) := a_1 \ldots a_5$):
if $K(x) = 00011$ **then** $x := 0$ **else** $x := a_2 + 2 \cdot a_3 + 4 \cdot a_4 + 7 \cdot a_5$

Erweiterung auf Codierung von Worten

$w = x_1 x_2 \cdots x_n \in \Sigma^n$, dann $K(w) = K(x_1)K(x_2)\cdots K(x_n)$

Beispiel: Morse-Code

$K(A) = \bullet-$, $K(B) = -\bullet\bullet\bullet$, $K(C) = -\bullet-\bullet$, $K(D) = -\bullet\bullet$, $K(E) = \bullet$, usw.
Code mit 3 Zeichen: $-$, $\bullet$, Pause

- Das Pausezeichen ist notwendig um die Codeworte der einzelnen Buchstaben zu separieren.
- Bei 2-out-of-5 Code ist die Pause nicht notwendig, da alle Codeworte die Länge 5 haben.
- Es existieren Codes ohne Pause, wobei nicht alle Codeworte $K(a)$ die gleiche Länge haben (Codes mit unterschiedlicher Länge). Das Decodieren kann jedoch schwierig werden.

0.1.4 Block Codes

Block Code : alle $K(a)$ haben die gleiche Länge
sehr oft sind die Codesymbole nur 0,1 (binäre Block Codes)

Beispiele von Block Codes

1. ASCII
   128 Zeichen, kodiert durch Blöcke von 8 Bits
   (7 Bits als Informationsbits + 1 Parity-Check-Bit)
   $x_1, \ldots, x_7$ -Informationsbits, x_8 so gewählt, daß $\sum x_i = 0 \bmod 2$
2. ISBN (Buchnummern)
   9 Ziffern (von 1 bis 9) + letzte Ziffer (1 bis 10 , 10 gezeichnet als X) als Kontrollzeichen
   $a_1 a_2 \ldots a_{10}$ ist korrekt, falls

$\sum_{i=1}^{10} i \cdot a_{11-i} = 0 \bmod 11$
Beispiel: ISBN 0-13-283796-X (Bindestriche sind ohne Bedeutung)

Interessante Eigenschaft von ISBN Codes:

- Änderung einer Ziffer ist erkennbar,
- Vertauschen von zwei unterschiedlichen benachbarten Ziffern ist erkennbar

Kapitel 1

Fehlererkennende und -korrigierende Codes

1.0.1 Störungen

Modelle:

1. Zufällige Fehler: an jeder Stelle kann ein Fehler mit Wahrscheinlichkeit p ($p \geq 0$) auftreten. Die Fehler an verschiedenen Positionen sind unabhängig.
2. Büschelstörungen (mehr realistisch) – die Fehler treten in Gruppen an benachbarten Positionen auf. Die fehlerhafte Region kann man jedoch abgrenzen.

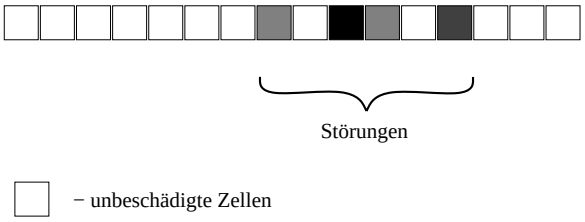


Abbildung 1.1: Büschelstörungen

1.0.2 Hamming-Distanz

Seien $\vec{a}$ und $\vec{b}$ zwei Folgen der Länge n von Symbolen aus Σ. Die Hamming-Distanz ist dann definiert durch $d_H(\vec{a},\vec{b}) := |\{i \mid a_i \neq b_i\}|$. Ist $\Sigma = \{0,1\}$ so kann die Hamming-Distanz auch wie folgt definiert werden: $d_H(\vec{a},\vec{b}) := \sum_{i=1}^{n}(a_i + b_i \bmod 2)$.

Die Hamming-Distanz ist eine Metrik

$d_H(\vec{a},\vec{b}) \geq 0; d_H(\vec{a},\vec{b}) = 0 \Leftrightarrow a = b$	(Definitheit)
$d_H(\vec{a},\vec{b}) = d_H(\vec{b},\vec{a})$	(Symmetrie)
$d_H(\vec{a},\vec{b}) \leq d_H(\vec{a},\vec{c}) + d_H(\vec{c},\vec{b})$	(Dreiecksungleichung)

1.0.3 Fehlererkennung

Block-Codierung K erkennt bis zu t Fehler

$$\Updownarrow$$

$$\forall a, b \in \Sigma; a \neq b : d_H(K(a), K(b)) \geq t + 1$$

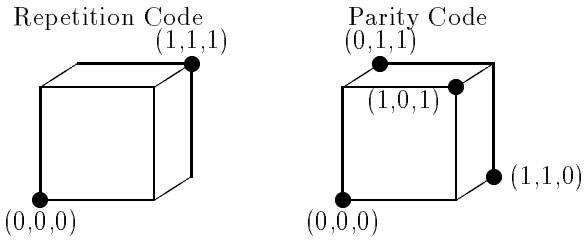


Abbildung 1.2: Beispiele von Codes

1.0.4 Fehlerkorrektur

$$\begin{array}{c}\text{Block-Codierung } K \text{ korrigiert bis zu } t \text{ Fehler}\\ \Updownarrow\\ \forall a,b \in \Sigma; a \neq b : d_H(K(a), K(b)) \geq 2t+1\end{array}$$

1.0.5 Notwendige Eigenschaften von fehlererkennden und fehlerkorrigierenden Codes

- $K(a)$ ist leicht berechenbar (leichtes Codieren) für jedes $a \in \Sigma$
- $K^{-1}(x)$ ist leicht berechenbar (leichtes Decodieren) für jedes Codewort $x \in \Gamma^*$
- die fehlerhaften Codewörter sind leicht zu entdecken bzw. korrigieren
- die Länge der Codewörter soll möglichst klein sein

Gute Codes zu finden ist nicht leicht!

1.0.6 Lineare Codes

Bei linearen Codes sind die Codesymbole aus einem endlichen Körper. Dabei wird im weiteren hauptsächlich von dem Körper $(\{0,1\}, \oplus_2, \odot_2) = (\{0,1\}, \text{XOR}, \wedge)$ ausgegangen; der Code heißt dann binär.

Sei F ein Körper. Eine Blockcodierung $K : \Sigma \to F^k$ heißt **linear** genau dann, wenn

1. $\forall a,b \in \Sigma : \exists c \in \Sigma : \overrightarrow{K(a)} \oplus_F^b \overrightarrow{K(b)} = \overrightarrow{K(c)}$
2. $\forall \lambda \in F, a \in \Sigma : \exists c \in \Sigma : \lambda \cdot \overrightarrow{K(a)} = \overrightarrow{K(c)}$

d.h. $\{K(a) \mid a \in \Sigma\}$ ist ein linearer Unterraum von F^k.

Beispiel

Der Parity-Check Code ist linear:
Seien $s_1 \cdots s_n, r_1 \cdots r_n$ Codewörter, d.h. $\sum_i s_i = \sum_i r_i = 0 \bmod 2$, dann ist mit $u_i := s_i \oplus_2 r_i$ das Wort $u_1 \cdots u_n$ ein Codewort, weil $\sum_i u_i = \sum_i s_i \oplus_2 r_i = \sum_i s_i + \sum r_i \bmod 2$

1.0.7 „Parity check“ für lineare Codes

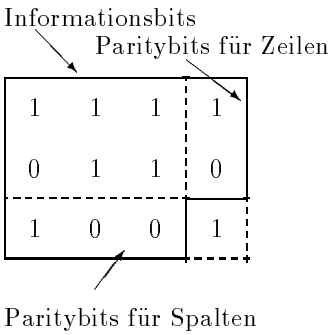
Für jeden linearen Unterraum L gibt es ein System linearer Gleichungen, so daß $\vec{x} \in L \iff \vec{x}$ erfüllt diese Gleichungen

Beispiele

- für Parity-Check Code: $x_1 + \cdots + x_n = 0 \bmod 2$
- Rechteck-Codes (Parity-Bit für Zeilen und Spalten; schlechte Codes, nur aus didaktischen Gründen erwähnt!)

x_1	x_2	x_3	x_4
x_5	x_6	x_7	x_8
x_9	x_{10}	x_{11}	x_{12}

Gleichungen:
$x_1 + x_2 + x_3 + x_4 = 0 \bmod 2$
$x_1 + x_5 + x_9 = 0 \bmod 2$
$x_1 + \cdots + x_{12} = 0 \bmod 2$



1.0.8 Distanz zwischen Codewörtern bei linearen binären Codes

Sei $\vec{v}$ ein Codewort. Dann ist das Gewicht von $\vec{v}$ die Anzahl der von 0 verschiedenen Stellen, bei binären Codes also die Anzahl der Einsen.

Sind $\vec{w}$ und $\vec{v}$ Codewörter, dann gilt, daß $d_H(\vec{w}, \vec{v}) = \text{Gewicht}(\vec{w} + \vec{v})$
($\vec{w} + \vec{v}$ ist auch ein Codewort !!!)

Lemma 1 *Sei K ein linearer binärer Code. Dann gilt:*
$$\min\{d_H(\vec{w}, \vec{v}) \mid \vec{w}, \vec{v} \in K : \vec{w} \neq \vec{v}\} = \min\{\mathit{Gewicht}(\vec{u}) \mid \vec{u} \in K : \vec{u} \neq \vec{0}\}.$$

Beweis: Sei $\vec{u}$ ein Codewort mit minimalem Gewicht

1. $d(\vec{u}, \vec{0}) = \text{Gewicht}(\vec{u})$, also $\min d(\vec{w}, \vec{v}) \leq \text{Gewicht}(\vec{u})$
2. $\min d(\vec{w}, \vec{v}) = d(\vec{w_0}, \vec{v_0}) = \text{Gewicht}(\vec{w_0} + \vec{v_0}) \geq \text{Gewicht}(\vec{u})$

□

1.1 Parity-Check-Matrix

$$\begin{bmatrix} \\ H \\ \\ \end{bmatrix} \cdot \begin{bmatrix} x_1 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} 0 \\ \vdots \\ 0 \end{bmatrix} \iff x_1 \ldots x_n \text{ ist ein Codewort}$$

(H entsteht aus Gleichungen, die die Codierung definieren)

- Sollte eine Zeile von H linear abhängig von einer anderen sein, so kann diese Zeile entfernt werden.

Wie groß sollte H sein?
Wieviele Zeilen gibt es in H?
Die Codierung definiert einen Unterraum der Dimension k. Dann muß H genau $n - k$ linear unabhängige Zeilen enthalten.
H sollte keine linear abhängigen Zeilen enthalten.

Beispiel: Parity-Check Code

$$H = \underbrace{[11 \cdots 1]}_{n-mal} \cdot \begin{bmatrix} x_1 \\ \vdots \\ x_n \end{bmatrix} = [x_1 \oplus_2 \cdots \oplus_2 x_n]$$

1.1.1 Parity-Check-Matrix für Korrektur von 1 Fehler

Sei $H \in M(m,n;R)$ eine $m \times n$ Matrix mit Einträgen aus dem Ring R. Die Matrix $H_{\bullet,i} \in M(m,1;R)$ ist die i-te Spalte und die Matrix $H_{j,\bullet} \in M(1,n;R)$ die j-te Zeile von H.

Lemma 2 *Ein binärer linearer Code K kann einen Fehler korrigieren*
$$\Updownarrow$$
für jede Parity-Check-Matrix H für K gilt:

1. *keine Spalte von H enthält nur Nullen, d.h. $\forall i: H_{\bullet,i} \neq \vec{0}^T$*
2. *in H gibt es keine identische Spalten, d.h. $\forall i,j; i \neq j: H_{\bullet,i} \neq H_{\bullet,j}$.*

Beweis:
($\Downarrow$)
Sei K ein binärer Code, der einen Fehler korrigieren kann und H eine Parity-Check-Matrix für K.

(1) Annahme: Spalte i enthalte nur Nullen.

$$H = \begin{bmatrix} & \overset{i}{\downarrow} & \\ & 0 & \\ \cdots & \vdots & \cdots \\ & 0 & \end{bmatrix} \Rightarrow H \cdot \begin{bmatrix} 0 \cdots 0 & \overset{i}{\downarrow} & 0 \cdots 0 \\ & 1 & \end{bmatrix}^T = H_{\bullet,i} = \begin{bmatrix} 0 \\ \vdots \\ 0 \end{bmatrix}$$

Somit existiert ein Codewort mit Gewicht 1. Widerspruch.

(2) Annahme: Spalte i und j sind identische Spalten.

$$H = \begin{bmatrix} \cdots & \overset{i}{\downarrow} & \cdots & \overset{j}{\downarrow} & \cdots \\ & S & & S & \end{bmatrix} \Rightarrow$$

$$H \cdot \begin{bmatrix} 0 \cdots 0 & \overset{i}{\downarrow} & 0 \cdots 0 & \overset{j}{\downarrow} & 0 \cdots 0 \\ & 1 & & 1 & \end{bmatrix}^T = H_{\bullet,i} + H_{\bullet,j} = S + S = \begin{bmatrix} 0 \\ \vdots \\ 0 \end{bmatrix}.$$

Somit existiert ein Codewort mit Gewicht 2. Widerspruch.

($\Uparrow$)
Sei nun K ein Code, dessen Parity-Check-Matrix die beiden Eigenschaften erfüllt, und $\vec{x}$ ein Codewort mit Gewicht$(\vec{x}) \leq 2$. Damit ist $H \cdot \vec{x}^T$ die Summe von ≤ 2 Spalten. Somit ist $\vec{x} = \vec{0}$ das einzige Codewort mit Gewicht ≤ 2. Es folgt, daß Gewicht$(K) \geq 3$ sein muß und damit kann K einen Fehler korrigieren. □

1.2 Hamming Codes

Parity-Check-Matrix:

- m Zeilen
- $2^m - 1$ Spalten und
- keine Spalte besteht aus Nullen, sonst entspricht jede Folge von m Bits genau einer Spalte

Bemerkung: Die Zeilen dieser Matrix sind linear unabhängig.
Beweis: Angenommen, die i-te Zeile ist eine Linearkombination von den anderen Zeilen. Betrachte die Spalte, die in Zeile i eine Eins hat und sonst nur Nullen. Dann ist diese Eins eine Linearekombination der Nullen. Widerspruch. □

Eigenschaften:

- Codelänge $= n = 2^m - 1$
- Anzahl der Codewörter $= \frac{2^n}{2^m} = 2^{2^m - m - 1}$
- (Dimension des Unterraums aller Codewörter) / (Codelänge n) = Redundanz $= \frac{2^m - m - 1}{2^m - 1} = 1 - \frac{m}{2^m - 1} \overset{m \to \infty}{\longrightarrow} 1$
- Gewicht$(K) = 3$

  Beweis: es gibt Spalten $H_{\bullet,i_1} = \begin{bmatrix} 0 \\ \vdots \\ 0 \\ 0 \\ 1 \end{bmatrix}$, $H_{\bullet,i_2} = \begin{bmatrix} 0 \\ \vdots \\ 0 \\ 1 \\ 0 \end{bmatrix}$, $H_{\bullet,i_3} = \begin{bmatrix} 0 \\ \vdots \\ 0 \\ 1 \\ 1 \end{bmatrix}$ in der Parity-Check-Matrix H.
  Das Wort $\vec{y}$, das Einsen genau auf den Positionen i_1, i_2, i_3 hat, ist dann ein Codewort, da $H \cdot \vec{y}^T = H_{\bullet,i_1} + H_{\bullet,i_2} + H_{\bullet,i_3} = \vec{0}$. Damit ist gezeigt, daß Gewicht$(K) \leq 3$ ist. DIe Tatsache, daß Gewicht$(K) \geq 3$ folgt aus Lemma 2, da H nach Definition keine Null-Spalte und paarweise unterschiedliche Spalten hat. □

1.2.1 Fehlerkorrektur-Prozedur

Sei:

- $\vec{w}$ ein Codewort,
- $\vec{e}$ der Fehlervektor, d.h. $\vec{w} \oplus_2^b \vec{e}$ ist die empfangene Nachricht und
- H die Parity-Check-Matrix des Hamming Codes.

Dann ist $H \cdot (\vec{w} \oplus_2^b \vec{e})^T = H \cdot \vec{w}^T \oplus_2^b H \cdot \vec{e}^T = \vec{0} \oplus_2^b H \cdot \vec{e}^T = H \cdot \vec{e}^T$.

Ist nun also $\vec{e} = \begin{bmatrix} 0 \cdots 0 & \overset{i}{\underset{\downarrow}{1}} & 0 \cdots 0 \end{bmatrix}$, so ist $H \cdot \vec{e}^T = H_{\bullet,i}$.

Algorithmus:

1. berechne $H \cdot \vec{v}^T$ ($\vec{v}$ = empfangene Nachricht)
   $H \cdot \vec{v}^T$ heißt **Syndrom** von $\vec{v}$
2. ist $H \cdot \vec{v}^T$ = Spalte j aus $H \Rightarrow$ tausche das j-te Bit von $\vec{v}$

1.2.2 Ein praktischer Hamming Code:

Ist die i-te Spalte von H gleich der binären Darstellung von i, so kann die Stelle des Fehlers leicht gefunden werden:
$H \cdot \vec{v}^T$ = binäre Darstellung von $i \Rightarrow$ Bit i muß getauscht werden

1.2.3 2 Fehler bei Hamming Codes

Bei 2 Fehlern korrigiert diese Prozedur *das falsche Bit !!!*
Beispiel: $\vec{e}$=[1,1,0, ... ,0]

$$\begin{bmatrix} 0 & 0 & \cdots \\ & \vdots & \\ & 0 & \cdots \\ 0 & 1 & \\ 1 & 0 & \end{bmatrix} \cdot \begin{bmatrix} 1 \\ 1 \\ 0 \\ \vdots \\ 0 \end{bmatrix} = H_{\bullet,1} \oplus_2^b H_{\bullet,2} = \begin{bmatrix} 0 \\ \vdots \\ 0 \\ 1 \\ 1 \end{bmatrix}$$

Man erhält also die binäre Darstellung von 3. Es wird das dritte Bit - statt das erste und zweite Bit - korrigiert.

1.2.4 Hamming Codes sind perfekt

Ein Code mit Blocklänge n, der t Fehler korrigiert, heißt perfekt, falls für jedes Wort der Länge n ein Codewort mit Hamming-Distanz höchstens t existiert.

Lemma 3 *Jedes Wort der Länge $n = 2^m - 1$ ist entweder:*

- *ein Codewort oder*
- *hat Hamming-Distanz 1 zu einem Codewort entfernt.*

Beweis:
$H \cdot \vec{v}^T = \vec{0} \Rightarrow \vec{v}$ ist ein Code.

$H \cdot \vec{v}^T \neq \vec{0} \Rightarrow H \cdot \vec{v}^T = H_{\bullet,j}$. Sei $\vec{e} = \begin{bmatrix} 0 \cdots 0 & \overset{j}{\underset{\downarrow}{1}} & 0 \cdots 0 \end{bmatrix}$, dann gilt $H \cdot (\vec{v} \oplus_2^b \vec{e})^T = H \cdot \vec{v}^T \oplus_2^b H \cdot \vec{e}^T = H_{\bullet,j} \oplus_2^b H_{\bullet,j} = \vec{0}$. Also ist $\vec{v} \oplus_2^b \vec{e}$ ein Codewort. □

1.3 Lineare Codes - Fortsetzung

1.3.1 Lineare Codes für beliebige Körper

Sei F ein endlicher Körper (z.B. $\mathrm{GF}(p) = (\{0, 1, \ldots, p-1\}, +_{\bmod p}, \cdot_{\bmod p}) = (\{0, 1, \ldots, p-1\}, \oplus_p, \odot_p)$ für eine Primzahl p).

Ein k-dimensionaler Unterraum von F^n heißt (n,k)-**Code** im Codealphabet F. Der Code hat per Definition k Informationssymbole und $n-k$ Parity-Check-Symbole (obwohl nicht leicht zu erkennen ist, welches Symbol zu welcher Klasse gehört).

1.3.2 Generatormatrix und Codierverfahren für lineare Codes

Seien $\vec{g}_1, \ldots, \vec{g}_k$ Generatoren des (n,k)-Codes (d.h. Basis des k-dimensionalen Unterraumes). Dann heißt die Matrix $G = \begin{bmatrix} \vec{g}_1 \\ \vdots \\ \vec{g}_k \end{bmatrix}$ **Generatormatrix**.

Anwendung:

Jedes Codewort hat die Form: $\lambda_1 \vec{g}_1 \oplus_p^b \lambda_2 \vec{g}_2 \oplus_p^b \cdots \oplus_p^b \lambda_k \vec{g}_k$
Codierung :
(Quelle:) $\lambda_1 \cdots \lambda_k \overset{1-1}{\longmapsto} (Codewort:) \lambda_1 \vec{g}_1 \oplus_p^b \cdots \oplus_p^b \lambda_k \vec{g}_k = [\lambda_1 \cdots \lambda_k] \cdot G$

Bemerkung: Da $\vec{g}_1, \ldots, \vec{g}_k$ eine Basis ist, ist diese Abbildung eine Bijektion zwischen den Nachrichten der Länge k und der Menge der Codewörter.

1.3.3 Decodierverfahren für lineare Codes

System von linearen Gleichungen $[x_1 \dots x_k] \cdot G =$ Codewort muß gelöst werden.
Im Spezialfall (siehe unten) – trivial

1.3.4 Systematische Codes:

K heißt **systematischer Code** falls K eine Generatormatrix $G = [I|B]$ besitzt, wobei I eine Identitätsmatrix ist.

Ist $G = [I|B]$, dann gilt: $[\lambda_1 \dots \lambda_k] \cdot [I|B] = [\lambda_1 \dots \lambda_k \underbrace{u_1 \dots u_{n-k}}_{\text{Kontrollzeichen}}])$

Beispiel

$$[\lambda_1 \ \lambda_2 \ \lambda_3] \cdot \left[\begin{array}{ccc|c} 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 \end{array}\right] = [\lambda_1 \ \lambda_2 \ \lambda_3 \ \lambda_1 + \lambda_2 + \lambda_3]$$

D.h.: für systematische Codes sind die Information- und Parity-Check-Symbole echt getrennt.

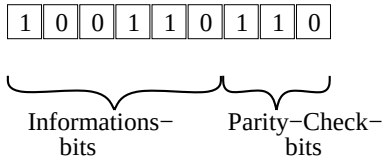


Abbildung 1.3: Die Form eines systematischen Codes

Problem: kann jeder lineare Code so umgebaut werden, daß ein systematischer Code entsteht? JA (siehe unten)

1.3.5 Äquivalente Codes

Gegeben sei ein Code $K : \Sigma \to \Gamma^n$. Bei jedem Codewort permutiere die Codesymbole, verwende dazu eine beliebige (aber feste) Permutation π. Dadurch erhält man eine neue Menge von Codeworten $\pi(K)$.

Definition:

Zwei Codes K_1, K_2 heißen **äquivalent**

$$\Updownarrow$$

$$\exists \pi : \ K_2 = \pi(K_1)$$

Anmerkungen: Man mache sich klar, daß dieses wirklich eine Äquivalenzrelation auf der Menge aller Codes der Länge n induziert. In der Definition der Äquivalenz ist nur gefordert, daß die Mengen der Codewörter gleich ist, jedoch nicht, daß $K_2(x) = \pi(K_1(x))$!

1.3.6 Äquivalenter systematischer Code - Bestimmung einer Generatormatrix

Versuchen wir eine Generatormatrix der Form $[I|G]$ zu finden :
Gegeben sei ein (n, k)-Code K mit einer Generatormatrix G.

1. finde k linear unabhängige Spalten in G (dies ist möglich, denn G hat k linear unabhängige Zeilen)

2. permutiere die Spalten von G, so daß die ersten k Spalten linear unabhängig sind (dadurch ersetzen wir K durch eine äquivalente Codierung)

3. durch Summieren und Vertauschen von Zeilen in G ändert sich der Code nicht (da derselbe Unterraum generiert wird!)
   z.B. $g_1, .., g_n$ kann man durch $g_1 + g_2, g_2, \ldots, g_n$ ersetzen
   wiederhole solche Zeilen-Operationen, bis G die Form $[I|G]$ erhält.

Beispiel

$$\begin{bmatrix} 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 \end{bmatrix} \overset{\text{Perm.}}{\rightsquigarrow} \begin{bmatrix} 1 & 1 & 1 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \end{bmatrix} \rightsquigarrow \begin{bmatrix} 1 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \end{bmatrix} \rightsquigarrow \begin{bmatrix} 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

1.3.7 Parity-Check-Matrix für systematische Codes

Theorem 4 *Sei K ein systematischer Code mit Generatormatrix $G = [I|B] \in M(k, n; R), k \leq n$, wobei $I \in M(k; R)$ eine Identitätsmatrix ist. Dann gilt: $H = [-B^T|I'] \in M(n-k, n; R)$ ist Parity-Check-Matrix für K, wobei $I' \in M(n-k; R)$ eine Identitätsmatrix ist.*

Beweis:
Behauptung 1: Für jedes Codewort $\vec{x}$ gilt $H \cdot \vec{x}^T = \vec{0}$,

denn $H \cdot G^T = [-B^T|I'] \cdot \left[\begin{array}{c} I^T \\ \hline B^T \end{array}\right] = [-B^T \cdot I^T] + [I' \cdot B^T] = [0]$.

Wir zeigen es mit einem Beispiel:

$$G = \left[\begin{array}{ccc|cc} 1 & 0 & 0 & a & b \\ 0 & 1 & 0 & c & d \\ 0 & 0 & 1 & e & f \end{array}\right]; \qquad \text{H=} \left[\begin{array}{ccc|cc} -a & -c & -e & 1 & 0 \\ -b & -d & -f & 0 & 1 \end{array}\right]$$

$$\text{Es gilt : } H \cdot G^T = \begin{bmatrix} -a & -c & -e & 1 & 0 \\ -b & -d & -f & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \\ a & c & e \\ b & d & f \end{bmatrix} = \begin{bmatrix} u & v & w \\ x & y & z \end{bmatrix}$$

$u = -a + (0 \cdot -c) + (0 \cdot -e) + a + (0 \cdot b) = 0$
$y = (-b \cdot 0) + (-d \cdot 1) + (-f \cdot 0) + (0 \cdot c) + (1 \cdot d) = 0$
$\vdots$
d.h. $H \cdot G^T =$ Nullmatrix.
Also gilt für jeden Generator $\vec{g}$ aus G, daß $H \cdot \vec{g}^T = \vec{0}$.Da jedes Codewort $\vec{x}$ die Form $\lambda_1\vec{g_1} + \ldots + \lambda_k\vec{g_k}$ hat, folgt daß $H \cdot \vec{x}^T = \lambda_1 \cdot H \cdot \vec{g_1}^T + \ldots + \lambda_k \cdot H \cdot \vec{g_k}^T = \vec{0} + \ldots + \vec{0} = \vec{0}$.

Behauptung 2: Die Zeilen von H sind linear unabhängig.

Trivial: Zeilen von I' sind linear unabhängig

Behauptung 3: H hat genügend Zeilen.

G hat n Spalten und k Zeilen $\Rightarrow \dim(K) = k \Rightarrow K$ ist durch $n-k$ linear unabhängige Gleichungen definiert. Aber H hat $n-k$ Zeilen! □

1.4 Erweiterter Hamming Code

Jedes Hamming Codewort wird um 1 Bit verlängert, so daß die Summe von allen Bits = 0 ist.

Sei H bzw. H^* die Parity-Check-Matrix für den Hamming Code bzw. erweiterten Hamming Code. Dann gilt: $H^* = \left[\begin{array}{ccc|c} & H & & \begin{matrix}0\\ \vdots \\ 0\end{matrix} \\ \hline 1 & \cdots & 1 & 1 \end{array}\right]$

Theorem 5 *Das minimale Gewicht des erweiterten Hamming Codes ist 4.*

Beweis:
Alle Codewörter haben gerades Gewicht (dieses sieht man z.B. auch an der letzten Zeile der Matrix). 3 = Gewicht(Hamming Code) $\leq$ Gewicht(erw. Hamming Code) $\Rightarrow$ Gewicht(erw. Hamming Code) ≥ 4. Daß das Gewicht gleich 4 ist, ist damit trivial. □

Eigenschaften von erweiterten Hamming Codes:

Ein erweiterter Hamming Code kann:

- 1 Fehler korrigieren und gleichzeitig
- 2 Fehler erkennen

Außerdem ist die Länge von erweiterten Hamming Codes eine Zweierpotenz (sehr praktisch!).
Ähnlich kann man jeden Code mit Codewörtern von ungeradem Gewicht um ein Symbol erweitern!

Ausblick:

Wie kann man Codes konstruieren, die mehrere Fehler korrigieren können?
Denn: eine falsche Ziffer in einer Kundennummer bedeutet mehr als ein falsches Bit.

1.5 Dualer Code

Sei K ein Code und $\vec{x}, \vec{y} \in K$. $\vec{x} \perp \vec{y}$ bedeutet $\vec{x} \cdot \vec{y}^T = 0$ und

$$K^\perp := \{\vec{x} : \forall \vec{y} \in K \ \vec{y} \perp \vec{x}\}$$

heißt *dualer Code* zu K.

Achtung: wir betrachten lineare Räume über endlichen Körpern. Manche Intuitionen von linearen Räumen über den reelen Zahlen sind hier nicht richtig!

1.5.1 Codes vs. duale Codes

Theorem 6 *Sei G eine Generator- und H eine Parity-Check-Matrix für K. Dann ist H eine Generator- und G eine Parity-Check-Matrix von $K^\perp$.*

Beweis:

- Es gilt $H \cdot G^T = [0]$. Somit ist für jedes i die Zeile $H_{i,\bullet}$ orthogonal zu den Generatoren aus G, also $H_{i,\bullet} \perp K$, d.h. $H_{i,\bullet} \in K^\perp$.

- Die Vektoren $H_{i,\bullet}$ sind linear unabhängig.
  Bilden sie eine Basis? Ja, weil die Anzahl der Vektoren $H_{i,\bullet}$ richtig ist:

  Fakt aus Algebra: $\dim(K) + \dim(K^\perp) =$ Dimension des ganzen Raumes $= n$

  $\dim(K) =$ Anzahl der Zeilen in G
  Anzahl der Zeilen in $H = n - \dim(K) = \dim(K^\perp)$!!
  Wir haben damit gezeigt, daß H eine Generatormatrix von $K^\perp$ ist.

- $G \cdot H^T = (H \cdot G^T)^T = [0]^T = [0]$. Weil $G \cdot H^T = [0]$ ist, gilt für jede Zeile $H_{i,\bullet}$, daß $G \cdot H_{i,\bullet}^T = \vec{0}$. Also auch für jede lineare Kombination $\vec{x}$ von Vektoren $H_{i,\bullet}$, d.h. für jedes $\vec{x} \in K^\perp$ gilt $G \cdot \vec{x}^T = \vec{0} \Rightarrow$
  G ist Parity-Check-Matrix für $K^\perp$. □

1.6 Reed-Muller Codes

1.6.1 Eigenschaften von $\Re(r, m)$ Codes

- Codelänge: 2^m
- Anzahl der Informationsbits: $k \stackrel{\text{def}}{=} \sum_{i=0}^{r} \binom{m}{i}$
  (r ist Parameter, beliebig gewählt $r < m$)
- Minimale Distanz: $d = 2^{m-r}$
  (also korrigiert dieser Code bis zu $2^{m-r-1} - 1$ Fehler)

1.6.2 Darstellung boolscher Funktionen durch Wörter

Sei f eine Boolsche Funktion, d.h. $f : \mathbb{Z}_2^m \to \mathbb{Z}_2 = \{0, 1\}$ und $p_0, p_2, \ldots, p_{2^m-1}$ alle Eingabewörter (Bitfolgen der Länge m) lexikographisch geordnet. Dann beschreibt das Wort $\underline{f} = f(p_0)f(p_2)\cdots f(p_{2^m-1})$ vollständig die Funktion f. Es gibt also eine bijektive Abbildung Funktionen$(\mathbb{Z}_2^m; \mathbb{Z}_2) \to \{0,1\}^{2^m}$ mit $f \mapsto \underline{f}$.

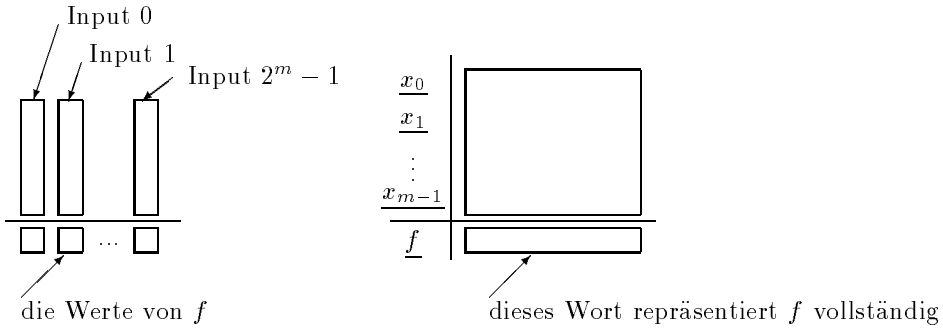


Abbildung 1.4: Repräsentation von boolschen Funktionen:

Einige wichtige Worte/Funktionen:

1. $\underline{0} = \underbrace{0 \cdots 0}_{2^m}$, $\underline{1} = \underbrace{1 \cdots 1}_{2^m}$
2. $\underline{x_i}$ = Zeile $i \in \{0, \ldots, m-1\}$ in Abbildung 1.4, d.h.
   $\underline{x_i}$ enthält eine 1 auf Position $j \in \{0, \ldots, 2^m\} \Leftrightarrow$ die Folge p_j enthält eine 1 auf Position i.

3. Man kann die Funktionen multiplizieren ($\boxdot$) und addieren ($\boxplus$):
   $(f \boxdot g)(\vec{x}) := f(\vec{x}) \odot_2 g(\vec{x}) \in \mathbb{Z}_2$ $\qquad$ $(f \boxplus g)(\vec{x}) := f(\vec{x}) \oplus_2 g(\vec{x}) \in \mathbb{Z}_2$

   Es gilt offensichtlich:

   $\underline{(f \boxdot g)} = \underline{f} \odot_2^{\flat} \underline{g}$ $\qquad$ $\underline{(f \boxplus g)} = \underline{f} \oplus_2^{\flat} \underline{g}$

1.6.3 Darstellung boolscher Funktionen durch Polynome

- Aus $\underline{0}, \underline{1}, \underline{x_0}, \ldots, \underline{x_{m-1}}$ bilden wir (formale) Polynome mit Koeffizienten aus $\mathbb{Z}_2 = (\{0,1\}, \oplus_2, \odot_2)$ also Polynome aus $\mathbb{Z}_2[\underline{x_0}, \ldots, \underline{x_{m-1}}]$.

- Ein Monom ist ein Produkt der Form $\underline{x_{i_1}} \cdot \ldots \cdot \underline{x_{i_s}}$ (die Folge $i_1, \ldots, i_s$ ist beliebig). Das leere Produkt ist die $\underline{1}$.

  Einem Monom $\prod \underline{x_{i_j}}$ kann man mit Hilfe der Abbildung F eine Funktion

  $$F : \text{Monome aus } \mathbb{Z}_2[\underline{x_0}, \ldots, \underline{x_{m-1}}] \quad \longrightarrow \quad \text{boolsche Funktionen } \mathbb{Z}_2^m \to \mathbb{Z}_2$$

  $$F(\prod \underline{x_{i_j}}) := x_{i_1} \boxdot \ldots \boxdot x_{i_s}$$

  bzw. mit Hilfe der Abbildung S ein Wort

  $$S : \text{Monome aus } \mathbb{Z}_2[\underline{x_0}, \ldots, \underline{x_{m-1}}] \quad \longrightarrow \quad \{0,1\}^{2^m}$$

  $$S(\prod \underline{x_{i_j}}) := \underline{x_{i_1}} \odot_2^{\flat} \ldots \odot_2^{\flat} \underline{x_{i_s}}$$

  zuordnen.

- Die Abbildungen F und S kann man auf Polynome (ein Polynom ist eine Summe von Monomen) erweitern, indem man für Monome $M_1, \ldots, M_n$ und das Polynom $\sum M_i$ definiert:

  $$F(\sum M_i) := F(M_1) \boxplus \cdots \boxplus F(M_n)$$

  bzw.

  $$S(\sum M_i) := S(M_1) \oplus_2^{\flat} \cdots \oplus_2^{\flat} S(M_n).$$

  Kann jedes Wort durch ein Polynom beschrieben werden? JA! Begründung siehe unten.

- Grad der Polynome:
  $\text{Grad}(\underline{0}) = -1, \text{Grad}(\underline{1}) = 0,$
  $\text{Grad}(\underline{x_{i_1}} \cdot \ldots \cdot \underline{x_{i_s}}) = s$
  M_i: verschiedene Monome $\Rightarrow \text{Grad}(\sum M_i) = \max\{\text{Grad}(M_i)\}$

Beispiel: für $m = 3$
$S(\underline{x_0}) \quad = 01010101$
$S(\underline{x_1}) \quad = 00110011$
$S(\underline{1} + \underline{x_0} \cdot \underline{x_1}) \quad = 11101110$

Lemma 7 [Lemma zur Repräsentation] *Jede boolsche Funktion auf $\mathbb{Z}_2^m$ läßt sich durch ein Polynom in $\underline{x_0}, \ldots, \underline{x_{m-1}}$ repräsentieren*

Beweis: (durch vollständige Induktion über m)
Ist $m = 0$, so können die beiden boolschen Funktionen durch $\underline{0}$ und $\underline{1}$ repräsentiert werden. Für alle Polynome $p(\underline{x_0}, \ldots, \underline{x_m}) \in \mathbb{Z}_2[\underline{x_0}, \ldots, \underline{x_m}]$ gilt:

$$p(\underline{x_0}, \ldots, \underline{x_m}) = p(\underline{x_0}, \ldots, \underline{x_{m-1}}, 0) + \Big(p(\underline{x_0}, \ldots, \underline{x_{m-1}}, 0) + p(\underline{x_0}, \ldots, \underline{x_{m-1}}, 1)\Big) \cdot \underline{x_m}.$$

Können also für ein $m \in \mathbb{N}_0$ alle boolschen Funktionen aus $\mathbb{Z}_2^m \to \mathbb{Z}_2$ durch Polynome aus $\mathbb{Z}_2[\underline{x_0}, \ldots, \underline{x_{m-1}}]$ repräsentiert werden (und damit auch $p(\underline{x_0}, \ldots, \underline{x_{m-2}}, 0)$ und $p(\underline{x_0}, \ldots, \underline{x_{m-2}}, 1)$), so kann auch jede boolschen Funktion aus $\mathbb{Z}_2^{m+1} \to \mathbb{Z}_2$ durch ein Polynom aus $\mathbb{Z}_2[\underline{x_0}, \ldots, \underline{x_m}]$ repräsentiert werden. □

Gemäß dem obigen Lemma gibt es somit bijektive Abbildungen[1] zwischen den folgenden drei Mengen:
(1e boolsche Funktionen $\mathbb{Z}_2^m \to \mathbb{Z}_2$
(2) Worte aus $\{0,1\}^{2^m}$
(3) Polynome aus $\mathbb{Z}_2[\underline{x_0}, \ldots, \underline{x_{m-1}}]$
Im folgenden wird (meistens) keine Unterscheidung mehr zwischen einem Polynom p, dem entsprechenden Wort $S(p)$ und der boolschen Funktion $F(p)$ gemacht.

Lemma 8 [Lemma zu Gewichten]

(i) $Gewicht(\underline{x_i}) = 2^{m-1}$

(ii) $Gewicht(\underline{x_{i_1}} \cdot \ldots \cdot \underline{x_{i_s}}) = 2^{m-s}$
für s paarweise verschiedene Indizes $i_1, \ldots, i_s$.

Der Beweis ist offensichtlich. □

Lemma 9 [Lemma zu Monomen] *Die Monome über* $\underline{x_0}, \ldots, \underline{x_{m-1}}$ *(für* $\mathbb{Z}_2^{2^m}$*) bilden eine Basis von* $\mathbb{Z}_2^{2^m}$.

Beweis:

- Es gibt $\binom{m}{k}$ Monome der Länge k und somit $\sum_{i=0}^{m} \binom{m}{i} = 2^m$ Monome.
- Die Monome spannen den ganzen Raum auf (Lemma 7)
- Offensichtlich $\dim(\mathbb{Z}_2^{2^m}) = 2^m$

$\Rightarrow$ Die Monome bilden eine Basis. □

Folgerung: Die Monome sind linear unabhängig.

1.6.4 Konstruktion von $\Re(r,m)$ - Codes

$\Re(r,m)$ Codes mit Grad r und Codewortlänge 2^m definiert man wie folgt:
x ist ein Codewort $\Leftrightarrow$ x wird durch ein Polynom mit Grad $\leq r$ repräsentiert[2].

Beispiele:

- $\Re(0,m)$ enthält nur $\underline{1} = \underbrace{1 \cdots 1}_{2^m}$ und $\underline{0} = \underbrace{0 \cdots 0}_{2^m}$ (Repetition Code)
- $\Re(1,m)$ enthält $m+1$ Generatoren: $1, \underline{x_0}, \underline{x_1}, \ldots, \underline{x_{m-1}}$
- $\Re(m-1,m)$ = Parity-Check Code
  Beweis:
  - $Gewicht(\underline{x_{i_1}} \cdot \ldots \cdot \underline{x_{i_s}}) = 2^{m-s}$
    also haben alle Generatoren von $\Re(m-1,m)$ ein gerades Gewicht.

[1] $S : (3) \to (2); F : (3) \to (1); S \circ F^{-1} : (1) \to (2)$
[2] Hier nocheinmal etwas formaler: $x \in \{0,1\}^{2^m}$ ist ein Codewort $\Leftrightarrow$ $\text{Grad}(S^{-1}(x)) \leq r$.

- Jede lineare Kombination von Codeworten mit geradem Gewicht hat gerades Gewicht.
- also: $\Re(m-1,m) \subseteq$ Parity-Check Codes
- $\Re(m-1,m)$ hat die Dimension 2^m-1, da nur der Basisvektor $\prod_{i=0}^{m-1} \underline{x_i}$ fehlt.
  Der Parity-Check Code hat auch die Dimension 2^m-1.
- $\Rightarrow \Re(m-1,m) =$ Parity-Check Codes $\square$

Lemma 10 *Monome mit positivem Grad $\leq r$ bilden eine Basis von $\Re(r,m)$.*
D.h. die Monome mit positivem Grad $\leq r$ sind Generatoren des $\Re(r,m)$ Codes.

Beweis Die Monome mit dem positivem Grad $\leq r$ sind linear unabhängig und spannen $\Re(r,m)$ auf. $\square$

1.6.5 Parity-Check-Matrix für Reed-Muller Codes

Theorem 11

(i) $\Re(r,m)$ *hat* $k = \sum_{i=0}^{r} \binom{m}{i}$ *Informationsbits.*

(ii) $\Re(r,m)^{\perp} = \Re(m-r-1,m)$ *(Anwendung: Konstruktion von Parity-Check-Matrix für* $\Re(r,m)$*).*

Beweis:

1. Die Basis von $\Re(r,m)$ enthält $\sum_{i=0}^{r} \binom{m}{i}$ Monomen.
2. - Sei $d^{\perp}$ die Dimension von $\Re(r,m)^{\perp}$
     $d^{\perp} = 2^m - \sum_{i=0}^{r} \binom{m}{i} = \sum_{i=r+1}^{m} \binom{m}{i} = \sum_{i=0}^{m-r-1} \binom{m}{i}$
     Also stimmen die Dimensionen überein.
   - Es reicht zu zeigen, daß für jedes Polynom p vom Grad $\leq m-r-1$ und q vom Grad $\leq r$ die Worte $S(p)$ und $S(q)$ orthogonal zueinander sind.
     $\text{Grad}(p \cdot q) \leq m-1$ und somit $p \cdot q \in \Re(m-1,m)$. Also ist das Gewicht$(S(p \cdot q))$ gerade (siehe Beispiel oben) und deswegen $= 0 \bmod 2$, womit $S(p) \perp S(q)$ gilt.

$\square$

Bemerkung: Offensichtlich $\Re(r,m) \subseteq \Re(m-r-1,m)$ für $r \leq m-r-1$. Andererseits $\Re(r,m)^{\perp} = \Re(m-r-1,m)$. Für "normale" lineare Räume (über den reelen Zahlen) $K \cap K^{\perp} = \{\vec{0}\}$.
$\Rightarrow$ die linearen Räume über endlichen Körper sehen etwas anders aus.

Konstruktion der Parity-Check-Matrix

nach dem Satz 11 und 6:
Parity-Check-Matrix von $\Re(r,m)$ = Generatormatrix von $\Re(m-r-1,m)$.
Generatormatrix von $\Re(m-r-1,m)$ ist bekannt: Die Zeilen sind Darstellungen von Monomen mit dem Grad $\leq m-r-1$.

1.7 Fehlerkorrektur bei Reed-Muller Codes

Wir benötigen noch eine geometrische Interpretation. Jetzt betrachten wir den linearen Raum $\mathbb{Z}_2^m$.

1.7.1 Definition: affine Unterräume

Affine r-Unterräume sind die Mengen $\vec{a}+K := \{\vec{a}+\vec{b} : \vec{b} \in K\}$, wobei K ein linearer Unterraum von $\mathbb{Z}_2^m$, $\dim(K) = r$ und $\vec{a} \in \mathbb{Z}_2^m$ ist.

- S sei ein affiner r-Unterraum, dann ist

$$S = \{\vec{x} : \vec{x} = \vec{a} + t_1\vec{b}_1 + \cdots + t_r\vec{b}_r, t_i \in \{0,1\}\},$$

  wobei $\vec{b}_1, \ldots, \vec{b}_r$ eine Basis von K ist.

- Die Anzahl der Punkte in einem affinen r-Unterraum ist 2^r.
  Beweis Jeder Punkt enspricht genau einer Kombination von $t_1, \ldots, t_r$:
  $a + t_1 \cdot \vec{b_1} + \cdots + t_r \cdot \vec{b_r} = a + t'_1 \cdot \vec{b_1} + \cdots + t'_r \cdot \vec{b_r}$
  $\Rightarrow (t_1 - t'_1) \cdot \vec{b_1} + \cdots + (t_r - t'_r) \cdot \vec{b_r} = \vec{0}$
  $\Rightarrow t_1 = t'_1, \ldots, t_r = t'_r$, weil $\vec{b_1}, \ldots, \vec{b_r}$ linear unabhängig sind. □

- $(\vec{a} + K) \cap (\vec{b} + K) \neq \emptyset \Rightarrow \vec{a} + K = \vec{b} + K$
  Beweis Sei $\vec{a} + \sum t_i\vec{b_i} = \vec{b} + \sum t'_i\vec{b_i}$. Dann ist $\vec{a} - \vec{b} = \sum(t'_i - t_i)\vec{b_i}$.
  $\Rightarrow$ für beliebiges $\vec{x} = \vec{a} + \sum s_i\vec{b_i}$ gilt:
  $$\vec{x} = (\vec{a} - \vec{b}) + \vec{b} + \sum s_i\vec{b_i} = \vec{b} + \sum(s_i + t'_i - t_i)\vec{b_i} \in \vec{b} + K$$
  also $\vec{a} + K \subseteq \vec{b} + K$.
  Ähnlich: $\vec{b} + K \subseteq \vec{a} + K$ □

- Fakt aus Algebra:
  es gibt eine Matrix H und Vektor $\vec{b}$, so daß $\vec{x} \in \vec{a} + K \Leftrightarrow H \cdot \vec{x}^T = \vec{b}^T$
  H enthält $m - r$ linear unabhängige Zeilen.

- 0-Unterräume – einzelne Punkte
  $m-1$-Unterräume – Hyperebene,

- es gibt insgesamt $2 \cdot (2^m - 1)$ Hyperebenen
  Beweis Sei $\dim(K) = m - 1$.
  - $|\vec{a} + K| = 2^{m-1}$
  - es gibt $2^m/2^{m-1} = 2$ affine Unterräume der Form $\vec{a} + K$
  - Anzahl der Hyperebenen = (Anzahl der linearen Unterräumen der Dimension $m-1$)$\cdot 2$
  - Anzahl der linearen Unterräume der Dimension $m-1$ =
    Anzahl der verschiedener Gleichungen von der Form
    $$\alpha_1 x_1 + \ldots + \alpha_m x_m = 0$$
    mit $\alpha_1, \ldots, \alpha_m$ nicht alle gleichzeitig 0 =
    $2^m - 1$ □

1.7.2 Darstellung von Unterräumen durch Wörter und Polynome

Sei L ein r-Unterraum.

- Wir betrachetn $\mathbb{Z}_2^m$ nun als m-dimensionalen Hypercube mit den Punkten:

$$\begin{aligned} p_0 &= 000 \ldots 000 \\ p_1 &= 000 \ldots 001 \\ p_2 &= 000 \ldots 010 \\ &\vdots = \vdots \\ p_{2^m-3} &= 111 \ldots 101 \\ p_{2^m-2} &= 111 \ldots 110 \\ p_{2^m-1} &= 111 \ldots 111 \end{aligned}$$

- Das folgende Wort aus $\{0,1\}^{2^m}$ charakterisiert L und wird charakteristische Funktion genannt: $\Phi_L = \phi_{2^m-1} \cdots \phi_1 \phi_0$, wobei $\phi_j = 1 \Leftrightarrow p_j \in L$
- Φ_L kann als Polynom interpretiert werden (s. Kap. 1.6.2), welches mit $\Psi_L \in \mathbb{Z}_2[x_0, \ldots, x_n]$ bezeichnet werden soll.

  Ist $p_i = y_0 y_1 \cdots y_{m-1}$ so folgt, daß

  $$\Psi_L(y_0, \ldots, y_{m-1}) = 1 \quad \Leftrightarrow \quad p_i \in L$$

  Das Polynom hat die folgende Form:
  $\vec{y} \in L \quad \Leftrightarrow \quad H \cdot \vec{y}^T = \vec{c}^T \quad \Leftrightarrow \quad \forall i : H_{i,\bullet} \cdot \vec{y}^T = c_i \quad \Leftrightarrow$
  $\forall i : (1 + c_i) + H_{i,\bullet} \cdot \vec{y}^T = 1 \quad \Leftrightarrow \quad \prod_i ((1 + c_i) + H_{i,\bullet} \cdot \vec{y}^T) = 1$
  Also ist $\Psi_L(x_0, \ldots, x_n) = \prod_i \left(1 + c_i + \sum_j h_{ij} x_j\right)$ ein Polynom vom Grad $\leq m - r$, da H diese Anzahl an Zeilen hat.

1.7.3 Charakterisierung von Reed-Muller Codes durch affine Unterräume

Lemma 12 *$\Re(r,m)$ wird durch die charakteristischen Funktionen der k-Unterräume aufgespannt, wobei $k \geq m - r$ ist.*

Beweis:

- die erwähnten charakteristischen Funktionen haben einen Grad $\leq r$, also gehören sie zu $\Re(r,m)$
- 1. Boolsche Funktion x_i entspricht dem Unterraum y_i=1.
  2. Seien L_1, L_2 Unterräume mit den charakteristischen Funktionen Φ_1, Φ_2. Dann ist $\Phi_1 \cdot \Phi_2$ die charakteristische Funktion von $L_1 \cap L_2$.
  3. Seien L_1 bzw. L_2 lineare Unterräume definiert durch die Matrizen H_1 bzw. H_2 (d.h. $\vec{y} \in L_k \Leftrightarrow H_k \cdot \vec{y}^T = \vec{0}; k = 1,2$).
     Dann ist $L_1 \cap L_2$ der durch die Matrix $\begin{bmatrix} H_1 \\ H_2 \end{bmatrix}$ definierte lineare Unterraum (linear abhängige Zeilen können entfernt werden).
     $\Rightarrow \dim(L_1 \cap L_2) \geq m - k_1 - k_2$, wobei $\dim(L_1) = m - k_1, \dim(L_2) = m - k_2$
  4. Monome vom Grad k entsprechen den Unterräumen der Dimension $m - k$

□

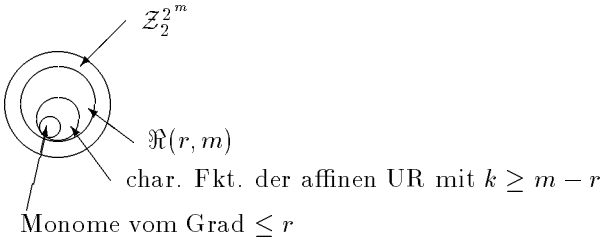


Abbildung 1.5: Übersicht der Räume

1.7.4 Grundideen der Fehlerkorrektur

$w = w_{2^m-1} \cdots w_0$ - empfangenes Wort (mit Fehlern)

- Sei L ein s-Unterraum, $s \leq r+1$, $\Phi = \phi_{2^m-1} \ldots \phi_0$ und $\vec{e}$ der Fehlervektor, d.h. $e_i = \begin{cases} 0 & w_i \text{korrekt} \\ 1 & w_i \text{fehlerhaft} \end{cases}$.

$$\text{PARITY}(L) := \sum \phi_i \cdot e_i \bmod 2$$

- es reicht $\text{PARITY}(L)$ von 0-Unterräumen zu berechnen, z.B. $L = \{p_j\}$. Dann ist $\Phi_L = \underset{}{0 \cdots 0} \underset{\downarrow}{\overset{j}{1}} 0 \cdots 0$ und $e_j = \text{PARITY}(L)$.

Lemma 13 *Sei L ein $r+1$-Unterraum, $\vec{w}$ ein Wort. Dann gilt:*

$$PARITY(L) = \vec{w} \cdot \Phi_L \bmod 2.$$

Beweis:

- Sei $\vec{w}$ ein Codewort (ohne Fehler). Es ist $\Re(r,m)^\perp = \Re(m-r-1,m)$. $(r+1)$-Unterräume entsprechen Polynomen maximalem Grad $m-r-1$ (s. 1.7.2) und gehören somit zu $\Re(m-r-1,m)$. Also: $\Phi_L \perp \vec{w}$ für jeden $(r+1)$-Unterraum L, d.h. $\vec{w} \cdot \Phi_L = 0 = \text{PARITY}(L)$.
- Sei $\vec{w} = \vec{v} + \vec{e}$, wobei $\vec{v} \in \Re(r,m)$ und $\vec{e}$ der Fehlervektor ist. Dann gilt:

$$\vec{w} \cdot \Phi_L = \vec{v} \cdot \Phi_L + \vec{e} \cdot \Phi_L = 0 + \text{PARITY}(L) = \text{PARITY}(L) \bmod 2.$$

Lemma 14 [zu Unterräumen]

(i) Jeder s-Unterraum L ist eine Untermenge von $2^{m-s}-1$ verschiedenen $(s+1)$-Unterräumen.

(ii) $\vec{x} \notin L \Rightarrow \vec{x}$ gehört zu genau einem von diesen Unterräumen.

Beweis:

(i.1) Wir nehmen an, daß L ein linearer s-Unterraum ist (ähnlich geht es auch für affine Unterräume). $(s+1)$-Unterräume die L enthalten haben die Form:

$$\text{Lin}(L, \vec{b}) = \left\{ \vec{a} + \alpha \cdot \vec{b} : \vec{a} \in L, \alpha \in \{0,1\} \right\},$$

da die Basis von L zur Basis des $(s+1)$-Unterraumes erweitert werden kann.

(i.2) Wann gilt $\text{Lin}(L, \vec{b}_1) = \text{Lin}(L, \vec{b}_2)$? Antwort: Wenn $\vec{b}_1 - \vec{b}_2 \in L$. Es gibt 2^s solche $\vec{b}_2$. **Folgerung:** Es gibt 2^{m-s} verschiedene Mengen der Form $\text{Lin}(L, \vec{b})$. Eine davon ist L selbst.

(ii) Falls $\vec{x}$ zu zwei solcher Unterräume L_1, L_2 gehört, dann ist $\vec{x} \in L_1 \cap L_2$. Weil $L \subseteq L_1 \cap L_2$ und $\vec{x} \notin L$ ist damit $\dim(L_1 \cap L_2) \geq s+1$. Es folgt, daß $L_1 = L_1 \cap L_2 = L_2$. □

1.7.5 Hauptsatz zur Fehlerkorrektur

Theorem 15 *Sei L ein s-Unterraum, $0 \leq s \leq r$ und die Anzahl der Fehler im empfangenen Wort $\vec{w}$ kleiner als 2^{m-r-1}. Dann gilt:*
Die Mehrheit der $(s+1)$-Unterräume, die L enthalten, haben die gleiche Parität wie L.

Beweis:
Sei t die Anzahl der Fehler. L ist eine Untermenge von $2^{m-s} - 1 \geq 2^{m-r} - 1 > 2t$ verschiedener $(s+1)$-Unterräumen. Ein Fehler auf Position i in $\vec{w}$ entspricht dem Punkt p_i in $\mathbb{Z}_2^m$.
PARITY(L) = Anzahl der Einsen in f_L und gleichzeitig im Fehlervektor (mod 2).
Es gibt nur einen $(s+1)$-Unterraum, der L und p_i enthält. Nur für diesen $(s+1)$-Unterraum wird PARITY durch p_i beeinflußt. Da es $\leq t$ Fehlerpositionen (bzw. Punkte) gibt, folgt, daß höchstens t von $(s+1)$-Unterräumen, die L enthalten, eine andere Parität als L haben. Das heißt der Rest enthält $> 2t - t = t$ Unterräume.
□

1.7.6 Algorithmus zur Fehlerkorrektur von $\Re(r, m)$

1. Die Parität von jedem $(r+1)$-Unterraum wird direkt (nach Lemma 13) berechnet.
2. Von den Paritäten der $(s+1)$-Unterräume werden die Paritäten der s-Unterräume rekursiv (nach Satz 15) berechnet.
3. Für jeden 0-Unterraum $\{p_i\}$ der Parität 1 wird das Bit i der Nachricht $\vec{w}$ geändert.

1.7.7 Schlußbemerkungen:

- Welche s-Unterräume Untermengen von welchen $(s+1)$-Untermengen sind, ist nicht offensichtlich. Aber es läßt sich einmal berechnen und dann immer wieder verwenden.
- Der Algorithmus ist leicht durch einen Schaltkreis zu implementieren.
- Durch die Auswahl von m und r werden die Eigenschaften von $\Re(r, m)$ festgelegt, aber man kann nicht für jedes n und t Codes der Länge n mit Fehlertoleranz $\geq t$ bekommen.
- Die Methode funktioniert für beliebig lange Codes.

1.7.8 Beispiel zur Fehlerkorrektur

Es sei das Wort $w = 0101\ 1101$ empfangen worden, wobei die versandte Nachricht ein Codewort des Reed-Muller Codes $\Re(1, 3)$ war.

Schritt I: Als erstes werden die $r + 1 = 2$-flats (Polonome mit Grad 1) betrachtet. Es wird PARITY$(L) = \sum_i w_i \cdot \phi_i \bmod 2$ berechnet.

Ψ_L	Φ_L	$PARITY(L)$	
x_0	0101 0101	4=0	lin. UR
x_0+1	1010 1010	1=1	
x_1	0011 0011	2=0	lin. UR
x_1+1	1100 1100	3=1	
x_2	0000 1111	3=1	lin. UR
x_2+1	1111 0000	2=0	
x_0+x_1	0110 0110	2=0	lin. UR
x_0+x_1+1	1001 1001	3=1	
x_0+x_2	0101 1010	3=1	lin. UR
x_0+x_2+1	1010 0101	2=0	
x_1+x_2	0011 1100	3=1	lin. UR
x_1+x_2+1	1100 0011	2=0	
$x_0+x_1+x_2$	0110 1001	3=1	lin. UR
$x_0+x_1+x_2+1$	1001 0110	2=0	

Schritt II: Als nächstes kommen die 1-flats (Polynome mit Grad 2) an die Reihe. Bei diesen wird PARITY(L) durch die Majorität der PARITY's der 3-flats gebildet, in denen das 2-flat als Teil enthalten ist (so ist z.B. das 2-flat x_0x_1 in den 3-flats x_0, x_1 und x_0+x_1+1 enthalten).

Ψ_L	Φ_L	Votes	PARITY(L)	
x_0x_1	0001 0001	0, 0, 1	0	lin. UR
$x_0(x_1+1)$	0100 0100	0, 1, 0	0	
$(x_0+1)x_1$	0010 0010	1, 0, 0	0	
$(x_0+1)(x_1+1)$	1000 1000	1, 1, 1	1	
x_0x_2	0000 0101	0, 1, 0	0	lin. UR
$x_0(x_2+1)$	0101 0000	0, 0, 1	0	
$(x_0+1)x_2$	0000 1010	1, 1, 1	1	
$(x_0+1)(x_2+1)$	1010 0000	1, 0, 0	1	
$x_0(x_1+x_2)$	0001 0100	0, 1, 0	0	lin. UR
$x_0(x_1+x_2+1)$	0100 0001	0, 0, 1	0	
$(x_0+1)(x_1+x_2)$	0010 1000	1, 1, 1	1	
$(x_0+1)(x_1+x_2+1)$	1000 0010	1, 0, 0	0	
x_1x_2	0000 0011	0, 1, 0	0	lin. UR
$x_1(x_2+1)$	0011 0000	0, 0, 1	0	
$(x_1+1)x_2$	0000 1100	1, 1, 1	1	
$(x_1+1)x_2$	1100 0000	1, 0, 0	0	
$x_1(x_0+x_2)$	0001 0010	0, 1, 0	0	lin. UR
$x_1(x_0+x_2+1)$	0010 0001	1, 0, 0	0	
$(x_1+1)(x_0+x_2)$	0100 1000	1, 1, 1	1	
$(x_1+1)(x_0+x_2+1)$	1000 0100	1, 0, 0	0	
$x_2(x_0+x_1)$	0000 0110	1, 0, 0	0	lin. UR
$x_2(x_0+x_1+1)$	0000 1001	1, 1, 1	1	
$(x_2+1)(x_0+x_1)$	0110 0000	0, 0, 1	0	
$(x_2+1)(x_0+x_1+1)$	1001 0000	0, 1, 0	0	
$(x_0+x_1)(x_0+x_2)$	0100 0010	0, 1, 0	0	lin. UR
$(x_0+x_1)(x_0+x_2+1)$	0010 0100	0, 0, 1	0	
$(x_0+x_1+1)(x_0+x_2)$	0001 1000	1, 1, 1	1	
$(x_0+x_1+1)(x_0+x_2+1)$	1000 0001	1, 0, 0	1	

Anmerkung: Es gilt $x_0(x_0+x_1)=x_0(x_1+1); x_0(x_0+x_2)=x_0(x_2+1);$
$x_0(x_0+x_1+x_2)=x_0(x_1+x_2+1); x_1(x_0+x_1)=x_1(x_0+1);$
$x_1(x_1+x_2)=x_1(x_2+1); x_1(x_1+x_2+x_3)=x_1(x_2+x_3+1);$
$x_2(x_0+x_2)=x_2(x_0+1); x_2(x_1+x_2)=x_2(x_1+1);$

$x_2(x_0+x_1+x_2) = x_2(x_0+x_1+1); (x_0+x_1)(x_1+x_2) = (x_0+x_1)(x_0+x_2+1);$
$(x_0+x_1)(x_0+x_1+x_2) = (x_2+1)(x_0+x_1);$
$(x_0+x_2)(x_1+x_2) = (x_0+x_1+1)(x_0+x_2);$
$(x_0+x_2)(x_0+x_1+x_2) = (x_1+1)(x_0+x_2);$
$(x_1+x_2)(x_0+x_1+x_2) = (x_0+1)(x_1+x_2).$

Schritt III: Im letzten Schritt werden die einzelnen Punkte betrachtet.

Φ_L	Votes	PARITY(L)
1000 0000	1, 1, 0, 0, 0, 0, 1	0
0100 0000	0, 0, 0, 0, 1, 0, 0	0
0010 0000	0, 1, 1, 0, 0, 0, 0	0
0001 0000	0, 0, 0, 0, 0, 0, 1	0
0000 1000	1, 1, 1, 1, 1, 1, 1	1
0000 0100	0, 0, 0, 1, 0, 0, 0	0
0000 0010	0, 1, 0, 0, 0, 0, 0	0
0000 0001	0, 0, 0, 0, 0, 1, 1	0

Fehlerkorrektur: Da bei $L = \{p_3\}$ gilt, daß PARITY(L) = 1 ist, muß das entsprechende Bit der empfangenen Nachricht korrigiert werden, welche damit zu 01010101 wird.

1.8 Zyklische Codes

Ziel: neue Klasse von guten Codes.

1.8.1 Definition: Zyklische Codes

Sei K ein linearer Code. K heißt zyklisch, falls:

$$v_0, \ldots, v_{n-1} \in K \quad \Leftrightarrow \quad v_{n-1} v_0 v_1 \cdots v_{n-2} \in K.$$

Beispiel: Even-Parity Code

1.8.2 Deutung von Polynomen

- Zyklische Codes über einem Körper F werden durch (formale) Polynome über F beschrieben: Code $a_0 a_1 \cdots a_k \longleftrightarrow$ Polynom $a_0 + a_1 x + a_2 x^2 + \cdots + a_k x^k$.
- Hier betrachten wir Polynome nicht als Funktionen, sondern als formale Ausdrücke, d.h. z.B. für $F = \mathbb{Z}_2$ sind die Polynome x^2 und x^4 nicht identisch obwohl sie dieselbe Funktion representieren.
- Polynome kann man: addieren, multiplizieren und dividieren
  (d.h., für Polynome $a(x)$ und $b(x)$ suchen wir Polynome $q(x)$ und $r(x)$, so daß

  1. $a(x) = q(x) \cdot b(x) + r(x)$ und
  2. Grad($r(x)$) < Grad($b(x)$)

  -Euklidischer Algorithmus!!)

1.8.3 Eigenschaften von Zyklischen Codes (als Polynome)

Lemma 16 *Sei K ein zyklischer Code mit Codelänge n. Dann gilt: ist $g(x) \in K$ und $Grad(g(x) \cdot q(x)) \leq n-1$, dann ist auch $g(x) \cdot q(x) \in K$*

Der **Beweis** folgt aus 2 Eigenschaften:

- die Summe von zwei Codeworten ist ein Codewort,
- $r(x) \in K$, $\text{Grad}(r(x)) < n-1 \Rightarrow x \cdot r(x) = \text{Zyklischer-Shift}(r(x)) \in K$. □

Theorem 17 [zum Generatorpolynom]
Sei K ein zyklischer Code der Länge n mit k Informationszeichen.
Dann $\exists g(x) \in K$:

1. $Grad(g(x)) = n-k$,
2. $\forall z(x): \quad z(x) \in K \Leftrightarrow \exists q(x) : Grad(q(x)) < k, z(x) = q(x) \cdot g(x)$,
3. *K besitzt die folgende Generatormatrix:*

$$G = \begin{bmatrix} g(x) \\ x \cdot g(x) \\ \vdots \\ x^{k-1} \cdot g(x) \end{bmatrix} =$$

$$\begin{bmatrix} g_0 & g_1 & g_2 & \cdots & g_{n-k} & 0 & 0 & \cdots & 0 \\ 0 & g_0 & g_1 & \cdots & & g_{n-k} & 0 & \cdots & 0 \\ 0 & 0 & g_0 & \cdots & & & g_{n-k} & \cdots & 0 \\ \cdots & \cdots & \cdots & & & & & \cdots & \cdots \\ & \vdots & & & & & & \vdots & \\ 0 & \cdots & \cdots & 0 & g_0 & g_1 & \cdots & \cdots & g_{n-k} \end{bmatrix}$$

Beweis: wähle $g(x) \in K$ mit $g(x) \neq 0$, so daß $\text{Grad}(g(x)) = s$ kleinst möglich ist.

- Wir zeigen (2).
  ($\Leftarrow$) folgt aus Lemma 16
  ($\Rightarrow$) Sei $z(x) \in K$. Finde $q(x)$ und $r(x)$, so daß $z(x) = q(x) \cdot g(x) + r(x)$ und $\text{Grad}(r(x)) < Grad(g(x))$, Also ist $r(x) = z(x) - q(x) \cdot g(x) \in K$ und letztlich $r(x) = 0$, da sonst $g(x)$ falsch gewählt wäre. Es folgt: $z(x) = q(x) \cdot g(x)$.
  Dann: $\text{Grad}(q(x)) = \underbrace{\text{Grad}(z(x))}_{<n} - \text{Grad}(g(x)) < n-s$
- Wir zeigen (3):
  Die Polynome $g(x), g(x) \cdot x, \ldots, g(x) \cdot x^{n-s-1}$
  - spannen K auf (nach Punkt 2),
  - sind linear unabhängig:
    $\alpha_0 g(x) + \alpha_1 g(x) \cdot x + \cdots + \alpha_{n-s-1} g(x) \cdot x^{n-s-1} = 0 \Rightarrow$
    $g(x) \cdot \sum_{i=0}^{n-s-1} \alpha_i x^i = 0 \quad \Rightarrow \quad (\alpha_0 \ldots \alpha_{n-s-1}) = \vec{0}$.
- Da die Basis $n-s$ Vektoren enthält, folgt, daß $k = n-s$ und daß somit (1) gilt.

□

Terminologie:

Jedes solche Codewort $g(x)$ heißt **Generatorpolynom**.

1.8.4 Codierung von zyklischen Codes

Verfahren 1 (nicht notwendigerweise systematisch)

$$[u_0 \cdots u_{k-1}] \cdot \begin{bmatrix} g(x) \\ x \cdot g(x) \\ \vdots \\ x^{k-1} \cdot g(x) \end{bmatrix} = (u_0 + u_1 \cdot x + \cdots + u_{k-1} \cdot x^{k-1}) \cdot g(x)$$

Verfahren 2 (systematisch)

Sei $u(x) = \sum_{i=n-k}^{n-1} u_i x^i$ ($u_{n-1}u_{n-2}\cdots u_{n-k}$ = Eingabefolge)
Wir suchen $q(x), r(x), \text{Grad}(r(x)) < n-k$, so daß $u(x) = q(x) \cdot g(x) + r(x)$
Dann definieren wir Codewort$(u_{n-1}u_{n-2}\cdots u_{n-k}) = u(x) - r(x)$.
Eigenschaften:

- $u(x) - r(x) = q(x) \cdot g(x)$, also ist $u(x) - r(x)$ ein Codewort.
- Die Koeffizienten von $x^{n-1}, \ldots, x^{n-k}$ sind bei $u(x)$ und $u(x) - r(x)$ gleich.
- Hardware-Implementation von Multiplikation und Division der Polynome ist einfach. Dadurch läßt sich $u(x) - r(x)$ leicht berechnen.

1.8.5 Generatorpolynome für zyklische Codes

Theorem 18

(i) Sei K eine zyklische (n,k)-Codierung über $\mathbb{Z}_2$, $g(x)$ ein Generator von K. Dann teilt $g(x)$ das Polynom $x^n - 1$ ($g(x) | x^n - 1$).

(ii) Jedes Polynom $g(x)$, das $x^n - 1$ teilt, ist ein Generator eines zyklischen Codes.

Beweis:

1. Es $\exists q(x), r(x) : x^n - 1 = q(x) \cdot g(x) + r(x)$, $\text{Grad}(r(x)) < \text{Grad}(g(x)) =: n-k$.
   Ist $q(x) := \sum_{i=0}^{k} q_i x^i$, dann ist
   $q(x) \cdot g(x) = \underbrace{(q_0 + \cdots + q_{k-1} \cdot x^{k-1}) \cdot g(x)}_{=:w(x) \in K} + \underbrace{q_k}_{\neq 0} x^k g(x)$
   K ist zyklisch, also ist $w'(x) := x^k \cdot g(x) - x^n + 1 = \text{Shift}(x^{k-1} g(x)) \in K$
   $\Rightarrow r(x) = -q(x) \cdot g(x) + x^n - 1 = -(w(x) + x^k g(x)) + x^n - 1) = w(x) - w'(x) \in K$
   Das einzige Codewort mit einem Grad $< \text{Grad}(g(x))$ ist das Nullpolynom also $r(x) = 0$ und $g(x)$ teilt $x^n - 1$.

2. Teile $g(x)$ das Polynom $x^n - 1$ mit $\text{Grad}(g(x)) = s$.
   Dann ist $K := \{q(x) \cdot g(x) : q(x)$ ist Polynom mit $\text{Grad}(q(x)) < n - s\}$

   - ein linearer Unterraum:
     $q_1(x) \cdot g(x) + q_2(x) \cdot g(x) = \underbrace{(q_1(x) + q_2(x))}_{\text{Grad} < n-s} \cdot g(x) \in K$
     $\text{Grad}(q_1(x) + q_2(x)) \leq \text{Grad}(q_1(x)) + \text{Grad}(q_2(x))$
   - zyklisch:
     Sei $w(x) = q(x) \cdot g(x)$. Dann:
     - $\text{Grad}(w(x)) < n-1 \Rightarrow \text{Shift}(w(x)) = x \cdot w(x) = \underbrace{(x \cdot q(x))}_{\text{Grad} < n-s} \cdot g(x) \in K$
     - $\text{Grad}(w(x)) = n-1$, $w(x) = a_{n-1} x^{n-1} + r(x) \Rightarrow$
       $\text{Shift}(w(x)) = x \cdot w(x) - a_{n-1} x^n + a_{n-1} = x \cdot q(x) \cdot g(x) - a_{n-1}(x^n - 1) =$
       $x \cdot q(x) \cdot g(x) - a_{n-1} \cdot h(x) \cdot g(x) = g(x) \cdot (\ldots) \in K$ □

Folgerung

Suche nach zyklischen Codes ≡ Zerlegung von Polynomen x^n-1

1.8.6 Orthogonalpolynom und Parity-Check-Matrix für zyklische Codes

Sei K ein zyklischer (n,k)-Code über $\mathbb{Z}_2$ und $g(x)$ ein Generator von K. Dann heißt $h(x) := \frac{x^n-1}{g(x)}$ **Orthogonal-** oder auch **Parity-Check-Polynom** des Codes K.

Zum Merken: Sei $g(x) = \sum_{j=0}^{n-k} g_j x^j$ und $h(x) = \sum_{j=0}^{k} h_j x^j$, dann ist $h(x) \cdot g(x) = \sum_{i=0}^{n} (\sum_{j=0}^{i} h_j g_{i-j}) x^i = x^n - 1$, also sind die Summen $\sum_{j=0}^{i} h_j \cdot g_{i-j}$ gleich Null für $i \notin \{0, n\}$.

Theorem 19 [zum Orthogonalpolynom]
Sei $h(x) = \sum_{i=0}^{k} h_i x^i$ das Orthogonalpolynom des (n,k)-Codes K über $\mathbb{Z}_2$ (mit $h_k = 1$). Dann hat K die folgende Parity-Check-Matrix:

$$H = \begin{bmatrix} 0 & & & \cdots & 0 & 1 & h_{k-1} & \cdots & h_1 & h_0 \\ 0 & & \cdots & 0 & 1 & h_{k-1} & & & h_0 & 0 \\ & & & & & \vdots & & & & \\ 1 & h_{k-1} & \cdots & & h_0 & 0 & \cdots & & & 0 \end{bmatrix} \in M(n-k, n; \mathbb{Z}_2)$$

Beweis:

1. $H \cdot g(x)^T = \vec{0}$:
   $H_{1,\bullet} \cdot g(x)^T = [h_{n-1} \ldots h_0] \cdot [g_0 \ldots g_{n-1}]^T = \sum_{i=0}^{n-1} h_i g_{n-1-i} =$
   „Koeffizient von x^{n-1} bei $h(x) \cdot g(x) = x^n - 1$ “$= 0$
   $H_{2,\bullet} \cdot g(x)^T = [h_{n-2} \ldots h_0 0] \cdot [g_0 \ldots g_{n-1}]^T = \sum_{i=0}^{n-2} h_i g_{n-2-i} =$
   „Koeffizient von x^{n-2} “$= 0$
   ...
2. Ähnliche Betrachtung führt dazu, daß auch $H \cdot (x^i \cdot g(x))^T = \vec{0}$ ist.
   Also gilt für alle $w(x) \in K$, daß $H \cdot w(x)^T = 0$.
3. H enthält $n-k$ linear unabhängige Zeilen, also:
   H entspricht einem k-dimensionalen Unterraum W. Es ist $K \subseteq W$. Außerdem haben K und W die gleiche Dimension. Also ist $W = K$. □

1.9 Fehlerkorrektur von linearen und zyklischen Codes

1.9.1 Allgemeine Fehlerkorrektur für lineare Codes

$K + x_6$
$K + x_5$
$K + x_4$
K
$K + x_1$
$K + x_2$
$K + x_3$

Sei K ein linearer Code und $x \notin K$. Die Menge

$$K + x := \{y + x : y \in K\}$$

heißt Nebenklasse von K (siehe z.B. nebenstehende Abbildung).

Die Nebenklassen partitionieren den gesamten Raum Γ^n, wobei n die Codewortlänge und Γ der betrachtete Körper ist.

Eigenschaften:

- $(K + x) \cap (K + y) \neq \emptyset \Rightarrow K + x = K + y$,
- jede Nebenklasse $K + x$ enthält genauso viele Elemente wie K.

Algorithmus: Wähle für jede Nebenklasse N ein e_N (mit minimalem Gewicht), so daß $K + e_N = N$. Empfangen wir w, *dann:* finden wir die Nebenklasse N, für die $w \in N$ ist und geben $w - e_N$ aus.

Nachteil: Es könnte sehr viele Nebenklassen geben. Der Algorithums benötigt eine Look-up-Tabelle (hoher Platzbedarf, langsam).

1.9.2 Syndrompolynom für zyklische Codes

Sei K ein zyklischer Code mit Generatorpolynom $g(x)$ und $w(x) = q(x)g(x) + s(x)$ die empfangene Nachricht mit $\text{Grad}(s(x)) < \text{Grad}(g(x))$. Dann heißt $s(x)$ **Syndrompolynom** von $w(x)$.

Bemerkung: Tritt bei der Übertragung ein Fehler auf und wird dadurch $w(x) + e(x)$ übertragen, so haben $w(x) + e(x)$ und $e(x)$ das gleiche Syndrompolynom. Da $g(x)$ das Polynom $w(x) - s(x)$ teilt – womit $w(x) - s(x) \in K$ ist – kann eine mögliche Fehlerkorrektur dadurch durchgeführt werden, daß ein Fehlerpolynom $f(x)$ gesucht wird, welches das Syndrompolynom $s(x)$ hat und unter diesen minimales Gewicht hat. Das korrigierte Wort wäre dann $w(x) + e(x) - f(x)$.

Andersherum betrachtet: Ist bekannt, daß der Code t Fehler korrigieren kann, so bildet man alle Polynome mit bis zu t von Null verschiedenen Koeffizienten und berechnet von diesen die Syndrompolynome. In dieser Tabelle kann man dann nachsehen, welche Fehlerkorrektur durchgeführt werden soll.

1.9.3 Syndrom-Fehlerkorrektur für Zyklische Codes

Da der Code zyklisch ist, braucht man nur zu überprüfen, ob das **erste** Zeichen des übertragenen Wortes falsch ist, es gegebenenfalls korrigieren und das so erhaltene Wort shiften, so daß wiederum nur das erste Zeichen korrigiert werden muß. Nach Wortlänge-vielen Shifts ist dann das Wort vollständig korrigiert. Vorteil: Anstatt alle Polynome mit bis zu t von Null verschiedenen Koeffizienten zu betrachten, betrachtet man nur noch die Polynome dieser Menge, deren Grad $n-1$ ist (Verbesserung von $\sum_{i=1}^{t}(|\Gamma|-1)^i \cdot \binom{n}{i}$ auf $\sum_{i=0}^{t-1}(|\Gamma|-1)^i \cdot \binom{n}{i-1}$).

Algorithmus für $\Gamma = \{0,1\}$ und 1-Fehlerkorrektur (Meggit Decoder):
Sei $w(x)$ die empfangene Nachricht und $s(x)$ das Syndrom für x^{n-1}.

for $i := 0$ **to** $n-1$ **do**
  if Syndrom$(w(x)) = s(x)$
    then $w(x) :=$ `Shift`$(w(x) - x^{n-1})$
    else $w(x) :=$ `Shift`$(w(x))$

Die Syndrome müssen nicht jedesmal neu berechnet werden, weil:

Lemma 20 *Syndrom*(`Shift`$(w(x))) = x \cdot$ *Syndrome*$(w(x)) \bmod g(x)$

Beweis: Sei $w(x) = q(x) \cdot g(x) + s(x)$
Shift$(w(x)) = x \cdot w(x) - w_{n-1}x^n + w_{n-1} = x \cdot w(x) - w_{n-1}(x^n - 1) =$
$x \cdot q(x) \cdot g(x) + x \cdot s(x) - w_{n-1} \cdot h(x) \cdot g(x) = x \cdot s(x) + g(x) \cdot [\ldots] = x \cdot s(x) \bmod g(x)$
□

1.9.4 Büschelstörungen und zyklische Codes

Beobachtungen

Sei K ein zyklischer (n,k)-Code, die Anzahl der aufgetretenen Fehler $\leq t$ und $w(x)$ die empfangene Nachricht.

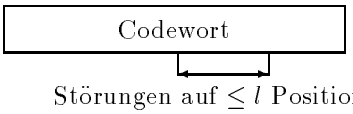


Abbildung 1.6: Büschelstörungen

1. Gewicht(Syndrom($w(x)$)) $\leq t \Rightarrow$ Syndrom($w(x)$) = Fehlermuster von $w(x)$
   Beweis:
   Sei $e(x) = q(x) \cdot g(x) + s(x)$ der Fehler, wodurch $s(x)$ auch das Syndrompolynom von $w(x)$ ist.
   Es ist $s(x) - e(x) \in K$ und damit Gewicht$(s(x) - e(x)) \leq$ Gewicht$(s(x)) +$ Gewicht$(e(x)) \leq 2t$. Da die Distanz zwischen Nullvektor und $s(x) - e(x)$ entweder 0 oder $\geq 2t + 1$ ist, folgt, daß das Gewicht von $(s(x) - e(x)) = 0$ ist, d.h. $e(x) = s(x)$. □

2. Seien nun alle Fehler im Fehlerpolynom $e(x)$ auf $n - k$ benachbarten Positionen. Dann existiert ein i mit Syndrom(Shift$^i(w(x))$) = Shift$^i(e(x))$ und Gewicht(Shift$^i(e(x))) \leq t$.
   Beweis: Wir shiften $w(x)$ bis alle Fehler auf den ersten $n - k$ Positionen stehen. Dann hat das aktuelle Fehlerpolynom den Grad $n - k - 1$, welcher kleiner als der Grad des Generatorpolynom $g(x)$ ist. Somit ist das Fehlerpolynom gleich dem Syndrompolynom. Da die Anzahl der Fehler $\leq t$ ist, folgt die Behauptung. □

Fehlerkorrektur bei Büschelstörungen

Gegeben sei ein zyklischer (n, k)-Code, der t Fehler korrigieren kann. Die Tatsache, daß der Code t Fehler korrigieren kann, beduetet nicht, daß das algorithmisch einfach ist! Manchmal ist die Korrekturprozedur sehr aufwendig und deswegen praktisch unbrauchbar. Der Meggit Decoder hilft auch nicht viel, weil er eine große Datei von Syndromen für große n und t benötigt. Es läßt sich aber zeigen, daß, falls alle t Fehler in einer Region der Länge $n - k$ liegen, dann die Korrektur einfach ist:

Algorithmus
for $i := 0$ **to** $n - 1$ **do**
  if Gewicht(Syndrom($w(x)$)) $\leq t$
    then `return`(`Shift`$^{-i}(w(x) -$ Syndrom$(w(x)))$)
    else $w(x)$:=Shift$(w(x))$

1.9.5 Erkennung von Büschelstörungen bei zyklischen Codes

Theorem 21 *Sei K ein zyklischer (n, k)-Code. Dann* **erkennt** *K Büschelstörungen der Länge $\leq n - k$.*

Beweis: Sei $e(x)$ das Fehlerpolynom mit $e(x) = x^i \cdot a(x)$, wobei Grad$(a(x)) \leq n - k - 1$. Ist $w(x)$ die empfangene Nachricht, so ist $w(x) - e(x)$ ein Codewort.

- Es ist $e(x) \notin K$,
  denn: angenommen $e(x) \in K$. Dann ist `Shift`$^{-i}(e(x)) = a(x) \in K$. Dies ist ein Widerspruch, weil Grad$(a(x)) < n - k$ und $a(x) \neq 0$ und somit $a(x) \notin K$ ist.

- Es ist $w(x) \notin K$,
  denn sonst würde gelten, daß $(w(x) - e(x)) - w(x) = e(x) \in K$ ist.

- Also ist $w(x)$ kein Code und die Störung wird erkannt. □

Beispiel:

Zyklische Hamming Codes der Länge 7 (4 Informationsbits) erkennen Büschelstörungen der Länge 3.

1.9.6 Golay Code

Der Golay Code ist der zyklische Code der Länge 23 mit dem Generatorpolynom:

$$1 + x^2 + x^4 + x^5 + x^6 + x^{10} + x^{11}$$

Eigenschaften:

- Die minimale Distanz zwischen zwei Codeworten ist 7 (nicht trivial !!!).
- Jedes Wort der Länge 23 hat eine Distanz ≤ 3 von einem Codewort (einfache Rechnung).
- Neben Hamming- und Repetition-Codes ist der Golay Code der einzige perfekte **binäre** Code, der noch existiert (existieren kann). Der Beweis hierzu ist sehr schwierig!
- Der Code ist praktisch. Er wird oft benutzt.

1.10 BCH Codes

Wir werden im folgenden die Elemente von endlichen Körpern durchnumerieren und diese Nummer binär codieren. Die Bits der Binärfolge eines Körperelements kann man dann als Koeffizienten eines Polynoms betrachten. Wir werden sehen: die entstandenen Polynome besitzen algebraische Eigenschaften, die eine einfache Fehlerkorrektur ermöglichen.

1.10.1 Endliche Körper

(Weitere Informationen, Beispiele und Beweise z.B. im Buch von Adámek)

Beispiele

- $\mathbb{Z}_p = (\{0, \ldots, p-1\}, \oplus_p, \odot_p)$ (Addition und Multiplikation modulo p) wobei p eine Primzahl ist.
- $\mathbb{Z}_p[x]/q[x]$ - Ring der Polynome modulo $q(x)$ wobei $q(x)$ Primpolynom ist, d.h. $q(x)$ ist nicht zerlegbar: es existieren keine Polynome $r(x), s(x) \in \mathbb{Z}_p[x]$, so daß $q(x) = r(x) \cdot s(x) \in \mathbb{Z}_p[x]$.
  Jedes Element des Körpers ist ein Polynom vom Grad $<$ Grad$(q(x))$. Dieses kann durch die Folge der Koeffizienten dargestellt werden (Folge der Länge Grad$(q(x))$).
  Beispiel: $q(x) := x^3 + x + 1 \in \mathbb{Z}_2[x]$, Körperelemente:
  x
  x^2
  $x^3 = x + 1$
  $x^4 = x^2 + x$
  $x^5 = x^3 + x^2 = x^2 + x + 1$
  $x^6 = x^3 + x^2 + x = x^2 + 1$
  $x^7 = x^3 + x = 1$

Es gibt 2^3 Möglichkeiten ein Polynom vom Grad ≤ 2 zu bilden. Somit gibt es 8 Elemente in $\mathbb{Z}_2[x]/_{x^3+x+1} = \{0\} \cup \{x^i : i = 1, \ldots, 7\}$

Theorem 22

(i) Sei p eine Primzahl, $i \in \mathbb{N}$ und $k := p^i$. Es gibt (bis zu Isomorphismus) genau einen Körper mit k Elementen.

(ii) Ist $k \neq p^i$ für eine Primzahl p und $i \in \mathbb{N}$, so existiert kein Körper mit k Elementen.

Notation:

GF(k) - endlicher Körper mit k Elementen.

Theorem 23 *Jeder Körper $GF(p^i)$ hat die Form $\mathbb{Z}_p[x]/q(x)$ für ein entsprechendes Polynom $q(x)$ mit Koeffizienten aus $\mathbb{Z}_p$.*

Theorem 24 [Kleiner Satz von Fermat]
Für jedes $\beta \in GF(k)\backslash\{0\}$ gilt: $\beta^{k-1} = 1$.

Beweis: Es gibt $k-1$ Elemente in der multiplikativen Gruppe von GF(k) und für jede endliche Gruppe $(G, \cdot)$ und jedes Element $x \in G$ gilt: $x^{|G|} = 1$. □

Definition (Minimalpolynom):

Sei $\beta \in \mathrm{GF}(p^i)$. Ein Polynom $w(x)$ heißt **Minimalpolynom** für β, falls:

(i) die Koeffizienten von $w(x)$ aus $\mathbb{Z}_p$ sind,

(ii) in GF(p^i) gilt, daß $w(\beta) = 0$,

(iii) der Grad von $w(x)$ minimal ist unter allen Polynomen, die (i) und (ii) erfüllen.

Nach Satz 24 ist das Minimalpolynom wohl definiert, d.h. für jedes Polynom existiert ein Minimalpolynom.

Definition (Erzeugendes Element in Körper):

Sei F ein Körper. Ein Element $\beta \in F$ heißt **erzeugendes Element** (oder **Generator**) von F genau dann, wenn $F = \{0\} \cup \{\beta^i \mid i \in \mathbb{N}\}$.
Da $\beta^{|F|-1} = 1$ und somit $\beta^{|F|} = \beta$ ist, folgt, daß $\{\beta^i \mid i \in \mathbb{N}\} = \{\beta^i \mid i \leq |F| - 1\}$

Theorem 25 *Ist K ein endlicher Körper, so existiert ein Generator für K.*

Darstellung von Elementen aus endlichen Körper

Sei $F = \mathbb{Z}_p[x]/q(x)$. Elemente in F sind Restpolynome mit Grad $<$ Grad($q(x)$).
$\rightsquigarrow$ Die Polynome werden als Folge der Koeffizienten dargestellt.

Beispiel: $p = 2$, $q(x) = x^3 + x + 1$

$\beta = 0 + 1 \cdot x + 0 \cdot x^2$	$\rightsquigarrow$	010
$\beta^2 = 0 + 0 \cdot x + 1 \cdot x^2$	$\rightsquigarrow$	001
$\beta^3 = 1 + 1 \cdot x + 0 \cdot x^2$	$\rightsquigarrow$	110
$\vdots$	$\vdots$	$\vdots$
$\beta^7 = 1 + 0 \cdot x + 0 \cdot x^2$	$\rightsquigarrow$	100

Die Summe zweier Polynome führt zur bitweisen Summe der Folgen,
z.B $(1 + x^2) + (1 + x + x^2) = x \quad \rightsquigarrow \quad (101)+(111)=(010)$.
Das Produkt zweier Folgen muß als Produkt der Polynome in $\mathbb{Z}_p[x]/q(x)$ interpretiert werden!!

1.10.2 BCH Codes die 1 Fehler korrigieren

Definition:

Sei $F = \mathrm{GF}(k)$ mit $k = p^n$ und $\beta \in F$ ein Generator. Wir betrachten $\mathbb{Z}_p^{k-1}$ (Folgen der Länge $k-1$ aus Elementen von $\mathbb{Z}_p$, wobei jede solche Folge $a = a_0 \cdots a_{k-2}$ als Polynom $a(x) = \sum_{i=0}^{k-2} a_i x^i$ betrachtet wird).
Dann ist $K = \{w \in \mathbb{Z}_p^{k-1} \mid w(\beta) = 0\}$ ein **BCH Code**.

Beispiel:

$F := \mathbb{Z}_2[x]/x^3 + x + 1$, $\beta := (010)$, d.h. β ist Restpolynom $0 + 1 \cdot x + 0 \cdot x^2 = x$.
(Minimalpolynom von β ist $x^3 + x + 1$.) $K = \{w \in \{0,1\}^7 : w(\beta) = 0\}$
$w(\beta) = w_0 \cdot (100) + w_1 \cdot (010) + w_2 \cdot (001) + w_3 \cdot (110) + w_4 \cdot (011) + w_5 \cdot (111) + w_6 \cdot (101) =$
$(w_0 + w_3 + w_5 + w_6 \ , \ w_1 + w_3 + w_4 + w_5 \ , \ w_2 + w_4 + w_5 + w_6)$.
$K = \{w_0 \cdots w_6 : w_0 + w_3 + w_5 + w_6 = 0, w_1 + w_3 + w_4 + w_5 = 0, w_2 + w_4 + w_5 + w_6 = 0\}$.

Theorem 26 *BCH Codes sind zyklisch.*

Beweis: Sei $m(x)$ das Minimalpolynom für β und $F = \mathrm{GF}(k)$.

- $w \in K \Leftrightarrow m(x) | w(x)$.
  Sei $w(x) = m(x) \cdot v(x) + r(x)$, wobei $\mathrm{Grad}(r(x)) < \mathrm{Grad}(m(x))$.
  Dann ist $0 = w(\beta) = m(\beta) \cdot v(\beta) + r(\beta) = 0 \cdot v(\beta) + r(\beta) = r(\beta)$.
  Da $m(x)$ das Minimalpolynom für β und $r(\beta) = 0$ ist, folgt, daß $r(x) = 0$ und $m(x) | w(x)$.

- Somit ist $K = \{v(x) \cdot m(x) : \mathrm{Grad}(v(x) \cdot m(x)) < k - 1\}$ und damit $m(x)$ ein Generator.

- $m(x) | x^{k-1} - 1$.
  Sei $x^{k-1} - 1 = m(x) \cdot u(x) + s(x)$, wobei $\mathrm{Grad}(s(x)) < \mathrm{Grad}(m(x))$.
  Dann ist $\beta^{k-1} - 1 = 0$ (Satz 24), also $s(\beta) = 0$. Da $m(x)$ Minimalpolynom für β ist, ist somit $s(x) = 0$. Es folgt $x^{k-1} - 1 = m(x) \cdot u(x)$.

- Nach Theorem 1.8.5 ist damit der Code zyklisch. □

Theorem 27 [Hamming Codes versus BCH Codes]
Sei β ein Generator von GF(2^m), $n = 2^m - 1$. Dann ist $K := \{w \in \mathbb{Z}_2^n \mid w(\beta) = 0\}$ ein Hamming Code.

Beweis: $w = w_0 \cdots w_{n-1}$ ist ein Codewort $\Leftrightarrow w_0 + w_1\beta + \cdots + w_{n-1}\beta^{n-1} = 0$.
Anders formuliert: $w = w_0 \cdots w_{n-1}$ ist ein Codewort $\Leftrightarrow$

$$[1 \ \beta \ \beta^2 \ \cdots \ \beta^{n-1}] \cdot \begin{bmatrix} w_0 \\ w_1 \\ \vdots \\ w_{n-1} \end{bmatrix} = \vec{0}^T$$

Jedes β^i ist als Vektor der Länge m darstellbar. Sei $H := [1^T \ \beta^T \ (\beta^2)^T \ \cdots (\beta^{n-1})^T]$.
Dann $w \in K \Leftrightarrow H \cdot w^T = \vec{0}^T$.
Es reicht zu zeigen, daß keine zwei Spalten von H gleich sind, d.h. $\beta^i \neq \beta^j$ für $i, j \leq n, i \neq j$. Ist $\beta^i = \beta^j$, so ist $\beta^{i-j} = 1$. Ohne Einschränkung sei $i - j > 0$. Dann ist $|\{\beta^k : k \in \mathbb{N}\}| \leq i - j$. Da aber β ist Generator von $\mathrm{GF}(2^m)$ ist, ist andererseits $|\{\beta^k : k \in \mathbb{N}\}| = n$. Widerspruch. Also sind keine 2 Spalten gleich. □

1.10.3 BCH Codes für 2 Fehler

Vorschlag 1:

$K = \{w : w(\beta) = 0 \text{ und } w(\beta^2) = 0\}$

Dieser Vorschlag ist falsch, denn ist $F = \mathrm{GF}(p^m)$ und $w \in \mathbb{Z}_p[x]$, so folgt aus $w(\beta) = 0$, daß $w(\beta^p) = 0$ ist:

Beweis Für alle Elemente $a, b \in F$ gilt: $(a \oplus_F b)^p = \sum_{i=0}^{p} \binom{p}{i} a^i \odot_F b^{p-i} = a^p \oplus_F b^p + p \odot_F (\ldots) = a^p \oplus_F b^p$.
Für $x \in \mathbb{Z}_p$ gilt $x^p = x$ (Fermat Satz!).
Also gilt: $0 = (w(\beta))^p = (w_0\beta + \ldots + w_{n-1}\beta^{n-1})^p = (w_0\beta)^p + \ldots + (w_{n-1}\beta^{n-1})^p = w_0^p\beta^p + \ldots + w_{n-1}^p(\beta^p)^{n-1} = w_0\beta^p + \ldots + w_{n-1}(\beta^p)^{n-1} = w(\beta^p)$.
Für den Fall $p = 2$ folgt die Behauptung.

Vorschlag 2: (BCH Codes: Korrektur von 2 Fehlern)

$K = \{w \in \mathbb{Z}_2^n : w(\beta) = 0 \text{ und } w(\beta^3) = 0\}$, für $n = 2^m - 1$, β – Generator des Körpers $\mathrm{GF}(2^m)$.

Parity-Check-Matrix:

$$H = \begin{bmatrix} 1 & \beta & \beta^2 & \ldots & \beta^{(n-1)} \\ 1 & \beta^3 & \beta^6 & \ldots & \beta^{3(n-1)} \end{bmatrix}$$

Beispiel

$F := \mathrm{GF}(16)$, β sei das Element mit Minimalpolynom $M_1(x) = x^4 + x + 1$. Damit besitzt β^3 das Minimalpolynom $M_2(x) = x^4 + x^3 + x^2 + x + 1$ (nicht offensichtlich!).

Es gilt : $w(\beta) = 0 \Leftrightarrow M_1(x) | w(x)$ und $w(\beta^3) = 0 \Leftrightarrow M_2(x) | w(x)$,
also: $w \in K \Leftrightarrow M_1(x) \cdot M_2(x) | w(x)$, weil $M_1(x)$ und $M_2(x)$ Primpolynome sind.

Somit ist K zyklisch mit Generator $M_1(x) \cdot M_2(x) = x^8 + x^7 + x^6 + x^4 + 1$!
$K = \{w(x) \mid \exists q(x) : w(x) = (x^8 + x^7 + x^6 + x^4 + 1) \cdot q(x)\}$

1.10.4 Fehlerkorrektur

Sei $w = w_0 \cdots w_{n-1}$ ein BCH-Codewort für Körper $\mathrm{GF}(2^m)$ $(2^m = n + 1)$,

$e = 0 \cdots 0 \underset{\downarrow}{1}^{i} 0 \cdots 0 \underset{\downarrow}{1}^{j} 0 \cdots 0$ und $w + e$ eine empfangene Nachricht. Dann

$$H \cdot (w + e)^T = H \cdot e^T = \begin{bmatrix} \beta^i + \beta^j \\ \beta^{3i} + \beta^{3j} \end{bmatrix}$$

Bekannt: $s_1 = \beta^i + \beta^j$, $s_2 = \beta^{3i} + \beta^{3j}$
Gesucht: i, j

- $s_1^3 = \beta^{3i} + 3\beta^{2i}\beta^j + 3\beta^i\beta^{2j} + \beta^{3j} = \beta^{3i} + (\beta^i + \beta^j)\beta^i\beta^j + \beta^{3j}$
- $s_1^3 = s_2 + \beta^i\beta^j s_1$, also $\beta^i\beta^j = s_1^2 + s_2 \cdot s_1^{-1}$
- $x^2 - (x_1 + x_2)x + x_1x_2 = 0$ hat Lösungen x_1, x_2

  Also reicht es, die Gleichung $x^2 - s_1x + (s_1^2 + s_2 \cdot s_1^{-1}) = 0$ zu lösen.
  Die Lösungen sind β^i, β^j.

1.10.5 BCH Codes - allgemeine Definition

Definition:

Sei F ein endlicher Körper.
$|F|$ und n - teilerfremd.
β - Element vom Grad n in einer Erweiterung von F (d.h. $\beta^n = 1$ und $\beta^i \neq 1$ für $i < n$.)

Dann heißt $K := \{w \in F^n \mid w(\beta) = w(\beta^2) = \ldots = w(\beta^{2t}) = 0\}$
t-fehlerkorrigierender BCH Code

Bemerkung:

Es gibt immer ein solches β (Satz aus der Algebra).

Beispiele:

- Sei $F = \mathbb{Z}_2$. Dann gilt: $w(\beta^i) = 0 \Rightarrow w(\beta^{2i}) = 0$.
  Die Bedingungen $w(\beta^2) = 0, w(\beta^4) = 0, \ldots$ sind zur Definition von BCH Codes über Erweiterungen von $\mathbb{Z}_2$ überflüssig.

- 3 - Fehlerkorrektur, β = Generator von GF(16) = $\mathbb{Z}_2[x]/_{x^4+x+1}$, $\beta = (0100)$ (siehe Adámek, Seite 322)
  $K = \{w \in \mathbb{Z}_2^{15} : w(\beta) = w(\beta^3) = w(\beta^5) = 0\}$

  minimale Polynome

  für β: $M_1(x) = x^4 + x + 1$
  für β^3: $M_2(x) = x^4 + x^3 + x^2 + x + 1$
  für β^5: $M_3(x) = x^2 + x + 1$

  $g(x) \in K \Leftrightarrow \underbrace{M_1(x) \cdot M_2(x) \cdot M_3(x)}_{\text{Grad } 10} \mid g(x)$

Bemerkungen:

- manchmal lohnt es sich für β einen Generator des Körpers zu wählen.

- BCH Codes sind zyklisch mit Generatorpolynom:
  kgV$(M_1(x), M_2(x), \ldots, M_{2t}(x))$, wobei $M_i(x)$ das Minimalpolynom von β^i ist.

  Parity-Check-Matrix: für BCH Codes

  $$H = \begin{bmatrix} 1 & \beta & \beta^2 & \beta^3 & \cdots & \beta^{n-1} \\ 1 & \beta^2 & \beta^4 & \beta^6 & \cdots & (\beta^2)^{n-1} \\ & & \vdots & & \vdots & \\ 1 & \beta^{2t} & & & \cdots & (\beta^{2t})^{n-1} \end{bmatrix}$$

  (linear abhängige Zeilen sollten eliminiert werden). Dann

  $$H \cdot w^T = \begin{bmatrix} w(\beta) \\ w(\beta^2) \\ \vdots \\ w(\beta^{2t}) \end{bmatrix}$$

- Falls $\beta \in F[x]/p(x)$ und Grad$(p(x)) = m$ ist, dann kann man β als Vektor der Länge m darstellen. In H kann man dann β^i durch eine Folge von m Zeichen aus F ersetzen.

1.10.6 Fehlerkorrigierende Eigenschaften von BCH Codes

Theorem 28 *t-Fehlerkorrigierende BCH Codes haben Gewicht $\geq 2t+1$ (mit Ausnahme des Nullvektors). Also korrigieren diese Codes tatsächlich bis zu t Fehler.*

Beweis:
Nehmen wir einen Codewort w mit höchstens $2t$ von 0 verschiedenen Koordinaten: $w_{i_1}, w_{i_2}, \ldots, w_{i_{2t}}$. Dann gilt $H \cdot w^T = \vec{0}^T$. Es folgt, daß

$$\underbrace{\begin{bmatrix} \beta^{i_1} & \beta^{i_2} & \cdots & \beta^{i_{2t}} \\ (\beta^{i_1})^2 & (\beta^{i_2})^2 & \cdots & (\beta^{i_{2t}})^2 \\ & & \vdots & \\ (\beta^{i_1})^{2t} & (\beta^{i_2})^{2t} & \cdots & (\beta^{i_{2t}})^{2t} \end{bmatrix}}_{=:A \text{ (eine } 2t\times 2t \text{ Matrix)}} \cdot \begin{bmatrix} w_{i_1} \\ w_{i_2} \\ \vdots \\ w_{i_{2t}} \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ \vdots \\ 0 \end{bmatrix}$$

Dann: $\det A = \beta^{i_1} \cdot \ldots \cdot \beta^{i_{2t}} \cdot \det \underbrace{\begin{bmatrix} 1 & 1 & \cdots & 1 \\ \beta^{i_1} & \beta^{i_2} & \ldots & \beta^{i_{2t}} \\ & & \vdots & \\ (\beta^{i_1})^{2t-1} & (\beta^{i_2})^{2t-1} & \cdots & (\beta^{i_{2t}})^{2t-1} \end{bmatrix}}_{=:B}$

Es ist $\det B = \prod_{p>q}(\beta^{i_p} - \beta^{i_q})$ (Satz aus der Algebra für so gennante Vandermonde-Matrizen) und somit sowohl $\det B \neq 0$ als auch $\det A \neq 0$, d.h. $A \cdot \begin{bmatrix} w_{i_1} \\ \vdots \\ w_{i_{2t}} \end{bmatrix}$ hat genau eine Lösung. Da der Nullvektor eine Lösung ist, folgt: $w_{i_1} = \ldots = w_{i_{2t}} = 0$. □

1.10.7 Fehlerkorrekturalgorithmus für BCH Codes

Es gibt meherere schnelle Algorithmen zur Fehlerkorrektur von BCH Codes. Alle sind mathematisch (etwas) kompliziert.
Wir betrachten t-korrigierende BCH Codes über einem Körper F.

- sei $v(x)$ Codewort
  $w(x)$ empfangene Nachricht
  $e(x) = w(x) - v(x)$ Fehlermuster mit p Fehler ($p \leq t$), $e_{i_1}, \ldots, e_{i_p} \neq 0$
- für den Algorithmus definieren wir:
  1. $a_k = \beta^{i_k}$ (error location number)
     $b_k = e_{i_k}$ (error evaluation number)
     (es reicht $a_1, \ldots, a_p$ und $b_1, \ldots, b_p$ zu finden!).
  2. $\sigma(x) = (1 - a_1 x)(1 - a_2 x) \cdots (1 - a_p x)$ (error-location-Polynom)
     $\omega(x) = \sum_{k=1}^{p} a_k \cdot b_k \frac{\sigma(x)}{(1 - a_k x)}$ (error-evaluation-Polynom)
     Fakt: es reicht $\sigma(x)$ und $\omega(x)$ zu finden.
     Beweis:
     - durch Nullstellen von $\sigma(x)$ erfahren wir die Elemente a_k
     - $b_j = -\omega(1/a_j) \cdot \sigma'(1/a_j)^{-1}$
  3. Syndrom:
     $s(x) = s_1 + s_2 x + \cdots + s_{2t} x^{2t}$ für $s_i = w(\beta^i)$
     (wobei β^i der Generator des Körpers ist).

Algorithmus:

1. $s(x)$ berechnen
2. - Euklidischer Algorithmus zur Berechnung des ggTs für x^{2t} und $s(x)$ einsetzen,
   Notation:
   $a_k(x) = k$-te erhaltene Rest;
   $a_0(x) = x^{2t}$, $a_1(x) = s(x)$, $a_{k-2}(x) = a_{k-1}(x) \cdot q_k(x) + a_k(x)$;
   $u_k(x) = u_{k-1} \cdot q_k(x) + u_{k-2}(x)$, $v_k(x) = v_{k-1} \cdot q_k(x) + v_{k-2}(x)$;
   $a_k(x) = (-1)^k (v_k(x) \cdot a_0(x) - u_k(x) \cdot a_1(x))$.
   - Berechnung unterbrechen, wenn
   Grad $a_k(x) < t$ und Grad $a_{k-1}(x) \geq t$
   Es läßt sich zeigen, daß dann
   $\sigma(x) = d \cdot u_k(x)$, $\omega(x) = (-1)^{k+1} \cdot d \cdot a_k(x)$, wobei $d = \frac{1}{u_k(0)}$
3. Korrektur von allen w_{i_k}, für die $\sigma(\beta^{-i_k})$=0 gilt. Der richtige korrekte Wert ist $w_{i_k} + b_k$.

Die Idee für den Algorithmus (Details im Adámek, Kapitel 13)

- es läßt sich zeigen, daß $\omega(x) \equiv \sigma(x) \cdot s(x) \bmod x^{2t}$
  (folgt aus Definition durch einfache Rechnungen).
- durch den euklidischen Algorithmus für x^{2t} und $s(x)$ erhalten wir
  $a_k(x) = (-1)^k \cdot (v_k(x) \cdot x^{2t} - u_k(x) \cdot s(x))$,

  also Polynome $\overline{\omega}(x)$ und $\overline{\sigma}(x)$, so daß

  $\overline{\omega}(x) = (-1)^{k+1} \cdot (-1)^k \cdot (v_k(x) \cdot x^{2t} - \overline{\sigma}(x) \cdot s(x))$,
  $\overline{\omega}(x) = \overline{\sigma}(x) \cdot s(x) + x^{2t} \cdot (...) = \overline{\sigma}(x) \cdot s(x) \bmod x^{2t}$

  also $\overline{\omega}(x)/\overline{\sigma}(x) = s(x) = \omega(x)/\sigma(x) \bmod x^{2t}$
  $\Rightarrow \quad \overline{\omega}(x) \cdot \sigma(x) = \omega(x) \cdot \overline{\sigma}(x)$

  $\sigma(x) = \prod (1 - a_i x)$, kein $(1 - a_i x)$ teilt $\omega(x)$
  $\Rightarrow \quad (1 - a_i x)$ teilt $\overline{\sigma}(x)$
  $\Rightarrow \quad \sigma(x)$ teilt $\overline{\sigma}(x)$
  $\Rightarrow \quad \overline{\sigma}(x) = b(x) \cdot \sigma(x)$ und $\overline{\omega}(x) = b(x) \cdot \omega(x)$.
- Es reicht zu zeigen, daß Grad$(b(x)) = 1$.
  Da $u_k(x)$ und $v_k(x)$ teilerfremd sind (Eigenschaft des euklidischen Algorithmus!), reicht es zu zeigen, daß $b(x) | \overline{\sigma}(x)$ (klar!) und $b(x) | v_k(x)$.
  Berechnung:
  $x^{2t} \cdot v_k(x) = \overline{\sigma}(x) \cdot s(x) - \overline{\omega}(x) = \overline{\sigma}(x) \cdot s(x) - b(x) \cdot \omega(x) =$
  $\overline{\sigma}(x) \cdot s(x) - b(x) \cdot (\sigma(x) \cdot s(x) + x^{2t} \cdot c(x)) = -b(x) \cdot x^{2t} \cdot c(x)$
  $\Rightarrow \quad v_k(x) = -b(x) \cdot c(x)$

Beispiel für den allgemeinen Fall

Sei w = 00120000 die empfangene Nachricht, wobei ein BCH-Code der Länge 8 über $\mathbb{Z}_3$, der zwei Fehler korrigieren kann, benutzt wurde.

Für GF(9) $\equiv \mathbb{Z}_3[x]/x^2 + x + 2$ und $\alpha = x$ als generierendes Element ergibt sich die folgende Tabelle:

0	1	α	$1+2\alpha$	$2+2\alpha$	2	2α	$2+\alpha$	$1+\alpha$
—	α^0	α	α^2	α^3	α^4	α^5	α^6	α^7

Schritt I. Mit dem empfangenen Wort w wird das Polynom $w(x) = x^2 + 2x^3$ assoziiert. Da der BCH-Code die Länge 8 hat, kann man als β das generierende Element α wählen. Die Koeffizienten des Syndrompolynoms $s(x)$ ergeben sich durch:

$$
\begin{aligned}
s_1 &:= w(\beta^1) = w(\alpha^1) = \alpha^2 + 2\alpha^3 = (1+2\alpha) + 2(2+2\alpha) = 5 + 6\alpha = 2 = \alpha^4 \\
s_2 &:= w(\beta^2) = w(\alpha^2) = \alpha^4 + 2\alpha^6 = (2) + 2(2+\alpha) \quad = 6 + 2\alpha \quad = \alpha^5 \\
s_3 &:= w(\beta^3) = w(\alpha^3) = \alpha^6 + 2\alpha^1 = (2+\alpha) + 2\alpha \quad = 2 + 3\alpha = 2 = \alpha^4 \\
s_4 &:= w(\beta^4) = w(\alpha^4) = \alpha^8 + 2\alpha^4 = 1 + 2(2) \quad = 5 \quad = \alpha^4
\end{aligned}
$$

und damit das Syndrompolynom durch $s(x) := \alpha^4 x^3 + \alpha^4 x^2 + \alpha^5 x + \alpha^4$.

Schritt II. Der euklidische Algorithmus liefert:

$$
\begin{array}{lll}
a_0(x) := x^{2t} = x^4; & & u_0(x) := 0 \\
a_1(x) := s(x); & & u_1(x) := 1 \\
a_2(x) = \alpha^2 x^2 + \alpha^6 x + 1; & q_1(x) = \alpha^4 x + \alpha^4; & u_2(x) = q_1(x) \\
a_3(x) = \alpha^5; & q_2(x) = \alpha^2 x + \alpha^6; & u_3(x) = q_1(x) q_2(x) + 1, \text{ denn}
\end{array}
$$

$$
\begin{array}{rl}
a_0(x) \text{ div } a_1(x) = x^4 \text{ div } (\alpha^4 x^3 + \alpha^4 x^2 + \alpha^5 x + \alpha^4) = 2x + 1 = q_1(x) & \\
\underline{x^4 + \ x^3 + \ \alpha x^2 + \quad x} & \\
\quad 2x^3 + \alpha^5 x^2 + \ 2x & \\
\underline{2x^3 + \ 2x^2 + 2\alpha x + 2} & \\
\qquad \alpha^2 x^2 + \alpha^6 x + 1 & = a_2(x)
\end{array}
$$

$$
\begin{array}{rl}
a_1(x) \text{ div } a_2(x) = (\alpha^4 x^3 + \alpha^4 x^2 + \alpha^5 x + \alpha^4) \text{ div } (\alpha^2 x^2 + \alpha^6 x + 1) = \alpha^2 x + \alpha^6 = q_2(x) & \\
\underline{\alpha^4 x^3 + \alpha^8 x^2 + \alpha^2 x} & \\
\qquad x^2 + \ 2x + \alpha^4 & \\
\qquad \underline{x^2 + \ 2x + \alpha^6} & \\
\qquad\qquad \alpha^5 & = a_3(x)
\end{array}
$$

Schritt III. Da der Grad von $a_k(x) = a_3(x)$ kleiner als $t = 2$ ist, wird definiert:

$$
\begin{aligned}
k \quad &:= \quad 3; \\
u_k(x) \quad &= \quad u_3(x) = q_1(x) q_2(x) + 1 = (2x+1)(\alpha^2 x + \alpha^6) + 1 \\
&= \quad 2\alpha^2 x^2 + (2\alpha^6 + \alpha^2) x + \alpha^6 + 1 = \alpha^6 x^2 + \alpha^6 x + \alpha; \\
d \quad &:= \quad [u_k(0)]^{-1} = \alpha^{-1}; \\
\sigma(x) \quad &:= \quad d \cdot u_k(x) = \alpha^5 x^2 + \alpha^5 x + 1; \\
\omega(x) \quad &:= \quad (-1)^{k+1} \cdot d \cdot a_k(x) = 2; \\
\sigma'(x) \quad &= \quad 2\alpha^5 x + \alpha^5.
\end{aligned}
$$

Schritt IV. Bestimmung der Nullstellen von $\sigma(x)$:

$$
\begin{aligned}
\sigma(\beta^0) = \sigma(\alpha^0) &= \alpha^5 + \alpha^5 + 1 = 2\alpha + 2\alpha + 1 \quad = \alpha + 1 \quad \neq 0 \\
\sigma(\beta^1) = \sigma(\alpha^1) &= \alpha^7 + \alpha^6 + 1 = (1+\alpha) + (2+\alpha) + 1 \quad = 2\alpha + 4 \neq 0 \\
\sigma(\beta^2) = \sigma(\alpha^2) &= \alpha^1 + \alpha^7 + 1 = \alpha + (1+\alpha) + 1 \quad = 2\alpha + 2 \neq 0 \\
\sigma(\beta^3) = \sigma(\alpha^3) &= \alpha^3 + \alpha^8 + 1 = (2+2\alpha) + 1 + 1 \quad = 2\alpha + 4 \neq 0 \\
\sigma(\beta^4) = \sigma(\alpha^4) &= \alpha^5 + \alpha^1 + 1 = 2\alpha + \alpha + 1 \quad = 1 \quad \neq 0 \\
\sigma(\beta^5) = \sigma(\alpha^5) &= \alpha^7 + \alpha^2 + 1 = (1+\alpha) + (1+2\alpha) + 1 = 3\alpha + 3 = 0 \\
\sigma(\beta^6) = \sigma(\alpha^6) &= \alpha^1 + \alpha^3 + 1 = \alpha + (2+2\alpha) + 1 \quad = 3\alpha + 3 = 0 \\
\sigma(\beta^7) = \sigma(\alpha^7) &= \alpha^3 + \alpha^4 + 1 = \alpha^3 + 2 + 1 \quad = \alpha^3 \quad \neq 0
\end{aligned}
$$

Da $\sigma(\beta^{-2}) = \sigma(\beta^{-3}) = 0$ ist, müssen die Zeichen an den Positionen 2 und 3 korrigiert werden und zwar durch:

$$
\begin{aligned}
v_2 &:= w_2 + \omega(\beta^{-2}) \left[\sigma'(\beta^{-2})\right]^{-1} = 1 + \omega(\alpha^6) \left[\sigma'(\alpha^6)\right]^{-1} = 1 + 2 \left[2\alpha^{11} + \alpha^5\right]^{-1} = \\
&\quad 1 + 2 \left[2(2+2\alpha) + 2\alpha\right]^{-1} = 1 + 2[4]^{-1} = 0
\end{aligned}
$$

$$v_3 := w_3 + \omega(\beta^{-3}) \left[\sigma'(\beta^{-3})\right]^{-1} = 2 + \omega(\alpha^5) \left[\sigma'(\alpha^5)\right]^{-1} = 2 + 2\left[2\alpha^{10} + \alpha^5\right]^{-1} = 2 + 2\left[2(1+2\alpha) + 2\alpha\right]^{-1} = 2 + 2[2]^{-1} = 0$$

Damit ist die empfangene Nachricht zum Codewort $v = 00000000$ korrigiert worden.

1.10.8 Reed-Solomon Codes

Definition

Sei F ein Körper, $|F| = n + 1$. Wir betrachten zyklische Kodierung der Länge $= n$ mit Codealphabet F.
Das Generatorpolynom des zyklischen Codes ist $g(x) = (x-\beta)(x-\beta^2)\cdots(x-\beta^{2t})$, wobei β ein Generator von F ist.
Eine solcher Code heißt t-fehlerkorrigierender **Reed-Solomon Code**.

Eigenschaften

- Reed-Solomon Codes sind BCH Codes.

  Beweis: $w(\beta^i) = 0 \Leftrightarrow (x - \beta^i)|w(x)$. Man Kann das Polynom $x - \beta^i$ verwenden, weil $\beta \in F$ ist und die Polynome Koeffiziente aus F haben!! Also $w(\beta) = w(\beta^2) = \ldots w(\beta^{2t}) = 0 \Leftrightarrow (x-\beta)\cdot(x-\beta^2)\cdot\ldots\cdot(x-\beta^{2t})|w(x)$. □

- Folgender **verallgemeinerter Reed-Solomon Code** läßt sich davon ableiten: Jedes Element aus $F = \mathrm{GF}(2^m)$ wird als Folge von m Bit dargestellt. Neue Codelänge $= n \cdot m = (2^m - 1) \cdot m$.

  Eingenschaft: Büschelstörungen der Länge $(t-1)m+1$ lassen sich korrigieren.

  Beweis: $(t-1)m+1$ benachbarte Bits beeinflußen höchstens t Symbole des ursprünglichen Reed-Solomon Codewortes. □

1.11 Goppa Codes

Definition von $1/(x-a)$

Gegeben sei ein Polynom $r(x)$ und ein a, mit $r(a) \neq 0$. Dann:

$$\frac{1}{x-a} \stackrel{\text{def}}{=} \frac{r(x)-r(a)}{x-a} \cdot \frac{1}{r(a)}$$

Es teilt $x - a$ das Polynom $r(x) - r(a)$, da $r(x) - r(a)$ für $x = a$ den Wert 0 hat. Also ist $\frac{r(x)-r(a)}{x-a}$ ein echtes Polynom.

Definition: Goppa Codes

Sei

- F ein Körper
- $r(x)$ = Polynom über eine Erweiterung von F mit $\mathrm{Grad}(r(x)) \geq 2$
- $a_1, \ldots, a_n : r(a_i) \neq 0$ (a_i aus dem Erweiterungskörper)

Dann ist $K := \{(v_1 \cdots v_n) \in F^n : \sum_{i=1}^n \frac{v_i}{x-a_i} = 0\}$ ein **Goppa-Code**.
Im weiteren sei $t := \mathrm{Grad}(r(x))$.

1.11.1 Parity-Check-Matrix für Goppa-Codes

Durch einfache Berechnung erhält man[3]:

$(v_1 \cdots v_n) \in K \Leftrightarrow \sum_{i=1}^{n} v_i \cdot \frac{a_i^s}{r(a_i)} = 0$ für $s = 0, 1, \ldots, t-1$.

Parity-Check-Matrix

$$H = \begin{bmatrix} \frac{1}{r(a_1)} & \frac{1}{r(a_2)} & \cdots & \frac{1}{r(a_n)} \\ \frac{a_1}{r(a_1)} & \frac{a_2}{r(a_2)} & \cdots & \\ \frac{a_1^2}{r(a_1)} & \frac{a_2^2}{r(a_2)} & \cdots & \\ \vdots & \vdots & \ddots & \\ \frac{a_1^{t-1}}{r(a_1)} & \frac{a_2^{t-1}}{r(a_2)} & \cdots & \frac{a_n^{t-1}}{r(a_n)} \end{bmatrix}$$

Fehlerkorrektur: Jede Quadratische Teilmatrix von H entspricht einer Vandermonde-Matrix. Wie bei BCH-Codes zeigt man, daß deswegen das minimale Gewicht mindestens $t+1$ sein muß. Das minimale Gewicht kann aber größer gemacht werden:

1.11.2 Irreduzible Goppa Codes

Definition:

Sei $r(x)$ ein Primpolynom über $F' := \{a_1, \ldots, a_n\}$. Dann heißt der durch die Parameter $r(x), a_1, \ldots, a_n$ und F' definierte Goppa-Code **irreduzibel**.

Theorem 29 *Irreduzible* binäre *Goppa-Codes, die von einem Polynom $r(x)$ bestimmt werden, können bis zu $t := Grad(r(x))$ Fehler korrigieren.*

Beweis:
Wir zeigen, daß für jedes Codewort $v \neq 0$ gilt: Gewicht$(v) \geq 2t+1$.
Sei $v_{i_1}, \ldots, v_{i_q} = 1$, sonst $v_i = 0$. Dann:

- $\sum_{j=1}^{q} \frac{1}{x - a_{i_j}} = 0$, weil v ein Codewort ist.
- Sei $f(x) = (x - a_{i_1})(x - a_{i_2}) \ldots (x - a_{i_q})$, dann sind das Primpolynom $r(x)$ und das Polynom $f(x)$ teilerfremd. (Hätten $r(x)$ und $f(x)$ einen gemeinsamen Teiler, dann muß es ein Vielfaches von einem $x - a_{i_j}$ sein. Dann wäre a_{i_j} auch eine Nullstelle von $r(x)$!!)
- $f'(x) := \sum_{s=1}^{q} \frac{f(x)}{x - a_{i_s}}$ sei die formale Ableitung von f.
- Sei χ der Restpolynom x aus $\mathbb{Z}_2[x]/_{r(x)}$. Dann folgt aus $r(\chi) = 0$, daß $f(\chi) \neq 0$. (Eine gemeinsame Nullstelle $\Rightarrow$ ein gemeinsamer Faktor!)
  Also existiert ein $g(x) \in \mathbb{Z}_2[x]/_{r(x)}$ mit $f(\chi) \cdot g(\chi) = 1$ in $\mathbb{Z}_2[x]/_{r(x)}$.
- $f'(\chi) \cdot g(\chi) = \sum_{s=1}^{q} \frac{f(\chi) \cdot g(\chi)}{\chi - a_{i_j}} = \sum \frac{1}{\chi - a_{i_j}} = 0$
  Die letzte Gleichung muß begründet werden:

$$\begin{aligned} \frac{1}{x - a_{i_j}}(\chi) &= \frac{r(x) - r(a_{i_j})}{x - a_{i_j}} \cdot \frac{1}{r(a_{i_j})}(\chi) = \frac{r(\chi) - r(a_{i_j})}{\chi - a_{i_j}} \cdot \frac{1}{r(a_{i_j})} \\ &= \frac{0 - r(a_{i_j})}{\chi - a_{i_j}} \cdot \frac{1}{r(a_{i_j})} = \frac{-1}{\chi - a_{i_j}}. \end{aligned}$$

  Also $\sum \frac{-1}{\chi - a_{i_j}}$ ist der Wert von $\sum \frac{1}{x - a_{i_j}}$ an der Stelle χ und somit 0.

[3] Für BCH Codes, war es analog: $(v_1 \cdots v_n) \in K \Leftrightarrow \sum_{i=1}^{n} v_i \cdot \beta^{is} = 0$ für $s = 1, \ldots, 2t$.

- $\Rightarrow r(x)|f'(x) \cdot g(x)$ (da $r(x)$ das minimale Polynom von χ ist)
  $r(x)$ teilt $g(x)$ nicht, weil $\text{Grad}(g(x)) < \text{Grad}(r(x))$ ist $\Rightarrow r(x)|f'(x)$.
  $(\Rightarrow t \leq \text{Grad}(f') = q-1)$

  Lemma 30 *In $GF(2^m)$ existiert für alle $h(x)$ ein $q(x)$ mit $h'(x) = q^2(x)$.*

  Beweis: Da 2=0 in $\text{GF}(2^m)$ ist, erhalten wir:

  - $(x^n)' = n \cdot x^{n-1} \Rightarrow h'(x)$ enthält keine x^i für ungerade i.
  - $\forall a \in \text{GF}(2^m) : \exists b \in \text{GF}(2^m) : b^2 = a$ bzw. $b = \sqrt{a}$,
    denn für α (Generator von $\text{GF}(2^m)$) gilt: $\alpha^{2^m} = \alpha$, also $\sqrt{\alpha} = \alpha^{2^{m-1}}$,
    und für andere Elemente gilt: $\sqrt{\alpha^i} = (\sqrt{\alpha})^i$
  - Sei $f'(x) = \sum_{i=0} b_{2i} x^{2i}$. Definiere $q(x) := \sum_{i=0} \sqrt{b_{2i}} x^i$. Dann gilt offensichtlich $q^2(x) = f'(x)$, da für alle $a, b \in \text{GF}(2^m)$ gilt, daß $(a+b)^2 = a^2 + b^2$ ist. □ (Lemma)

- $r(x)|f'(x) =: q^2(x) \Rightarrow r(x)|q(x) \Rightarrow r^2(x)|q^2(x) = f'(x)$
  $\Rightarrow 2 \cdot \text{Grad}(r(x)) \leq \text{Grad}(f'(x)) = q-1$
  Also: $2t \leq q-1$. D.h. $\text{Gewicht}(v) \geq 2t+1$. □

1.11.3 Eigenschaften von Goppa Codes

- Goppa Codes sind für gewisse Parameter BCH Codes.
- Goppa Codes sind nicht immer zyklisch.
- da Goppa Codes viele Möglichkeiten für die Parameterauswahl bieten, sind sie flexibel.

1.12 Büschelstörungen–Interleaving

Eine einfache Technik, die viele praktische Anwendungen hat (z.B. Codes für CD):

Beispiel:
Seien $C = c_1c_2 \ldots c_k$, $D = d_1d_2 \ldots d_k$, $E = e_1e_2 \ldots e_k$, $F = f_1f_2 \ldots f_k$ vier Codewörter. Man kann diese Wörter mischen:

$$c_1d_1e_1f_1c_2d_2e_2f_2 \ldots c_kd_ke_kf_k$$

Falls man mit der ursprünglichen Codierung t Fehler korrigieren kann, dann kann man nach dem Mischen Büschelstörungen der Länge $4t$ korrigieren.
Begründung: Ein Block von $4t$ fehlerhaften Bits enthält t Bits c_j, t Bits d_j, t Bits e_j und t Bits f_j. Die Unterfolgen C, D, E, F können also separat korrigiert werden.

Diese Technik heißt **interleaving**.

1.13 Convolutional Codes

Noch eine oft benutzte Technik:

Input: Polynom $u(x)$ (möglicherweise unendliche Folge von Symbolen).
Output: Polynom $C(u(x))$, wobei die Funktion C folgende Eigenschaften hat:

1. C ist linear:
   $C(u(x) + v(x)) = C(u(x)) + C(v(x))$
   $C(t \cdot u(x)) = t \cdot C(u(x))$

2. C ist zeitinvariant:
   (Verzögerung von k beim Input verursacht eine Verzögerung von n beim Output)
   $C(x^k \cdot u(x)) = x^n \cdot C(u(x))$

siehe auch Abbildung 1.7 und 1.9.

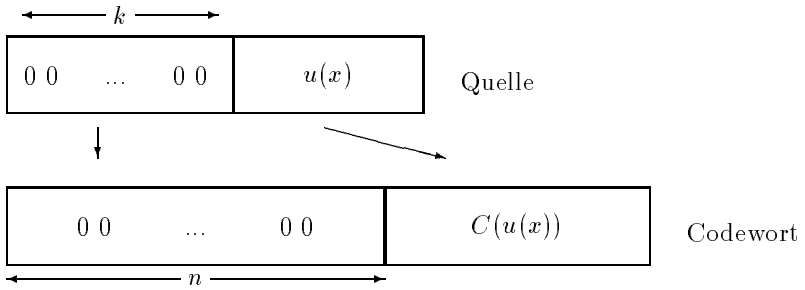


Abbildung 1.7: Zeitinvarianz von Convolutional Codes

Um einen linearen Block-Code zu bekommen braucht man noch eine Eigenschaft: Die ersten k Input-Symbole beeinflußen nur die ersten n Output-Symbole.

Beispiel:

$k = 1, n = 2, C(1) = 111101 (= 1 + x + x^2 + x^3 + x^5)$
$C(1) = 1111010...$
$C(01) = 001111010...$ [bzw. $C(0 + x)$]
Die convolutional Codes können einfach durch Schaltkreise mit Shiftregistern berechnet werden. Für den obigen Code siehe Abbildung 1.8.

Definition: (n, k)-Convolutional Codes

(n, k)-Convolution Code:
Eine Verzögerung von k Bits beim Input
$\Rightarrow$ eine Verzögerung von n Bits beim Output.
(siehe Abbildung 1.9)
Beispiel: binäre $(n, 1)$-Code
Sei $C(1) = g_0(x)$. Dann ist
$C(0) = C(0 \cdot 1) = 0 \cdot C(1) = 0$
$C(x) = C(x \cdot 1) = x^n g_0(x)$
$C(x^2) = C(x \cdot x \cdot 1) = x^n \cdot C(x \cdot 1) = x^{2n} g_0(x)$
Somit gilt:

- $C(x^i) = x^{in} \cdot g_0(x)$
- $u = \sum_{i=0} u_i \cdot x^i \rightsquigarrow C(u) = \sum_{i=0} u_i \cdot C(x^i) = \sum_{i=0} u_i \cdot x^{in} \cdot g_0(x) = g_0(x) \cdot (\sum_{i=0} u_i \cdot x^{in}) = u(x^n) \cdot g_0(x)$
- Ein Schaltkreis für C:
  Sei $g_0(x) = (a_0 + a_1 x + \cdots + a_{n-1} x^{n-1}) + (a_n x^n + \cdots + a_{2n-1} x^{2n-1}) + \ldots + (a_{mn} x^{mn} + \cdots + a_{(m+1)n-1} x^{(m+1)n-1})$.

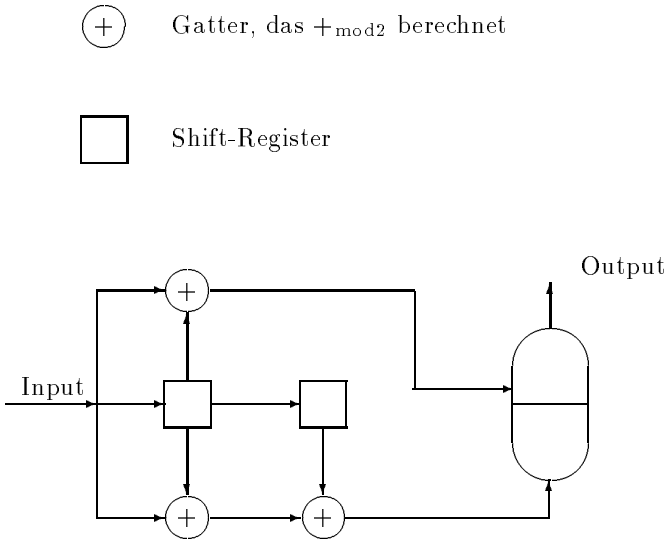
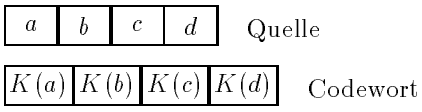


Abbildung 1.8: ein Schaltkreis, der convolutional Codes berechnet

Situation bei Block-Codes:



Situation bei Convolutional Codes:

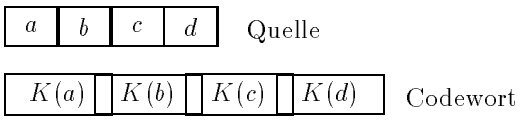


Abbildung 1.9: Convolutional vs. Block-Codes

Dabei entspricht A_i dem Polynom $a_{in}x^{in} + a_{in+1}x^{in+1} + \cdots + a_{in+n-1}x^{in+n-1}$.

Die Symbole aus Input und Shiftregister aktivieren bzw. deaktivieren die Teile A_i. Symbol 0 deaktiviert A_i, so daß A_i Nullen an alle Output-Gatter schickt. Symbol 1 als Eingabe für A_i verursacht, daß A_i den Wert $a_{n\cdot i+j}$ an das Output-Gatter j schickt.

Binäre (n,k)-Convolutional Codes

- Die Polynome $g_i(x) = C(x^i)$ für $i = 0, \ldots, k-1$ bestimmen C, z.B.
  $C(x^k) = C(1 \cdot x^k) = x^n \cdot C(1)$
  $C(x^{k+1}) = C(x \cdot x^k) = x^n \cdot C(x)$
- $C(u(x)) = \sum_{i=0}^{k-1} u^{(i)}(x^n) \cdot g_i(x)$,
  wobei für $u = u_0u_1u_2\ldots$ der i-te Shift $u^{(i)} := u_iu_{i+k}u_{i+2k}\ldots$ ist.

Bemerkung: Gute Convolutional Codes (bezüglich der Fehlerkorrekturmöglichkeiten) wurden durch Rechner gefunden. Eine genaue Analyse fehlt.

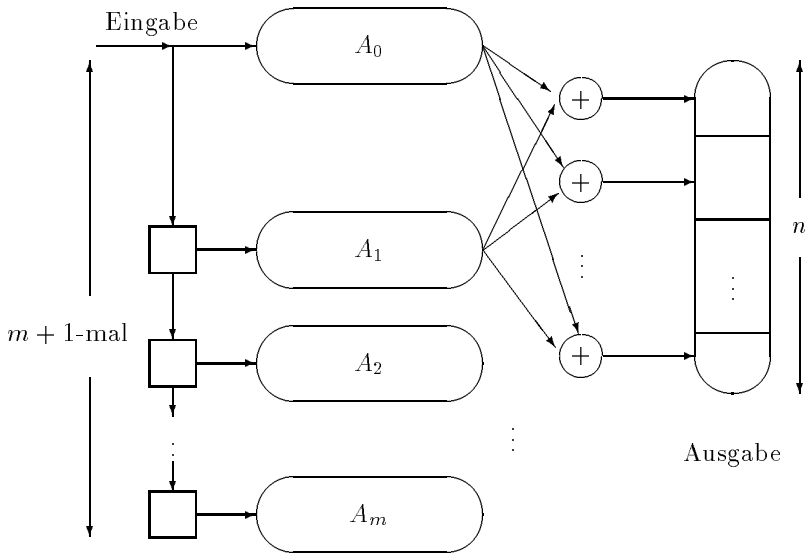


Abbildung 1.10: Beispiel: $(n, 1)$ Convolutional Code

1.13.1 Trellis-Diagramm

Convolutional Code $\leftrightarrow$ Schaltkreis
Fehlerkorrektur durch Analyse des Schaltkreises

Definition: Trellis-Diagramm

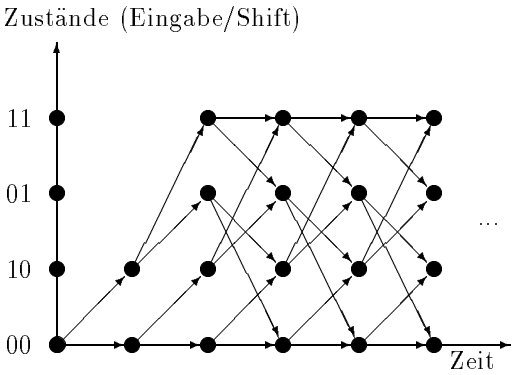


Abbildung 1.11: Trellis-Diagramm für 2 Shiftregister

Zustand des Schaltkreises = Inhalt aller Shiftregister (als Folge der Länge m, wobei m = Anzahl der Shiftregister ist).
Trellis-Diagramm: siehe Abbildung 1.11

- Knoten $\leftrightarrow$ Tupel: (Zustand der Shiftregister, Zeitpunkt)

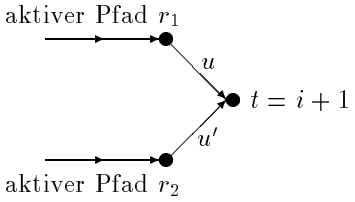


Abbildung 1.12: Bestimmung eines neuen aktiven Pfades

- gerichtete Kanten:
  ij (im Moment t) $\rightarrow si$ (im Moment $t+1$) [bei Eingabebit s]
  allgemein (bei m Shiftregister):
  $i_1 i_2 \cdots i_m \rightarrow s i_1 \cdots i_{m-1}$ [bei Eingabebit s]
- die Kanten haben Etiketten:
  $ij \xrightarrow{\alpha} si$
  Folge α = Ausgabefolge, die im Zustand ij bei Eingabe s erzeugt wird.

1.13.2 Viterbi Algorithmus zur Fehlerkorrektur

empfangene Nachricht $= s_1 s_2 s_3 \cdots$
Für jeden Knoten A des Trellis - Diagramms gilt: ein Pfad vom Anfangsknoten bis A ist **aktiv**, falls die Etiketten entlang des Pfades ein Wort w bilden, so daß die Hamming-Distanz von w und s minimal ist (für alle Pfade vom Anfangsknoten zu den 2^m Knoten, die dem selben Zeitpunkt entsprechen).

Schritt 0
Alle Knoten des Zeitpunkts werden mit ihrer Hamming-Distanz zum bis dahin empfangenen Wort markiert ($\Rightarrow d_H(\epsilon, \epsilon) = 0$). Bestimme die aktiven Pfade vom Anfangsknoten bis zum Anfangsknoten ($\Rightarrow$ der einzige aktive Pfad ist der leere Pfad).

Schritt $i \rightarrow i+1$
Zu jedem Knoten des Zeitpunkts $i+1$ existieren höchstens zwei eingehende Kanten. Von den beiden Vorgängerknoten (die den aktiven Pfaden r_1 und r_2 entsprechen mögen) sind die aktiven Pfade und deren Hamming-Distanz zur bis dahin empfangenen Nachricht bekannt. Hieraus lassen sich der/die aktiven Pfade des Knotens zum Zeitpunkt $i+1$ berechnen:
$d_H(r_1, s_1 s_2 \cdots s_i)$, $d_H(r_2, s_1 s_2 \cdots s_i)$ sind bekannt. Hieraus berechnen wir:

$$d_H(r_1 u, s_1 s_2 \cdots s_i s_{i+1}) = d_H(r_1, s_1 s_2 \cdots s_i) + \begin{cases} 0 & \text{für } u = s_{i+1} \\ 1 & \text{sonst} \end{cases}$$

und analog $d_H(r_2 u', s_1 s_2 \cdots s_i s_{i+1})$.
Ist $d_H(r_1 u, s_1 \cdots s_{i+1}) = d_H(r_2 u', s_1 \cdots s_{i+1})$, so ist eine eindeutige Fehlerkorrektur u.U. später nicht möglich! Auf jeden Fall wird der (die) Pfad(e) mit der geringeren Distanz als aktiver Pfad zu dem neuen Knoten bestimmt (was auch der Definition von aktivem Pfad entspricht).

Diese Iteration wird unterbrochen falls alle aktiven Pfade denselben Präfix $R \neq 0$ haben. Ist $v_0 \cdots v_{n-1}$ das Etikett von Pfad R, so:

- korrigiere die ersten n Zeichen aus $s_1 s_2 \cdots$ in $v_0 \cdots v_{n-1}$ und
- beginne den Algorithmus von neuem mit $s_{n+1} s_{n+2} \cdots$ als empfangene Nachricht und mit dem Startzustand, der am Ende von R erreicht wurde. Hat man früher die aktiven Pfade bis zum Zeitpunkt t berechnet, dann sind diese Pfade auch für das neue Problem gültig. Man muß lediglich für jeden Pfad den Präfix R entfernen.

Kapitel 2

Data compression

2.1 Codes mit variabler Länge

Beispiel: Morse Code,
Die Buchstaben, die oft vorkommen, haben kurze Codewörter. Die Buchstaben, die selten vorkommen, haben lange Codewörter. Dadurch wird die *durchschnittliche* Länge des Codewortes kleiner.

2.1.1 Eindeutige Codes

Definition:

Eine Codierung $K : \Sigma \to \Gamma^*$ heißt **eindeutig** falls für alle $x_1 x_2 \dots x_L \in \Sigma^L$ und $y_1 y_2 \dots y_S \in \Sigma^S$ gilt:

$$x_1 x_2 \dots x_L \neq y_1 y_2 \dots y_S \quad \Rightarrow \quad K(x_1 x_2 \dots x_L) \neq K(y_1 y_2 \dots y_S).$$

Beispiele

Sei $\Sigma = \{a, b, c, d\}$ und $\Gamma = \{0, 1\}$.

- $K(a) = 00, K(b) = 10, K(c) = 101, K(d) = 110$.
  Für K ist keine eindeutige Decodierung möglich, weil
  $K(c)K(b) = 10110 = K(b)K(d)$.
- $K(a) = 0, K(b) = 01, K(c) = 011, K(d) = 111$
  Die Codierung K ist eindeutig, weil
  - jede 0 einen zu decodierenden Teil abgrenzt und
  - jeder String 01^i (mit i maximal) eindeutig decodiert werden kann.

  Beispiel: Für das Codewort $C = 011111111$ müßen die letzten drei Einsen $K(d)$ bilden, also $C = 011111K(d)$. Die nächste drei Einsen müssen auch $K(d)$ bilden, $\cdots$. Letztendlich bekommen wir $C = K(cdd)$.

  Problem: Wir müßen u.U. den ganzen Codestring kennen, bevor wir das erste Zeichen decodieren können!

2.1.2 Unmittelbare Codierung

Definition:

Eine Codierung $K : \Sigma \to \Gamma^*$ heißt **unmittelbar** falls:

$$\forall a, b \in \Sigma; a \neq b : K(a) \text{ ist kein Präfix von } K(b).$$

Decodierung von unmittelbaren Codes

1. lese Codesymbole bis sie mit einem $K(a)$ übereinstimmen
2. output a
3. go to 1

Decodierungsbaum

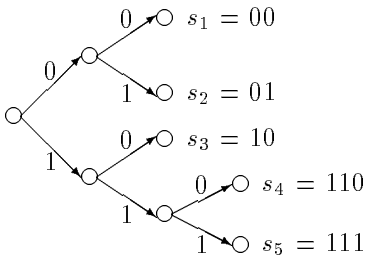


Abbildung 2.1: Decodierungsbaum

Für die Decodierung ist (bei Schritt 1 des Algorithmus) ein so genannter Decodierungsbaum hilfreich. Die Abbildung 2.1 zeigt den Decodierungsbaum für die Codeworte $s_1 = 00, s_2 = 01, s_3 = 10$ und $s_4 = 110, s_5 = 111$, die den Blättern entsprechen. Die Etiketten auf dem Pfad von der Wurzel zu einem Blatt s_i bilden das Codewort für s_i.
Es ist sinnvoll die Codeworte so zu bestimmen, daß die inneren Knoten des Decodierungsbaumes möglichst viele Kinder haben (im Falle von $\Gamma = \{0, 1\}$ also zwei Söhne).

2.1.3 Erzeugung einer unmittelbaren Codierung

Algorithmus

Eingabe: $a_1, \ldots, a_n \in \Sigma$, sowie die gewünschte Länge der Codewörter: $d_1, \ldots, d_n$ (ohne Einschränkung $d_1 \leq d_2 \leq \ldots \leq d_n$).

1. Wähle einen beliebigen String der Länge d_1 als $K(a_1)$.
2. Wähle einen beliebigen String der Länge d_2, so daß $K(a_1)$ kein Präfix von $K(a_2)$ ist. Dies ist möglich, falls die Anzahl an Strings der Länge d_2 größer der Anzahl an Strings der Länge d_2 mit Präfix $K(a_1)$ ist, d.h. $2^{d_2} > 2^{d_2 - d_1}$ bzw. $2^{d_2} \geq 2^{d_2 - d_1} + 1$ und somit
$$1 \geq 2^{-d_1} + 2^{-d_2}.$$
3. wähle einen beliebigen String der Länge d_3, so daß $K(a_1)$ und $K(a_2)$ keine Präfices von $K(a_3)$ sind. Dies ist möglich, falls $2^{d_3} \geq 2^{d_3 - d_1} + 2^{d_3 - d_2} + 1$, oder:
$$1 \geq 2^{-d_1} + 2^{-d_2} + 2^{-d_3}.$$
4. ...

Die Prozedur läuft problemlos bis zum Ende, wenn:

$$1 \geq 2^{-d_1} + 2^{-d_2} + \ldots + 2^{-d_n}$$

(das ist die so genannte *Kraftsche Ungleichung*)

Theorem 31 *Gegeben: Quellenalphabet* $\Sigma = \{a_1, \ldots, a_n\}$, *Codealphabet* Γ, $|\Gamma| = k$, *so daß*

$$\sum_{i=1}^{n} \frac{1}{k^{d_i}} \leq 1.$$

Dann gibt es eine unmittelbare Codierung $K : \Sigma \to \Gamma^*$, *so daß für alle* $i \in \{1, \ldots, n\}$ *gilt:* $|K(a_i)| = d_i$.

Beweis: durch die obige Konstruktion. □

Theorem 32 [McMillans Satz] *Sei* $K : \{a_1, \ldots, a_n\} \to \Gamma^*$ *mit* $|\Gamma| = k$ *eine eindeutige Codierung und* $d_i := |K(a_i)| \neq 0$ *für* $i = 1, \ldots, n$. *Dann gilt*

$$\sum_{i=1}^{n} \frac{1}{k^{d_i}} \leq 1.$$

Beweis: Sei $c := \sum_{i=1}^{n} \frac{1}{k^{d_i}}$. Es reicht zu zeigen, daß $\lim_{\substack{r\to\infty \\ r\in\mathbb{N}}} \frac{c^r}{r} \leq M \in \mathbb{R}$. (weil: $c > 1 \implies \lim_{r\to\infty} \frac{c^r}{r} = +\infty$). Es gilt:

$$c^2 = (\sum_{i=1}^{n} \frac{1}{k^{d_i}}) \cdot (\sum_{j=1}^{n} \frac{1}{k^{d_j}}) = \sum_{i,j=1}^{n} \frac{1}{k^{d_i+d_j}}$$

$$c^r = \sum_{i_1,\ldots,i_r=1}^{n} \frac{1}{k^{d_{i_1}+\ldots+d_{i_r}}}$$

Da wir eine eindeutige Codierung vorausgesetzt haben, existiert für jedes Wort $w \in \Gamma^j$ höchstens ein Quellstring $a_{i_1} \cdots a_{i_s} \in \Sigma^s$ mit $w = K(a_{i_1}) \cdots K(a_{i_s})$. Es gilt außerdem $j = |w| = |K(a_{i_1}) \cdots K(a_{i_s})| = |K(a_{i_1})| + \ldots + |K(a_{i_s})| = d_{i_1} + \ldots + d_{i_s}$. Also existieren für jedes j höchstens k^j Summanden der Form $j = d_{i_1} + \cdots + d_{i_s}$.
Sei $d := \max\{d_1, \ldots, d_n\}$. Damit ist:

$$c^r = \sum_{i_1,\ldots,i_r=1}^{n} \frac{1}{k^{d_{i_1}+\ldots+d_{i_r}}} \leq \sum_{j=r}^{d\cdot r} \frac{k^j}{k^j} \leq dr, \text{ also } \frac{c^r}{r} \leq d =: M.$$

□

Korollar 33 *Es existiert eine* eindeutige *Codierung mit Codelängen* $d_1, \ldots, d_k$
$\Updownarrow$
es existiert eine unmittelbare *Codierung mit Codelängen* $d_1, \ldots, d_k$.

2.2 Huffman-Codierungen

Problem: Wie kann man Codierungen entwickeln, bei denen im Mittel die Codewörter kurz sind?

- Beschränkung: Kraftsche Ungleichung.
- Die am häufigsten auftretenden Symbole sollten kurze Codeworte besitzen.

Das Model (nicht für alle Anwendungen berechtigt)

Wir betrachten eine Quelle mit Alphabet $\Sigma = \{a_1, \ldots, a_n\}$, wobei jedes Quellsymbol a_i, $i = 1 \ldots n$, mit Wahrscheinlichkeit $P(a_i)$ auftritt. Dabei sollen die Ereignisse „a_i erscheint bei einer Nachricht an Stelle j“ und „a_k erscheint bei einer Nachricht an Stelle l“ unabhängig für alle $i, j, k, l, (j \neq l)$ sein, d.h. daß die Wahrscheinlichkeit, daß eine Nachricht der Länge r die Form $a_{i_1} \cdots a_{i_r}$ hat, durch $P(a_{i_1}) \cdot \ldots \cdot P(a_{i_r})$ berechnet werden kann.

Definition: erwartete Länge eines Codewortes

Sei $K : \{a_1, \ldots, a_n\} \to \Gamma^*$ eine Codierung, $d_i = |K(a_i)|$ und $P(a_i)$ die Wahrscheinlichkeit, daß das Quellsymbol a_i auftritt, wobei $i = 1 \ldots n$ ist. Die **erwartete Länge** eines Codewortes ist $L_K = \sum_{i=1}^{n} d_i \cdot P(a_i)$.

Definition: Huffman-Codierung

Eine Huffman-Codierung $K : \{a_1, \ldots, a_n\} \to \Gamma^*$ ist eine unmittelbare Codierung mit minimaler erwarteter Länge eines Codewortes.

Beispiel:

Quellalphabet	A	B	C	D	E	F
Wahrscheinlichkeiten	0.4	0.1	0.1	0.1	0.2	0.1
Codelänge (Vorschlag)	1	4	4	4	2	4
Codewörter	0	1100	1101	1110	10	1111

Die durch die obige Tabelle definierte Codierung und die vorgeschlagenen Codelängen erfüllen die Kraftsche Ungleichung, denn $\frac{1}{2^1} + \frac{1}{2^2} + 4 \cdot \frac{1}{2^4} = \frac{1}{2} + \frac{1}{4} + \frac{4}{16} = 1 \leq 1$. Somit ist eine unmittelbare Codierung mit den angegebenen Codewortlängen möglich. Die erwartete Länge eines Codewortes ist bei diesem unmittelbaren Code $L = 1 \cdot 1 \cdot 0.4 + 4 \cdot 4 \cdot 0.1 + 2 \cdot 1 \cdot 0.2 = 2.4$. Die Frage ist nun: Ist die obige Codierung eine Huffmann-Codierung, bzw. ist L minimal?

2.2.1 Aufbau von binären Huffman-Codierungen

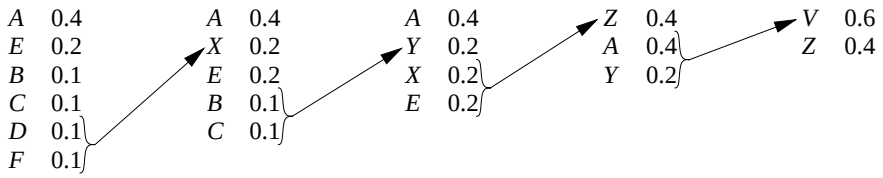
Sei $\Sigma = \{a_1, \ldots, a_n\}$ und $P(a_i)$, $i = 1 \ldots, n$, die Wahrscheinlichkeit, daß das Quellsymbol a_i auftritt. Der folgende Algorithmus ermittelt eine Huffman-Codierung für Σ und P.

(rekursiver) Algorithmus:

1. Wenn $n \leq 2$, dann codiere direkt (Codewörter sind 0 und 1).
2. Ordne $a_1, \ldots, a_n$ nach Wahrscheinlichkeiten, so daß $P(a_{i_1}) \geq P(a_{i_2}) \geq \ldots \geq P(a_{i_n})$.
3. Ersetze $a_{i_{n-1}}$ und a_{i_n} durch ein neues Symbol a^* und setze $P(a^*) := P(a_{i_{n-1}}) + P(a_{i_n})$.
4. Finde eine Huffman-Codierung für $a_{i_1}, \ldots, a_{i_{n-2}}, a^*$.
5. ersetze das Codewort $K(a^*)$ durch die zwei Codewörter: $K(a_{i_{n-1}}) := K(a^*)1$ und $K(a_{i_n}) := K(a^*)0$.

Beispiel: Huffman-Codierung (Praxis)

Reduktion



Aufbau des Codes

$K(V)=0$	$K(Z)=1$	$K(A)=00$	$K(A)=00$	$K(A)=00$
$K(Z)=1$	$K(A)=00$	$K(Y)=01$	$K(X)=10$	$K(E)=11$
	$K(Y)=01$	$K(X)=10$	$K(E)=11$	$K(B)=010$
		$K(E)=11$	$K(B)=010$	$K(C)=011$
			$K(C)=011$	$K(D)=100$
				$K(F)=101$

2.2.2 Korrektheit des Algorithmus

Lemma 34 *Der Algorithmus konstruiert Huffman-Codierungen, d.h. die erwartete Codelänge ist optimal.*

Beweis:

1. Sei K eine Huffman-Codierung für $\Sigma = \{a_1, \ldots, a_n\}$ und P. Sei $d_i = |K(a_i)|$ für $i = 1 \ldots n$. Sind die Quellsymbole entsprechend ihrer Wahrscheinlichkeiten geordnet, d.h. $P(a_{i_1}) \geq P(a_{i_2}) \geq \ldots \geq P(a_{i_n})$, so gilt, daß $d_{i_1} \leq d_{i_2} \leq \ldots \leq d_{i_n}$ ist.

   Beweis: Falls $d_{i_j} > d_{i_k}$ und $j < k$ ist, dann können wir die Codewörter von a_{i_j} und a_{i_k} vertauschen. Hierdurch wird die erwartete Codelänge höchstens kleiner:

$$L_{K,\text{neu}} = L_{K,\text{alt}} - d_{i_j}P_{i_j} - d_{i_k}P_{i_k} + d_{i_j}P_{i_k} + d_{i_k}P_{i_j} =$$

$$L_{K,\text{alt}} + \underbrace{(d_{i_k} - d_{i_j})}_{<0}\underbrace{(P_{i_j} - P_{i_k})}_{\geq 0}.$$

2. Es gibt eine Huffman-Codierung K_1, so daß $K_1(a_{i_{n-1}})$ und $K_1(a_{i_n})$ sich nur im letzten Bit unterscheiden.
   Beweis:
   Nehme eine Huffman-Codierung K_0, entferne das letzte Zeichen von $K_0(a_{i_n})$. Man erhält eine Codierung $\tilde{K}_0$. Da $L_{\tilde{K}_0} < L_K$, ist $\tilde{K}_0$ keine unmittelbare Codierung. Es folgt, daß ein j existiert mit $\tilde{K}_0(a_{i_n})$ ist Präfix von $\tilde{K}_0(a_{i_j})$. Da a_{i_n} in K_0 ein längstes Codewort hatte, müssen sich $K_0(a_{i_n})$ und $K_0(a_{i_j})$ im letzten Bit unterscheiden und nach Punkt (1.) ist $d_{i_j} = d_{i_{j+1}} = d_{i_{j+2}} = \ldots = d_{i_n}$. Damit erhält man K_1 indem man definiert: $K_1(a_{i_j}) := K_0(a_{i_{n-1}})$, $K_1(a_{i_{n-1}}) := K_0(a_{i_j})$ und für $k \in \{1, \ldots, n\} \backslash \{j, n-1\} : K_1(a_{i_k}) := K_0(a_{i_k})$.

3. Das Lemma wird mit vollständiger Induktion über die Anzahl der Quellsymbole bewiesen.

   IA: Das Lemma gilt trivialerweise für Quellen mit bis zu zwei Symbolen.

   IV: Das Lemma gelte für Quellen mit $|\Sigma| < n$.

   IS: Sei nun das Quellalphabet $\Sigma = \{a_1, \ldots, a_n\}$ und K_1 eine Huffman-Codierung für Σ mit der Eigenschaft von (2.) und $d_{i_1}, \ldots, d_{i_n}$ die Längen der Codewörter in K_1. Definiere die Codierung K_1^* auf Quellalphabet $\{a_1, \ldots, a_{n-2}, a^*\}$ indem die Symbole $a_{i_{n-1}}$ und a_{i_n} in das neue Symbol a^* verschmolzen werden, das Codewort $K_1(a_{i_n})$ ohne das letzte Bit als Codewort $K_1^*(a^*)$ genommen wird und $P(a^*)$ auf $P(a_{i_{n-1}}) + P(a_{i_n})$ gesetzt wird. Dann gilt:

$$L_{K_1^*} = L_{K_1} - (d_{i_n}(P(a_{i_{n-1}}) + P(a_{i_n})) + (d_{i_n} - 1)(P(a_{i_{n-1}}) + P(a_{i_n}))$$

   Also $L_{K_1} = L_{K_1^*} + P(a_{i_{n-1}}) + P(a_{i_n})$.

Sei K^* die Huffman-Codierung für $a_{i_1}, \ldots, a_{i_{n-2}}, a^*$, die der Algorithmus in der Rekursion berechnet hat (IV !), und K die durch den Algorithmus für $a_{i_1}, \ldots, a_{i_n}$ erhaltene Codierung. Auf ähnliche Art und Weise wie oben zeigt man: $L_K = L_{K^*} + P(a_{i_{n-1}}) + P(a_{i_n})$.

Da K^* eine Huffman-Codierung ist, ist $L_{K^*} \leq L_{K_1{}^*}$. Also ist $L_K \leq L_{K_1}$. Weil L_{K_1} minimal ist, folgt, daß $L_K = L_{K_1}$ sein muß, d.h. L_K ist minimal und somit liefert der Algorithmus eine Huffman-Codierung K. □

2.2.3 Nicht binäre Huffman-Codierungen

Der Algorithmus muß für $|\Gamma| = k > 2$ folgendermaßen modifiziert werden:

1. Bei der ersten Reduktion werden $i \in \{2, \ldots k\}$ Quellsymbole verschmolzen, so daß gilt: $|\Sigma| - i + 1 \equiv 1 \bmod (k-1)$, da in jedem weiteren Reduktionsschritt (s.u.) k Symbole zu einem neuen verschmolzen werden (totale Änderung des Symbolanzahl: $k-1$) und die Rekursion beendet ist, wenn nur noch k Symbole zu codieren sind.

2. Bei den nächsten Reduktionen werden jeweils k Symbole zu einem neuen Symbol verschmolzen.

2.2.4 Blockquellen

Definition

Sei S eine Quelle mit Quellalphabet Σ. Dann ist S^k eine neue Quelle, **Blockquelle** genannt, mit Quellalphabet Σ^k. Ist das betrachtete Modell, daß die Quellzeichen unabhängig voneinander generiert werden, so induziert die Quelle S mit Wahrscheinlichkeit $P(a)$ für $a \in \Sigma$ die Wahrscheinlichkeit für die Blockquelle S^k durch $P(a_1 \cdots a_k) := P(a_1) \cdot \ldots \cdot P(a_k)$ für alle $a_1 \cdots a_k \in \Sigma^k$.

Beispiel

Sei S eine binäre Quelle mit folgenden Wahrscheinlichkeiten:

$a \in \Sigma$	0	1
$P(a)$	0.9	0.1

. Im folgenden werden die erwarteten Codelängen von Huffman-Codierungen betrachtet.

1. Für S ist die Huffman-Codierung trivial und $L_K = 1$.

2. Für S^2 ergeben sich die induzierten Wahrscheinlichkeiten und die Huffman-Codeworte durch:

$a \in \Sigma^2$	00	01	10	11
$P(a)$	0.81	0.09	0.09	0.01
Huffman-Codewort(a)	0	10	110	111

. Damit ergibt sich die erwartete Codewortlänge durch $L_K = 1.29$, wofür $L_K/2 = 0.645$ Codebits pro Symbol benötigt werden.

3. Für S^3 ist $L_K = 1.598$, wofür $L_K/3 = 0.533$ Codebits pro Symbol benötigt werden.

Problem : Wie weit kann man die Anzahl von Codebits pro Symbol reduzieren?

Eine Antwort hierauf wollen wir im folgenden mit Hilfe der Entropie einer Wahrscheinlichkeitsverteilung geben.

2.3 Informationstheorie - Entropie

2.3.1 Entropie

Intuition zur Entropie

Die Entropie ist der „mittlere Informationsgehalt pro Zeichen“

oder

die „erwartete Überraschung durch ein Zeichen der Nachricht“

Beispiel

Quellalphabet: $\Sigma = \{a, b\}$; Wahrscheinlichkeiten: $P(a) = 0.9999, P(b) = 0.0001$.

Bei einer solchen Quelle stellt der Empfang von a keine große Überaschung dar. Wenn man hingegen b enthält, so ist dies eine große Überraschung und somit enthält b viel Information (analog funktionieren die Nachrichten in Fernsehen – man spricht nur über Ereignisse die wenig wahrscheinlich sind, z.B. Flugzeugkatastrophen).

Problem: Wie kann/soll man die „Überraschung“ messen?

Notation für Entropie

$H(p_1, \ldots, p_n)$ steht für die Entropie einer Quelle mit einem Alphabet von n Zeichen, die mit den Wahrscheinlichkeiten $p_1, \ldots, p_n$ auftreten. Falls S eine bestimmte Quelle (mit bekannten Wahrscheinlichkeiten) bezeichnet, dann ist $H(S)$ die entsprechende Entropie.

Notwendige Eigenschaften der Entropie:

1. $H(p_1, \ldots, p_n) \geq 0$
2. Die Funktion H ist stetig.
3. $H(p_1, \ldots, p_n) = H(p_{\pi(1)}, \ldots, p_{\pi(n)})$ für jede Permutation π
4. Coherence:
   $H(p_1, \ldots, p_n) = H(p_1 + p_2, p_3, \ldots, p_n) + (p_1 + p_2) \cdot H(\frac{p_1}{p_1+p_2}, \frac{p_2}{p_1+p_2})$

   Begründung: Betrachte einen Prozeß, in dem wir jeweils ein Symbol empfangen, aber zuerst nicht zwischen a_1 und a_2 unterscheiden (als ob wir a_1 und a_2 ohne Lesebrillen nicht unerscheiden könnten, jedoch alle anderen Symbole identifizieren könnten). Falls wir feststellen, daß wir entweder a_1 oder a_2 empfangen haben, holen wir die Lesebrille und überprüfen, welches Zeichen wir erhalten haben.

   Damit ist $H(p_1 + p_2, p_3, \ldots, p_n) + (p_1 + p_2) \cdot H(\frac{p_1}{p_1+p_2}, \frac{p_2}{p_1+p_2})$ unsere Gesamt-„Überraschung“. Hätten wir sofort die Lesebrille aufgesetzt, dann wäre die Überraschung $H(p_1, \ldots, p_n)$.

Theorem 35 *Falls H die Bedingungen (1) bis (4) erfüllt, dann existiert eine Konstante $k > 0$, so daß gilt:*

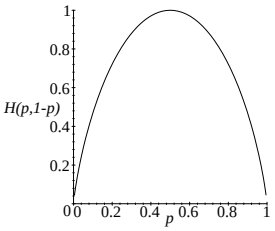
$$H(p_1, \ldots, p_n) = k \cdot \sum_{\substack{i \\ p_i \neq 0}} p_i \cdot \log \frac{1}{p_i}.$$

Definition: Entropie

$$H(p_1, \ldots, p_n) = \sum_i p_i \cdot \log_2 \frac{1}{p_i}.$$

wobei wir $0 \cdot \log \frac{1}{0}$ als 0 berachten (dies ist sinnvoll, da $\lim_{x \to 0} x \cdot \log \frac{1}{x} = 0$).

Beispiel zur Entropie



Es gebe zwei Quellenzeichen mit Wahrscheinlichkeiten p und $1-p$. Somit gilt:

$$H(p, 1-p) = p \cdot \log \frac{1}{p} + (1-p) \cdot \log \frac{1}{1-p}.$$

Auf dem Bild ist leicht zu erkennen, daß:
- die Funktion $H(p, 1-p)$ symmetrisch ist,
- das Maximum von H bei $p = \frac{1}{2}$ und
- das Minimum bei $p = 0$ und $p = 1$ erreicht wird.

Lemma 36

1. *$H(p_1, \ldots, p_n) = 0 \iff \exists i \in \{1, \ldots, n\} : p_i = 1$.*
2. *$H(p_1, \ldots, p_n) \leq \log n$;*
   wobei die Gleichheit gilt, genau dann wenn $p_i = \frac{1}{n}$ für alle $i \in \{1, \ldots, n\}$.

Intuitiv: Wir verschicken viel Information falls alle Symbole gleich wahrscheinlich sind.

Beweis:

1. offensichtlich
2. $H(p_1, \ldots, p_n) - \log n = \sum p_i \log \frac{1}{p_i} - \sum p_i \log n = \frac{1}{\ln 2} \sum p_i (\ln \frac{1}{p_i} - \ln n) = \frac{1}{\ln 2} \sum p_i (\ln \frac{1}{p_i \cdot n}) \leq \frac{1}{\ln 2} \sum_{i=1}^{n} p_i (\frac{1}{p_i \cdot n} - 1) = \frac{1}{\ln 2} (\sum_{i=1}^{n} \frac{1}{n} - \sum_{i=1}^{n} p_i) = \frac{1}{\ln 2}(1-1) = 0$ □

2.3.2 Entropie versus erwartete Codelänge

Theorem 37 *Sei K eine unmittelbare binäre Codierung für eine Quelle S und L_K die erwartete Länge der Codierung K. Dann gilt:*

$$L_K \geq H(S).$$

Beweis: Sei d_i die Länge des i-ten Codewortes. Dann ist die erwartete Codewortlänge $L_K = \sum p_i d_i = \sum p_i \log 2^{d_i}$. Also

$$\begin{aligned} H(S) - L_K &= \sum p_i \log \tfrac{1}{p_i} - \sum p_i \log 2^{d_i} = \sum p_i \log \tfrac{1}{p_i \cdot 2^{d_i}} \\ &\leq \tfrac{1}{\ln 2} \sum p_i (\tfrac{1}{p_i \cdot 2^{d_i}} - 1) = \tfrac{1}{\ln 2} \left(\sum \tfrac{1}{2^{d_i}} - \sum p_i \right) = \tfrac{1}{\ln 2} (\sum \tfrac{1}{2^{d_i}} - 1). \end{aligned}$$

Nach Kraftscher Ungleichung ist die letzte Zahl nicht größer als 0. Es folgt, daß $H(S) - L_K \leq 0$. □

Theorem 38 [Erstes Shannonsches Theorem] *Für jede Quelle S gilt*

$$L_{\min}(S) \leq H(S) + 1$$

wobei $L_{\min}$ die minmal mögliche erwartete Länge einer Codierung für S ist.

Beweis:

1. Nehmen wir an, daß $P(a_i) = p_i > 0$ für alle $i \in \{1, \ldots, n\}$. Sei $d_i \in \mathbb{N}$, so daß
$$\log \frac{1}{p_i} \leq d_i < 1 + \log \frac{1}{p_i}.$$
Dann:
$$\sum \frac{1}{2^{d_i}} \leq \sum 1/2^{\log \frac{1}{p_i}} = \sum p_i = 1.$$
Also ist die Kraftsche Ungleichung erfüllt und es gibt eine unmittelbare Codierung K mit Codelängen $d_1, \ldots, d_n$. Für diese Codierung gilt:
$$L_K = \sum p_i d_i < \sum p_i (1 + \log \frac{1}{p_i}) = \sum p_i + \sum p_i \log \frac{1}{p_i} = 1 + H(S).$$

2. Sei nun $p_i = 0$ erlaubt.
Definiere, $p_i^{(k)}$, so daß $\lim_{k\to\infty} p_i^{(k)} = p_i$ und $p_i^{(k)} > 0$. Sei $S^{(k)}$ die entsprechende Quelle. Dann gilt nach Punkt 1, daß $L_{\min}(S^{(k)}) \leq 1 + H(S^{(k)})$. Außerdem ist $\lim_{k\to\infty} H(S^{(k)}) = H(S)$, weil die Entropie stetig ist. Es ist auch $\lim L_{\min}(S^{(k)}) = L_{\min}(S)$, da eine sehr kleine Änderung der Wahrscheinlichkeit unsere Konstruktion der Huffman-Codierung nicht ändert. Es folgt, daß $L_{\min}(S) \leq 1 + H(S)$. □

Entropie für Blockquellen

Lemma 39 $$H(S^k) = k \cdot H(S)$$

Intuitiv: Der Informationsgehalt bei einem Block aus k Symbolen ist k mal der Informationsgehalt von einem Symbol.

Beweis: Wir zeigen das Lemma für $k = 2$. Andere Fälle sind ähnlich.

$$\begin{aligned}
H(S^2) &= \sum_{i,j} \left(\text{Prob}(a_i a_j) \cdot \log \frac{1}{\text{Prob}(a_i a_j)} \right) \\
&= \sum_{i,j} \left(P(a_i) \cdot P(a_j) \cdot \log \frac{1}{P(a_i) \cdot P(a_j)} \right) \\
&= \sum_{i} \left(\sum_{j} P(a_i) \cdot P(a_j) \cdot \left(\log \frac{1}{P(a_i)} + \log \frac{1}{P(a_j)} \right) \right) \\
&= \sum_{i} \left(P(a_i) \cdot \sum_{j} \left(P(a_j) \cdot \log \frac{1}{P(a_i)} + P(a_j) \cdot \log \frac{1}{P(a_j)} \right) \right) \\
&= \sum_{i} \left(P(a_i) \cdot \left(\log \frac{1}{P(a_i)} + H(S) \right) \right) \\
&= \sum_{i} \left(P(a_i) \cdot \log \frac{1}{P(a_i)} \right) + \sum_{i} \left(P(a_i) \cdot H(S) \right) = 2 \cdot H(S)
\end{aligned}$$

□

Die nächste Folgerung zeigt, daß man durch Anwendung von Blockquellen die Entropie als erwartete Codelänge fast erreichen kann.

Korollar 40 [zum ersten Shannon Theorem]

$$\frac{L_{\min}(S^{(k)})}{k} \leq H(S) + \frac{1}{k}$$

Also beträgt die erwartete Codelänge für **ein** Zeichen des Quellalphabets $\approx H(S)$.

Beweis:

$$L_{\min}(S^{(k)}) \leq H(S^k) + 1 = k \cdot H(S) + 1$$

□

2.3.3 Entropie für nicht binäre Codealphabete

Sei r die Anzahl der Codesymbole: z.B. $r = 4$. Um $L_{\min} \geq H(S)$ zu gewährleisten muß auch $H(S)$ zweimal kleiner werden (bei Huffman-Codierungen: 1 Symbol ersetzt jetzt 2 Bits). Es reicht also $\log_2 \frac{1}{p_i}$ in der Definition von Entropie durch $\log_4 \frac{1}{p_i}$ zu ersetzen. Allgemein: $\log_2 \frac{1}{p_i}$ wird in der Definition durch $\log_r \frac{1}{p_i}$ ersetzt.

2.4 Einige andere Codierungen

Huffman-Codierungen sind optimal, **aber** die Codelängen müssen aufwendig berechnet werden, da jede Länge von der Gesamtheit der Wahrscheinlichkeiten abhängt. Bei Änderungen der Wahrscheinlichkeiten müßen alle Codeworte neu bestimmt werden. Deswegen opfert man die Optimalität und sucht andere etwas bequemere Codierungen.

2.4.1 Shannon-Fano Codierung

Die erwartete Codelänge kann bei Shannon-Fano Codierungen größer sein als Huffman-Codierungen, aber die Codewortlängen sind *direkt* aus den Wahrscheinlichkeiten berechenbar.

Definition

Sei S eine Quelle mit Wahrscheinlichkeiten $P(a_i) = p_i$ und $r = |\Gamma|$ die Anzahl der Codesymbole. Dann wird die Codewortlänge l_i für a_i so gewählt, daß

$$\log_r(1/p_i) \leq l_i < \log_r(1/p_i) + 1.$$

Durch Umformung ergibt sich:

$$\frac{1}{p_i} \leq r^{l_i} < \frac{r}{p_i}, \quad p_i \geq \frac{1}{r^{l_i}} > \frac{p_i}{r}.$$

Summation über i liefert, daß die Kraftsche Ungleichung erfüllt ist:

$$1 \geq \sum \frac{1}{r^{l_i}} > \frac{1}{r}.$$

Nach dem Algorithmus aus Kapitel 2.1.3 konstruieren wir eine unmittelbare Codierung mit den Codewortlängen $l_1, \ldots, l_n$. Jede solche Codierung ist eine **Shannon-Fano Codierung**.

Erwartete Codewortlänge bei Shannon-Fano Codierungen

Es gilt $p_i \log_r(1/p_i) \leq p_i l_i < p_i \log_r(1/p_i) + p_i$ für jedes i. Durch summieren über alle i erhalten wir

Lemma 41 *$H_r(S) \leq L_K \leq H_r(S)+1$, wobei K eine Shannon-Fano Codierung mit r Codesymbolen für die Quelle S ist.*

Beispiel: Sei S eine Quelle mit n Symbolen, die alle mit der gleichen Wahrscheinlichkeit aufteten. Dann hat bei einer binären Shannon-Fano Codierung jedes Codewort die gleiche Länge $l = \lceil \log q \rceil$. Ist n keine Zweierpotenz, so sind bei einer binären Huffman-Codierung einige Symbole durch Codeworte der Länge $l - 1$ codiert.

2.4.2 Universelle Codierungen

Sei $\kappa : \mathbb{N} \to \{0,1\}^*$ eine Funktion mit der Eigenschaft, daß

$$\forall i, j \in \mathbb{N}; i \neq j : \kappa(i) \text{ ist kein Präfix von } \kappa(j).$$

κ definiert eine **universelle Codierung** auf folgende Weise:

1. Ordne die Quellsymbole entsprechend ihrer Wahrscheinlichkeiten (die mit großen Wahrscheinlichkeiten nach vorne);
2. das i-te Zeichen codiere durch $\kappa(i)$, für jedes i.

Anmerkung: Die Codeworte sind also (fast) unabhängig von der Wahrscheinlichkeitsverteilung der Quellsymbole bestimmt und die Codeworte einer Quelle mit i Symbolen ist eine Teilmenge von jeder Quelle mit $i+1$ Symbolen.

Beispiele für universelle Codierungen sind: Elias Codes, Fibonacci Codes.

2.4.3 Elias Codes

Codierung γ

Für $x \in \mathbb{N}$ setze

$$\gamma(x) = \underbrace{0 \ldots 0}_{\lfloor \log x \rfloor} D(x)$$

wobei $D(x)$ die binäre Darstellung von x ist.

- Die Codierung γ ist unmittelbar: für $x \neq y$ ist $\gamma(x)$ kein Präfix von $\gamma(y)$.
  Beweis: die führenden Nullen zeigen eindeutig an, wie lang die binäre Darstellung ist. □
- Die erwartete Codewortlänge kann fast zweimal größer als die Entropie sein.
  Beispiel: S enthält 2^l Zeichen, die gleich wahrscheinlich sind. $H(S) = \log 2^l = l$. Da fast alle Codewörter die Länge $2l$ haben, ist die erwartete Codewortlänge fast $2l$.

Codierung δ

Für $x \in \mathbb{N}$ setze

$$\delta(x) = \gamma(\lfloor \log x \rfloor + 1) T(x)$$

wobei $T(x)$ die binäre Darstellung von x ohne die führende Eins ist.

- Die Codierung δ ist unmittelbar.
  Beweis: Da γ unmittelbar ist, müssen nur Codeworte x und y mit $\gamma(\lfloor \log x \rfloor + 1) = \gamma(\lfloor \log y \rfloor + 1)$ also $\lfloor \log x \rfloor = \lfloor \log y \rfloor$ betrachtet werden. Diese unterscheiden sich aber gerade in der Darstellung $T(X)$. □
- Die Codeworte $\delta(x)$ sind asymptotisch viel kürzer als $\gamma(x)$ (die zusätzliche Folge, die vor der „binären Darstellung“ von x steht, hat bei $\delta(x)$ nur eine logarithmische Länge).

Beispiele von Elias- und Fibonacci-Codewörtern

x	$\gamma(x)$	$\delta(x)$	Fibonacci Codewort
1	1	1	11
2	010	0100	011
3	011	0101	0011
4	00100	01100	1011
5	00101	01101	00011
6	00110	01110	10011
7	00111	01111	01011
8	0001000	00100000	000011
16	000010000	001010000	0010011
17	000010001	001010001	1010011
32	00000100000	0011000000	00101011

2.4.4 Fibonacci Code

Definition: Fibonacci Zahlen

$$F_0 = 1, \qquad F_1 = 1, \qquad F_i = F_{i-2} + F_{i-1} \text{ für } i > 1$$

Lemma 42 *Jede natürliche Zahl n kann eindeutig als*

$$\sum_{i=1}^{n} d_i \cdot F_i$$

dargestellt werden, wobei $d_i \in \{0,1\}$ und keine benachbarte Koeffizienten d_i, d_{i+1} gleichzeitig 1 sind.

Existenzbeweis durch vollständige Induktion:

- $1 * F_1 = 1$.
- Sei bereits gezeigt, daß alle Zahlen $< m$ sich wie oben angegeben angegeben darstellen lassen.
- Sei $\sum_{i=1}^{n} d_i \cdot F_i = m-1$, wobei $d_i \in \{0,1\}$ und keine benachbarte Koeffizienten d_i, d_{i+1} gleichzeitig 1 sind. Ist $d_1 = 0$, so setze $d_1 = 1$, andernfalls ist $d_2 = 0$ und setze $d_1 = 0; d_2 = 1$. Dann gilt: $\sum_{i=1}^{n} d_i \cdot F_i = m$.

  Finde ein $j \in \{1, \ldots, n-1\}$ mit $d_j = d_{j+1} = 1$ (Invariante: es existiert höchstens *ein* solches). Existiert ein solches, so setze $d_j = d_{j+1} = 0$ und $d_{j+2} = 1$ (Invariante: vorher galt $d_{j+2} = 0$). Hierdurch verändert sich die Summe nicht. Da die Folge der gefundenen j monoton wächst, terminiert die Prozedur und liefert eine Darstellung für m, in der keine benachbarten Koeffizienten d_i, d_{i+1} gleichzeitig 1 sind. □
- Die Berechnung der ersten Darstellungen:

  $1 \to 1$
  $2 \to 01$
  $3 \to 11 \to 001$
  $4 \to 101$
  $5 \to 011 \to 0001$
  $6 \to 1001$
  $7 \to 0101$
  $8 \to 1101 \to 0011 \to 00001$
- Die Einzigkeit folgt aus dem Bildungsgesetz der Fibonacci Zahlen.

Definition: Fibonacci Code

Sei $x = \sum_{i=1}^{k} d_i \cdot F_i$, wobei $d_k = 1$ und $d_i \in \{0,1\}$ so gewählt worden sind, daß keine benachbarten Koeffizienten d_i, d_{i+1} gleichzeitig 1 sind. Dann ist

$$d_1 d_2 \cdots d_k 1$$

das Fibonacci Codewort von x. Die zwei benachbarten Einsen bestimmen das Ende des Codewortes.
Es gibt keine benachbarte Einsen (mit der Ausnahme von den letzten zwei Positionen) in den Codeworten. Deswegen ist dieser Code für große Zahlen fast doppelt so lang wie ein optimaler Code.
Der Fibonacci Code ist jedoch für Zahlen, die kleiner sind als ca. 500000, besser als die Elias Codes.

2.4.5 Arithmetische Codes

Für arithmetische Codes müssen die Wahrscheinlichkeiten der Quellensymbole bekannt sein (wie bei Huffman-Codes), da in Abhängigkeit hiervon die Codewörter erzeugt werden (also sind sie keine universellen Codes). Die gesamte Nachricht wird als eine Zahl aus einem Subinterval von $[0,1)$ codiert.

Beispiel für binäre Codierung:
Das Codierungsprinzip funktioniert folgendermaßen für Wahrscheinlichkeiten p für 0 und $1-p$ für 1: Wissen wir, daß für $\alpha \in \{0,1\}^*$ das Codewort eine Zahl aus dem Subintervall $[a,b)$ ist, so ist für $\alpha 0$ das Codewort eine Zahl aus dem Intervall $[a, a+p(b-a))$ (analog für $\alpha 1$ aus dem Intervall $[b-(1-p)(b-a), b)$).

Sei $P(A) = \frac{1}{3}, P(B) = \frac{2}{3}$.

A	wird codiert durch Codewort aus Intervall	$[0\ ,\ \frac{1}{3})$
B	wird codiert durch Codewort aus Intervall	$[\frac{1}{3}\ ,\ 1)$
AA	wird codiert durch Codewort aus Intervall	$[0\ ,\ \frac{1}{9})$
AB	wird codiert durch Codewort aus Intervall	$[\frac{1}{9}\ ,\ \frac{1}{3})$
BA	wird codiert durch Codewort aus Intervall	$[\frac{1}{3}\ ,\ \frac{5}{9})$
BB	wird codiert durch Codewort aus Intervall	$[\frac{5}{9}\ ,\ 1)$
BBA	wird codiert durch Codewort aus Intervall	$[\frac{5}{9}\ ,\ \frac{19}{27})$
$\vdots$		

Implementation:

- Jedes Intervall wird durch eine Zahl aus dem Intervall repräsentiert. Um ein Intervall $T \subseteq [0,1]$ der Länge s so darzustellen, nimmt man die einzige Zahl $i \in T$ mit Binärdarstellung der Länge $\log_2(1/s)$.

- Eine Nachricht $a_1 a_2 \cdots a_k$ entspricht einem Interval der Länge $s = P(a_1) \cdot P(a_2) \cdot \ldots \cdot P(a_k)$. Die entsprechende Zahl hat die Länge $\sum_{i=1}^{k} \log(1/P(a_i))$. Die erwartete Länge beträgt

$$\sum_{i=1}^{k} \sum_{j} P(a_j) \cdot \log(1/P(a_j)) = k \cdot H(S)$$

  Also erreicht der arithmetische Code eine erwartete Codewortlänge, die fast gleich der Entropie ist (aber nicht gleich, $\log(1/s)$ muß manchmal aufgerundet werden).

- Es gibt Probleme bei der Decodierung: eine Zahl kann einer Nachricht unterschiedlicher Länge entsprechen. In unserem Beispiel kann das Codewort 0 den Nachrichten A, AA, AAA, usw. entsprechen. Dieses Problem kann auf die folgenden zwei Arten gelößt werden:
  - die Nachrichtenlänge wird separat codiert, oder
  - jede Nachricht wird durch ein extra Zeichen, das das Ende der Nachricht markiert, abgeschlossen.

2.5 Varianten von Huffman-Codierungen

Die Huffman-Codierungen, die wir kennengelernt haben, haben zwei Nachteile:

(1) Huffman-Codierungen ignorieren die Tatsache, daß in allen Texten starke Abhängigkeiten zwischen den Auftrittswahrscheinlichkeiten der Quellsymbole bestehen. Zum Beispiel kommt in den englischen Texten nach einem „t“ sehr oft ein „h“ vor und zwar viel öfter als die Wahrscheinlichkeit von einem „h“ in dem gesamten Text.

(2) Die Konstruktion ist nur dann möglich, wenn die Wahrscheinlichkeiten bekannt sind. Man könnte die Wahrscheinlichkeiten für einen Text ausrechnen, aber das benötigt einen extra Lauf durch die Datei.

2.5.1 Prädiktive Huffman-Codierungen

Annahme über den zu komprimierenden Text: Für viele Quellsymbole unterscheiden sich für $t_1, t_2 \in \Sigma^l$ die Wahrscheinlichkeiten Prob(x| die Folge vor x ist t_1) und Prob(x| die Folge vor x ist t_2) stark, d.h.

$$\text{Prob}(x| \text{ die Folge vor } x \text{ ist } t \in \Sigma^l) \not\approx P(x)/|\Sigma|^l.$$

Prädiktive Huffman-Codierungen nutzen diese Eigenschaft aus.

Beispiel: In deutschsprachigen Texten kommt nach der Folge „␣`ve`“ sehr häufig ein „`r`“ vor. Bei Huffman-Codierungen fließt diese Tatsache überhaupt nicht ein.

Vorschläge zur Dateikomprimierung

Vorschlag 1

Man wendet die Huffman-Codierung nicht auf Buchstaben an, sondern auf Blöcke fester Länge.
Nachteil: Blöcke trennen die Worte ziemlich zufällig.

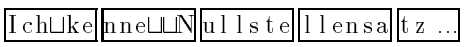


Abbildung 2.2: Unterteilung von Text in Blöcke

Vorschlag 2 (prädiktive Huffman-Codes)

Man betrachtet den Text als Markovschen Prozeß.

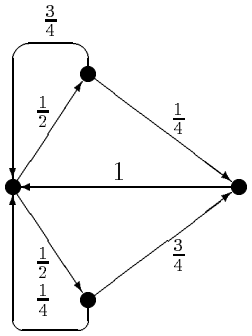


Abbildung 2.3: Gerichteter Graph

Definition: Markovscher Prozeß

- Bei einem Markovschen Prozeß ist ein gerichteter Graph gegeben, bei dem jede Kante eine Wahrscheinlichkeit als Etikett trägt. Für jeden Knoten x ist deshalb die Summe der Wahrscheinlichkeiten, der von x ausgehenden Kanten, gleich 1.
- Ein Markovscher Prozeß ist dann eine Wanderung durch den Graphen. Die Auswahl der Wege erfolgt gemäß den Wahrscheinlichkeiten: der Übergang zum nächsten Knoten wird nach den Wahrscheinlichkeiten der Ausgangskanten gewählt.
- Der Prozeß ist **ergodisch**, falls für alle Knoten x und y ein Pfad von x nach y existiert (der Graph ist stark zusammenhängend). Für ergodische Prozesse existiert eine umfangreiche Literatur.

Übergangsmatrix

$$\begin{array}{c} \\ a \\ b \\ c \end{array} \begin{array}{c} \begin{array}{ccc} a & b & c \end{array} \leftarrow \text{Knoten} \\ \begin{pmatrix} p_{aa} & p_{ab} & p_{ac} \\ p_{ba} & p_{bb} & p_{bc} \\ p_{ca} & p_{cb} & p_{cc} \end{pmatrix} \end{array}$$

Abbildung 2.4: Übergangsmatrix (für Knoten a, b, c)

p_{bc} = Wahrscheinlichkeit eines Übergangs von b nach c, bzw.
p_{bc} = Etikett auf der Kante $< b, c >$

Theorem 43 *Für alle Knoten a existiert bei einer ergodischen Markovkette* $\lim_{t\to\infty}$ *Prob(wir befinden uns nach t Schritten in a) und ist unabhängig vom Anfangsknoten.*
Die Grenzwerte kann man durch die Übergangsmatrix U berechnen:
Ist $U' = \lim_{t\to\infty} U^t$, *dann ist* $(p_a, p_b, \ldots, p_c) \cdot U' = (p_a, p_b, \ldots, p_c)$.

2.5.2 Prädiktive Codierung

- Wir wählen einen Parameter m (Anzahl der Buchstaben in einem Block), so daß die Wahrscheinlichkeit von jedem Buchstaben *praktisch* nur von m

vorhergehenden Buchstaben abhängt. Für praktische Zwecke soll m klein gewählt werden ($m \leq 10$).

- Die bedingten Wahrscheinlichkeiten müssen berechnet werden:
  $\text{Prob}(s|s_{i_1}s_{i_2}\ldots s_{i_m})$ ($s_{i_1}\ldots s_{i_m}$ ist der Text vor s).
- Wir definieren einen Markovschen Prozeß, in dem die Knoten die Folgen von m Buchstaben sind. Für jeden Buchstaben s gibt es die Kanten: $(s_{i_1}\ldots s_{i_m}) \rightarrow (s_{i_2}\ldots s_{i_m}s)$ denen die Wahrscheinlichkeiten $\text{Prob}(s|s_{i_1}\ldots s_{i_m})$ zugeordnet sind. Dann entspricht jeder Text dem Ablauf eines Markovschen Prozesses. Für jeden Knoten des Graphen sind die Ausgänge mit Hilfe von (statischen) Huffman-Codes codiert. Sei also $R(x|s_1\cdots s_m)$ der Code für die Kante $(s_{i_1}\ldots s_{i_m}) \rightarrow (s_{i_2}\ldots s_{i_m}x)$. Dann wird ein Text $a_1\cdots a_k$ durch

  $$a_1a_2\cdots a_mR(a_{m+1}|a_1\cdots a_m)R(a_{m+2}|a_2\cdots a_{m+1})\cdots R(a_k|a_{k-m}\cdots a_{k-1})$$

  codiert.

Beispiel:

Die Quelle enthält die Buchstaben A, B, C. Die Wahrscheinlichkeiten betragen: $\text{Prob}(X|X) = 0.5$ und $\text{Prob}(X|Y) = 0.25$ für $X \neq Y$. Den Kanten im Graphen entsprechen die folgenden (statischen) Huffman-Codeworte:

$$\begin{array}{lll} R(A|A)=0 & R(B|A)=10 & R(C|A)=11 \\ R(B|B)=0 & R(A|B)=10 & R(C|B)=11 \\ R(C|C)=0 & R(A|C)=10 & R(B|C)=11 \end{array}$$

Dann wird $AAABBBCCACBBBB$ als $A0010001101011110 00$ codiert.

Vorteile

1. Für Texte bei denen starke Abhängigkeiten vorkommen ergibt sich eine viel bessere Komprimierung als bei Huffman-Codierungen.
2. Gut geeignet für off-line Methoden: man kann erst die notwendigen Statistiken erzeugen und dann entsprechend komprimieren. Dadurch kann man z.B. häufig vorkommende Wörter, Stil des Autors, ... ausnutzen.

Nachteile

1. Fehler sind auffällig (ein falscher Schritt im Graph und der Rest ist ungültig).
2. Langsam, da die Codierung mit Hilfe von großen Look-Up Tabellen geschieht.
3. Somit auch sehr großer Speicherbedarf. Verbesserung: Nur sehr häufig vorkommende Folgen werden komprimiert codiert. Die anderen Folgen werden mit einem Header unkomprimiert codiert (dies hat keinen wesentlichen Einfluß auf die erwartete Codewortlänge bzw. sollte keinen Einfluß haben).

2.5.3 Adaptive Huffman-Codes

Bei normalen Huffman Codierungen sind die Häufigkeiten der Buchstaben im Text am Anfang gegeben. Bei *adaptiven Huffman Codierungen* wird die Häufigkeitsverteilung der Buchstaben im Laufe der Komprimierung analysiert und gemäß den Teilergebnissen werden die gelesenen Teile codiert.

Vorteile: Ohne einen zusätzlichen Lauf durch das File ist eine gute Anpassung an die Charakteristik der Datei möglich. Keine Vorkenntnisse über die Dateiart sind notwendig.

Nachteile: Die Teilergebnisse müssen als Datenstruktur gespeichert werden – Laufzeit-Kosten; am Anfang ist die Komprimierung sehr ineffizient, da die eine genaue Statistik noch fehlt.

Bäume mit sibling-Eigenschaft

Sei B ein binärer Baum bei dem jeder Knoten ein Gewicht besitzt. Wir sagen, daß B die sibling-Eigenschaft erfüllt, falls

- die Knoten des Baumes so in einem eindimensionalen Array gespeichert werden können, daß die Gewichte von links nach rechts wachsen, wobei
- Geschwister in benachbarten Positionen gespeichert werden müssen (siehe Abbildung 2.5).

Lemma 44 *Ein Decodierungsbaum B entspricht einer Huffman-Codierung genau dann, wenn B die sibling-Eigenschaft erfüllt (in einem Decodierungsbaum B entspricht das Gewicht der Blätter von B der Wahrscheinlichkeit, daß das Zeichen auftritt, und das Gewicht eines inneren Knotens der Summe der Gewichte seiner Kinder).*

Beweis: (für binäre Bäume)
($\Rightarrow$) Induktion über die Anzahl der Zeichen: Jeder Decodierungsbaum mit einem Knoten erfüllt die sibling-Eigenschaft. Für $\Sigma = \{a_1, \ldots, a_n\}$ seien a_1, a_2 die Zeichen mit den geringsten Wahrscheinlichkeiten. a_1 und a_2 werden im Konstruktionsalgorithmus für Huffman Codierungen (s. S. 46) durch ein neues Zeichen a^* ersetzt. Für $a^*, a_3, \ldots, a_n$ erfüllt nach Induktionsvoraussetzung jeder Decodierungsbaum, den der Algorithmus erzeugt, die sibling-Eigenschaft. Die sibling-Eigenschaft bleibt auch am Ende des Algorithmus bestehen, weil wir die Knoten a_1 und a_2 vor die ersten Elemente des aus der Rekursion gewonnenen Arrays setzen können. Nach Konstruktion der Huffman Codierung sind a_1 und a_2 die Knoten mit den geringsten Wahrscheinlichkeiten und alle (neuen) Kinder des Knotens a^* sind an benachbarten Positionen gespeichert.
($\Leftarrow$) Ist die Anzahl der Blätter gleich 1, so gibt es nur einen Knoten und es folgt trivialerweise, daß dieser Decodierungsbaum einer Huffman Codierung entspricht. Für einen Decodierungsbaum mit n Blättern betrachte nun in dem Array die ersten Einträge $a_1, \ldots, a_k$, die zu Geschwistern mit Elter a^* gehören. Das Array um die Einträge $a_1, \ldots, a_k$ gekürzt entspricht nach Induktionsvoraussetzung einem Decodierungsbaum einer Huffman Codierung. Da $a_1, \ldots, a_k$ die Blätter mit der geringsten Wahrscheinlichkeit sind, entspricht somit auch der Decodierungsbaum mit n Blättern einer Huffman Codierung. □

Für gegebene Wahrscheinlichkeiten existieren mehrere Huffman Codierungen mit unterschiedlichen Decodierungsbäumen. Die entsprechende Arrays sind jedoch (trivialerweise) gleich. Anstelle der Anordnung der Knoten im Array an Hand der Wahrscheinlichkeitsverteilung zu betrachten, kann man äquivalent dazu die Zugriffhäufigkeitsverteilung betrachten.

Updates im Decodierungsbaum für Huffman Codierungen

Situation: Ein Buchstabe x ist empfangen worden. Dann müssen wir die vorhandene Statistik über die Häufigkeitsverteilung der Buchstaben ändern. Im Decodierungsbaum führen wir dazu die folgenden Operationen durch:

1. Der Decodierungsbaum wird (nach festen, eindeutigen Regeln) so umgestaltet, daß die Erhöhung des Zugriffhäufigkeitszähler am Blatt für x und auf dem Weg von diesem Knoten zur Wurzel keine neue Umgestaltung benötigt.

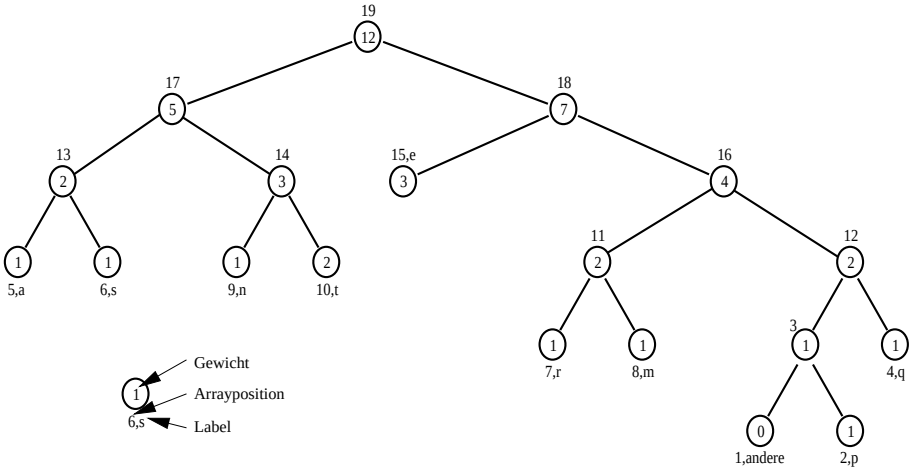


Abbildung 2.5: Ein Baum mit sibling-Eigenschaft

2. Erhöhe die Zugriffhäufigkeitszähler für alle Knoten auf dem Weg vom Blatt für x bis zur Wurzel.

Umbau des Decodierungbaumes und des Arrays:

Beobachtungen:
Sei A das Array der Zugriffhäufigkeitszählerstände, welches zu einem Decodierungsbaum B gehört.

- Der Zugriffhäufigkeitszähler für einen Knoten v mit Zugriffhäufigkeitszählerstand $A[i]$ kann gefahrlos um einen Zähler erhöht werden, wenn $A[i+1]$ *echt* größer als $A[i]$ ist.
- Ist v ein Knoten mit Zugriffhäufigkeitszählerstand $A[i]$ und ist $A[i+1] = A[i]$, so kann man die Unterbäume die zu den Knoten mit Indices i und $i+1$ in A gehören im Decodierungsbaum (und in A) vertauschen und erhält einen äquivalenten Decodierungsbaum.

Aufbauend auf diesen Beobachtungen kann man ein Verfahren entwerfen, der schnell einen Decodierungsbaum so umgestaltet, daß die Erhöhung des Zugriffhäufigkeitszähler am Blatt für x und auf dem Weg von diesem Knoten zur Wurzel keine neue Umgestaltung benötigt. Das Verfahren startet in der ersten Iteration mit dem Blatt für x, der im weiteren v heißen soll. Vertauscht diesen Knoten ggf. mit seinem rechten Nachbarn im Array A, solange bis v nicht mehr vertauscht werden muß. Unter Umständen hat der betrachtete Knoten jetzt einen neuen Elter. Mit dem Elter von v wird dann die nächste Iteration gestartet (der dann die Rolle von v übernimmt).

Anmerkungen:

- Das oben skizzierte Verfahren terminiert, weil das Array A von links nach rechts durchlaufen wird (mit jeder Elementaroperation um mindestens einen Schritt) und die Länge von A endlich ist.
- Das wiederholte Vertauschen des Knotens v mit seinem Nachbarn in A innerhalb einer Iteration kann durch eine Vertauschung mit dem letzten Element

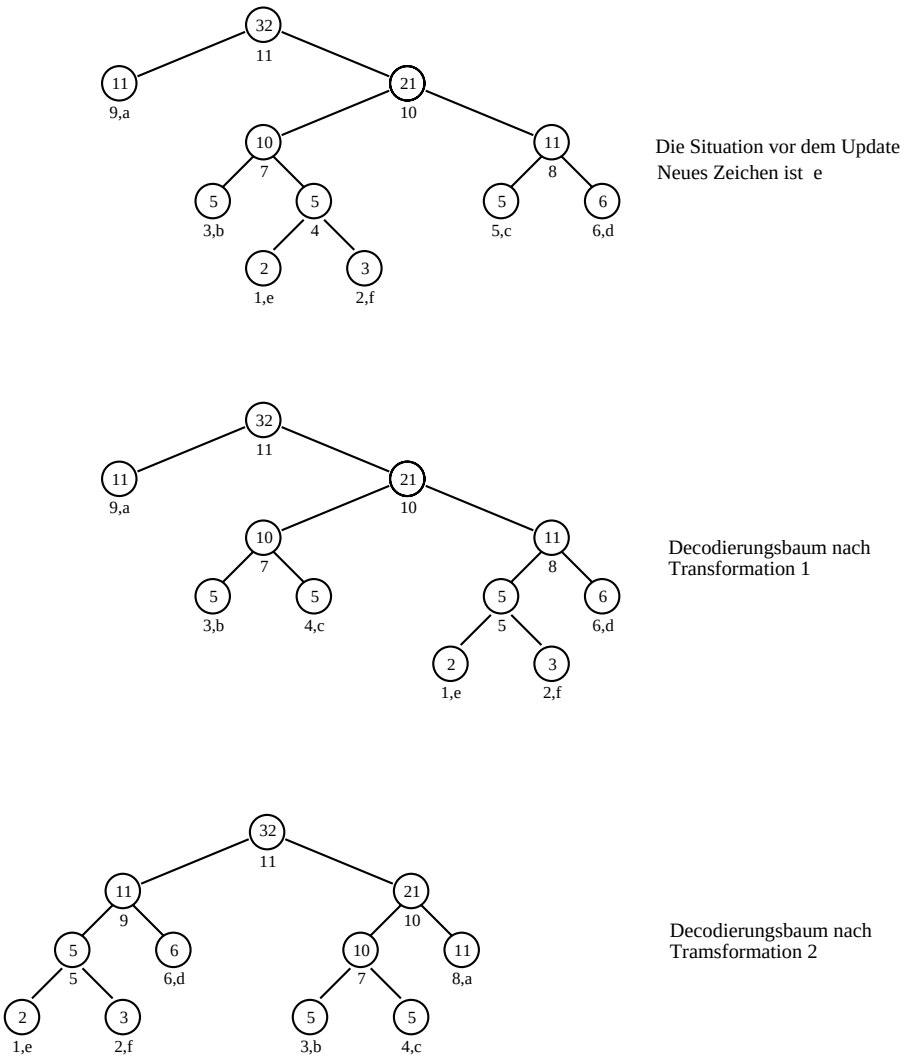


Abbildung 2.6: Update von einem Huffman Code-Baum

des Arrays mit gleichem Zugriffhäufigkeitszählerstand effektiver duchgeführt werden.

- Das Vertauschen von zwei Knoten und ihren Unterbäumen erfolgt durch eine Transposition von 2 Knoten in dem Array und Änderung von konstant vielen Zeigern!

Verlauf des Algorithmus

- Noch nicht codierte Buchstaben werden mit Zugriffhäufigkeitszählerstand 0 durch den Knoten N repräsentiert.
- Soll ein noch nicht codierter Buchstabe codiert werden, so übertragen wir das Codewort für N gefolgt von einer Codierung für den Buchstaben (die nichts mit dem Baum zu tun hat) und fügen ein neues Blatt in den Baum ein (bei

binären Codierungen indem das alte Blatt N zwei Kinder bekommt: das neue N und den neuen Buchstaben).

- Soll ein noch schon mal codierter Buchstabe codiert werden, so codieren wir gemäß der vorliegenden Codierung und verändern anschließend den Baum, falls dieses nötig sein sollte.

Qualität der Codierung

- Jeder Schritt (Codieren eines Buchstaben, Update, Decodieren) kostet $O(l)$ Zeit, wobei l die aktuelle Länge des Huffman Codewortes dieses Buchstaben ist (folgt leicht aus der Konstruktion).
- Es ist bekannt, daß falls eine Nachricht der Länge t mit n Buchstaben ein (statisches) Huffman Codewort der Länge S besitzt, dann gilt für die Länge L des adaptiven Codeworts:

$$S - n + 1 \leq L \leq 2S + t - 4n + 2.$$

Vitter Algorithmus
Verbesserung gegenüber dem letzten Algorithmus:

- Es gilt

$$S - n + 1 \leq L \leq S + t - 2n + 1.$$

- Jeder Schritt kostet weiterhin $O(l)$ Zeit.
- Die Gesamtlänge aller Pfade von den Blättern zur Wurzel und die Tiefe des Decodierungbaumes wird optimiert (die Bäume sind flacher).

2.6 Praktische on-line Methoden

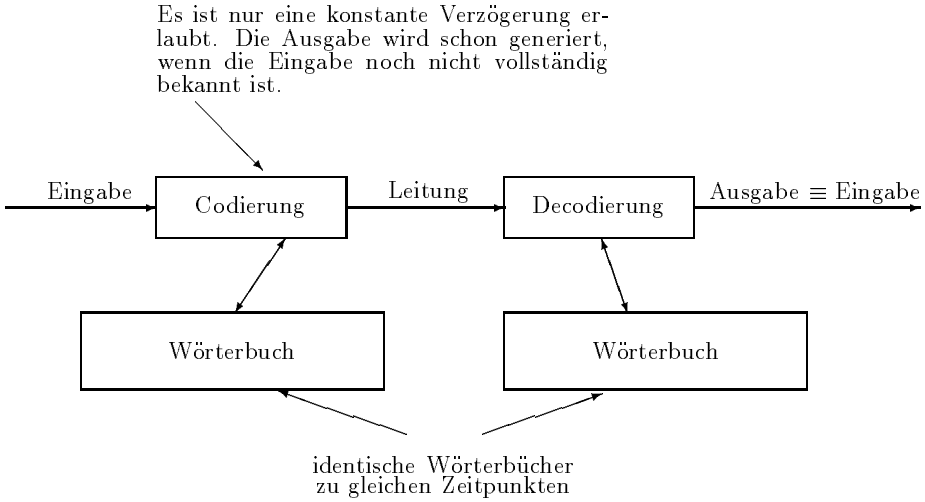


Abbildung 2.7: allgemeine Struktur von on-line Komprimierungsverfahren

Für Anwendungen wie Disk-Treiber und Modem-Kommunikation verwendet man on-line Komprimierungsmethoden die extrem schnell sein müssen. Die Ausgabe muß sofort beim Lesen der Eingabe erzeugt werden. Nur eine kleine Verzögerung zwischen Eingabe und Ausgabe ist erlaubt.

2.6.1 Ziv-Lempel Methode

Ziv und Lempel haben ein allgemeines Schema vorgeschlagen, das dann viele andere Algorithmen übernommen haben. Der Verlauf der Berechnungen ist in Abbildung 2.7 dargestellt. Der Kernpunkt bei Komprimierung und Entkomprimierung ist ein Wörterbuch. Im Wörterbuch stehen einige Wörter und ein extra Eintrag für „unbekannte Folgen".

Die Komprimierung sieht folgendermaßen aus:

1. Ein neuer Block aus Quellzeichen wird gelesen (das Ende des Blocks wird nach irgendwelchen Regeln (match heuristic) gewählt).
2. Der gelesene Block wird im Wörterbuch gesucht.
3. Wurde der Block im Wörterbuch gefunden, so wird der Zeiger zum Eintrag als Ausgabe geschickt. Falls der Block nicht im Wörterbuch steht, so wird der „ist unbekannt" Zeiger und die Eingabefolge (unkomprimiert) geschickt.
4. Bei den meisten Verfahren wird wird zusätzlich noch ein Update des Wörterbuches vorgenommen. Die Änderungen können das folgende umfassen:

   Änderungen der Wahrscheinlichkeiten der einzelnen Blöcke,

   Eintragung von neuen Wörtern ins Wörterbuch,

   Löschen von einigen Wörtern (Begrenzung der Größe des Wörterbuches).

2.6.2 Typen von Wörterbüchern

Die Algorithmen unterscheiden sich in ihrem Umgang mit dem Wörterbuch.

Statische Wörterbücher

Nach der Initialisierung wird das Wörterbuch nicht mehr geändert und für alle Texte benutzt. Statische Wörterbücher

- sind nützlich, falls der erste Teil des Textes typisch für den ganzen Text ist.
- liefern schlechte Ergebnisse, falls zum Beispiel ein Text ein Programm mit einem langen Kommentar am Anfang ist.
- haben die folgende Vorteile:
  - Sie sind wegen ihres kleinen Overheads besonders geeignet für kurze Files.
  - Sie ermöglichen einen schnellen Zugriff auf das Wörterbuch.

Auf Grund der statischen Struktur sind einige Tricks möglich. Als Beispiel betrachten wir ein Wörterbuch mit k-Bit Schlüssel mit 0 am Anfang für die oft vorkommende Blöcke, und $2k$-Bit Schlüssel mit 1 am Anfang für seltener vorkommende Blöcke. Normalerweise verursacht ein Fehler bei der Übertragung den Verlust der Synchronisation und wird die Datei dann nicht korrekt entkomprimiert. Für dieses Wörterbuch ist jedoch automatische Resynchronisation möglich:
Sei h die Wahrscheinlichkeit von 1 auf der ersten Stelle eines k-Bit Blocks ($h \ll 1$).

Lemma 45 *Die Wahrscheinlichkeit, daß nach einem Fehler bei der Übertragung Coder und Decoder während n Blöcken mit k Bits unsynchronisiert bleiben, ist höchstens h^n (in den n Blöcken treten keine Fehler mehr auf).*

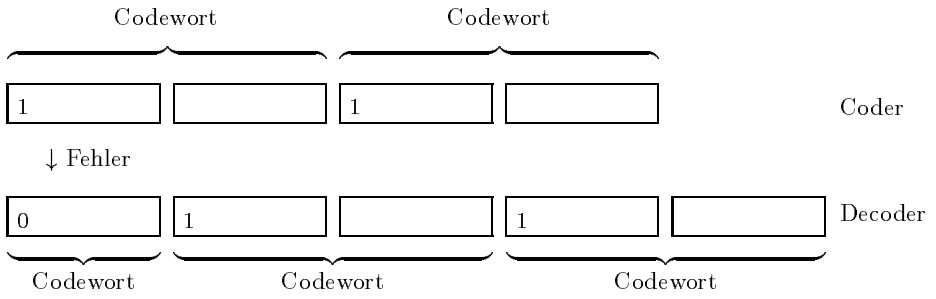


Abbildung 2.8: Unsynchrone Folgen von Coder und Dekoder

Beweis: (siehe auch Abbildung 2.8)
Tritt der Fehler nicht auf der ersten Stelle eines k-Bit oder $2k$-Bit-Codewortes auf, so sind Coder und Decoder immer richtig synchronisiert (auch wenn der Decoder ein Symbol falsch decodiert).

Also muß der Fehler am ersten Bit eines Codewortes aufgetreten sein, damit Coder und Decoder nicht synchron bleiben. Sei nun das übertragene $2k$-Bit Codewort $1\ldots$ und das empfangene (so interpretierte) k-Bit Codewort $0\ldots$. Damit der Versatz von k-Bit auch weiterhin bestehen bleibt, muß der Decoder weiterhin am Anfang von jedem neuen Codewort eine 1 vorfinden, damit er die Codewortlänge als $2k$ interpretiert. Hierdurch ergibt sich die obige Wahrscheinlichkeit. Im anderen Fall, daß das übertragene $2k$-Bit Codewort verfälscht wird, ergibt sich das Ergebnis analog.

□

2.6.3 Sliding dictionary

Die letzten n Eingabezeichen (n ist Parameter) bilden das so gennante Fenster (siehe Abbildung 2.9). Das Wörterbuch umfaßt alle Teilfolgen der Folge aus dem Fenster. Statt dem Block wird ein Zeiger zu der entsprechenden Unterfolge aus dem Fenster ausgegeben.

Beispiel zur Sliding Dictionary

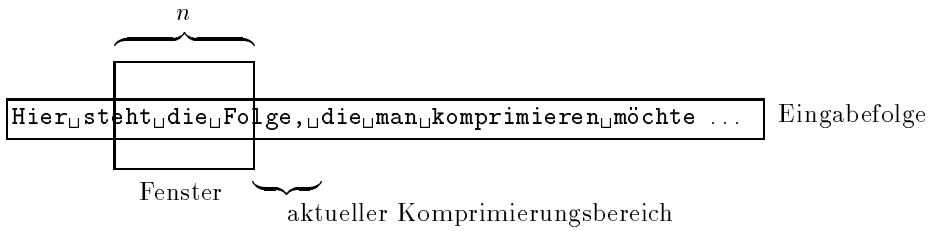


Abbildung 2.9: Ein Fenster von Sliding Dictionary

Eingabefolge `-mama-ma-kota-tata-ma-kota`
Bei Fensterlänge 3 (für richtige Komprimierung viel zu wenig, aber das ist nur ein Beispiel) wäre die Ausgabe:

`-ma[2,2]-[1,3]kota-[1,2][2,2]-m[1,2]kota`

wobei [i,j] die Teilfolge im aktuellen Fenster bezeichnet, die ab Position i beginnt und j Zeichen umfaßt.

Verlauf der Komprimierung (Beispiel)
Nach dem Komprimieren von `ma[2,2]-[1,2][1,1]kota-` ist der Fensterinhalt `ta-`

Schritt 1: `t` wird gelesen und im Fenster gesucht. Gefunden!
Schritt 2: `a` wird gelesen und `ta` im Fenster gesucht. Gefunden!
Schritt 3: `t` wird gelesen und `tat` im Fenster gesucht. Nicht gefunden!

Deswegen bildet `ta` den längsten Match und `ta` wird durch den Zeiger [1,2] in der Ausgabe ersetzt. Das zuletzt gelesene `t` wird bei der nächsten Runde versucht zu komprimieren.

Schwierigkeiten bei der Komprimierung
Bei der Suche nach dem längsten Match kann eine Situaton auftreten, wo viele Teilfolgen des Fensters mit der aktuellen Folge der Eingabebits übereinstimmen. Beispiel: Bei Fensterinhalt `mamak-mat-mamamamu` gibt es drei Teilfolgen in dem Fenster, die mit `mama` übereinstimmen. Wenn wir also als erste vier Zeichen der zu komprimierenden Eingabe die Folge `mama` erhalten, wissen wir nicht, welche von den drei Teilfolgen gebraucht werden, wenn wir das nächste Eingabezeichen bekommen. Deswegen muß man alle solche Teilfolgen im Fenster betrachten (welches aber effizient geschehen kann, siehe Buch von Storer).

Verlauf der Dekomprimierung (Beispiel)

Eingabe	aktuelles Fenster	Ausgabe
`-ma`	irrelevant	`-ma`
[2,2]	`-ma`	`ma`
`-`	irrelevant	`-`
[1,3]	`ma-`	`ma-`
`kota-`	irrelevant	`kota-`
[1,2]	`ta-`	`ta`
[2,2]	`-ta`	`ta`
`-m`	irrelevant	`-m`
[1,2]	`a-m`	`a-`
`kota`	irrelevant	`kota`

Das Sliding Dictionary ist insbesondere in der folgenden Situationen anwendbar:

- für Texte mit hohem Lokalitätsniveau,
- falls aus technischen Gründen eine einfache Implementierung notwendig ist – das Sliding Dictionary läßt sich einfach durch kleine Schaltkreise implementieren.

Das Sliding Dictionary hat den Nachteil, daß einige Teilfolgen im Fenster öfters vorkommen können. Das verringert die Kapazität des Wörterbuchs.

2.6.4 Dynamische Wörterbücher

Dynamische Wörterbucher führen zwei Operationen durch: UPDATE (neue Wörter werden eingetragen) und DELETE (einige Wörter werden gelöscht).

Update Heuristiken

Sei pm der vorherige Block (**p**revious **m**atch) aus der Eingabefolge, und cm der aktuelle Block (current **m**atch).

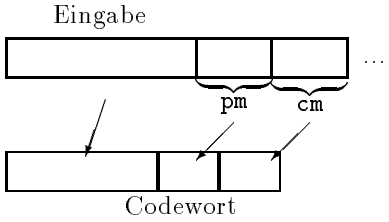


Abbildung 2.10: Die Blöcke pm und cm

In das Wörterbuch werden die Wörter aus der Menge

$$\{\texttt{pm} \circ x : x \in \mathrm{INC}(\texttt{cm})\}$$

neu eingetragen, wobei $\texttt{pm} \circ x$ die Konkatenation von pm mit x und INC(cm) die folgenden Wörter bezeichnet:

Alternative 1 – FC (first character):
INC(cm) enthält nur den ersten Buchstabe von cm,
Motivation: wir nehmen neue Wörter sehr vorsichtig auf, es hilft, wenn z.B. die Wörter viele grammatikalisch unterschiedliche Endungen besitzen können,

Alternative 2 – ID (identity):
INC(cm) enthält nur cm,
Motivation: das ganze Wort wird sofort aufgenommen, weniger Overhead als bei FC,

Alternative 3 – AP (all prefixes):
INC(cm) ist die Menge aller Präfixe von cm,
Motivation: diese Alternative benötigt mehr Speicher, aber durch baumartige Datenstruktur (trie) kann das effizient organisiert werden. Es wird nicht nur cm gespeichert, aber es gibt Möglichkeiten verschiedene Varianten des cm später effizient einzutragen.

DELETE Heuristiken

FREEZE: Wenn das Wörterbuch voll ist, werden keine weiteren Änderungen gemacht (ähnlich zum statischen Fall),

LRU oder *least recently used*: Wenn das Wöterbuch voll ist und ein neuer Eintrag eingetragen werden soll, so wird der als letzte nicht benutzte Wörterbucheintrag gelöscht.

LFU oder *least frequently used*: – zum Löschen wird der Eintrag gewählt, der am seltesten benutzt wurde.
Bei der Implementierung tauchen Probleme auf: ein neues Wort sollte eine fiktive Häufigkeit erhalten, damit es nicht sofort wieder gelöscht wird (und damit die Heuristik zu FREEZE wird). Probleme gibt es hier bei Häufigkeiten, die sich sehr stark unterscheiden.

SWAP: es werden immer 2 Wörterbücher unterstützt: das *primary dictionary* und das *auxiliary dictionary*. Falls das primary dictionary voll wird, werden

die neuen Wörter ins auxiliary dictionary eingetragen. Falls das auxiliary dictionary voll ist, wird das auxiliary dictionary zum primary dictionary und das auxiliary dictionary als leer definiert.

Einige Kombinationen der Heuristiken

FC-FREEZE:	einfache Implementierung, geeignet für kurze Texte (wenig overhead)
FC-LRU:	benötigt mehr Zeit und Speicherplatz, aber bessere Komprimierung; funktioniert gut für lange Texte
AP-LRU:	Zeit und Speicherplatz entsprechen FC-LRU, typischerweise eine etwas bessere Komprimierung
ID-LRU:	aufwendig bei der Implementierung

2.6.5 Zusätzliche Techniken

Es gibt zwei Methoden die Komprimierungeffizienz ohne Zeitverlust zu erhöhen:

- Pipelining: es werden gleichzeitig hintereinander mehrere Verfahren ausgeführt, die Ausgabe einer Unterprozedur ist die Eingabe für die nächste. Dadurch kann man mehrere Verfahren anwenden, die in unterschiedlichen Situationen gut funktionieren.
- Parallele Verarbeitung: man läßt mehrere Verfahren gleichzeitig laufen. Das beste Ergebnis wird dann als Ausgabe gewählt.

2.6.6 Vergleich der Verfahren

Die Kompressionsverfahren sind generell unvergleichbar!

Der Komprimierungsfaktor $= \frac{\text{Größe der komprimierten Datei}}{\text{Größe der Originaldatei}}$ ist abhängig von :

1. der Größe der Datei (komplizierte Wörterbücher verursachen Overhead, für kleine Texte lohnt sich das nicht),
2. dem Grad der Lokalität,
3. dem Typ der Datei:

   Textfiles $\approx$ 30–40% (Unterschiede bei verschiedenen Sprachen),

   Programme mit Text (Programmiersprache) $\approx$ 20–40%,

   „reine Programme“ (nur Code) $\approx$ 20–30%,

   gescannte Bilder $\approx$ 70–80%.

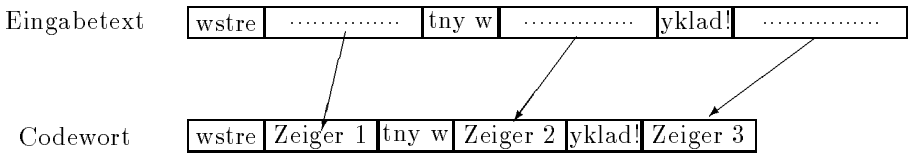
Ranking der Kompressionsverfahren

- Verfahren, die auf Huffman-Codierungen basieren, sind die schlechtesten.
- Das Sliding Dictionary kann nicht wesentlich besser sein, aber ist in einigen Situationen wesentlich schlechter.
- Die besten „dynamischen Wörterbücher“ sind fast immer besser als das Sliding Dictionary, durchschnittlich sind sie deutlich besser als das Sliding Dictionary.
- FREEZE ist „chaotisch“ – kann unerwartet sehr gute oder sehr schlechte Ergebnisse liefern.

- Typisch ist FC-LRU $\leq$ AP-LRU $\leq$ ID-LRU, aber die Unterschiede sind nicht wesentlich (X $\leq$ Y bedeutet, daß Y nie viel schlechter jedoch manchmal wesentlich besser als X ist).

2.6.7 Match-Heuristiken

GREEDY Heuristik ist die erste Wahl. Es wird das längste Wort gesucht, das an der aktuellen Position anfängt und gleichzeitig im Wörterbuch steht.



Eine andere Methode:

- einige Zeichen sollten ohne Kompression übergeben werden
- In einem Block mit fester Länge sucht man mit Hilfe von dynamischer Programmierung die optimale Aufteilung in komprimierfähige Unterblöcke.

Beispiel
Falls der Text gleich `1)␣Betriebsysteme␣sind␣ ...` ist, dann macht es keinen Sinn z.B. `1)␣Be` im Wörterbuch zu suchen, dagegen eher `Betriebsysteme`.

Konstruktion eines optimalen Wörterbuches
Das ist leider schwierig (NP-vollständige Probleme tauchen auf; Details im Kapitel über off-line Komprimierung)

2.6.8 Probleme bei On-line Komprimierung

Für Hardware-Lösungen sind dynamische Wörterbücher nicht gut geeignet. Einfach zu implementieren sind dagegen statische Wörterbücher oder das Sliding Dictionary.

Ein wichtiges Problem ist die Synchronisation, da die codierten und uncodierten Nachrichten verschiedene Anzahlen von Bits haben. Verlust oder Änderung im einen Teil kann dazu führen, daß der Rest nicht korrekt interpretiert wird.

Bei Software-Lösungen ist es sehr wichtig, die Algorithmen effizient zu implementieren. Die Laufzeit ist kritisch bei praktischen Anwendungen. Programme, die sehr gut aber langsam komprimieren, sind nicht besonders nützlich.

2.7 Lossy compression und Komprimierung von Bildern

Ziel
Es soll möglich sein, Dateien je nach Bedarf beliebig zu komprimieren. Die bis jetzt vorgestellten Methoden ermöglichen es, die ursprünglichen Daten vollständig wiederherstellen, aber liefern nur begrenzte Komprimierungsmöglichkeiten.

Eine vollständige Wiederherstellung von Daten ist bei einigen Anwendungen wie Speicherung und Übertragung von *Bildern, Musik*, usw. nicht notwendig (das gilt nicht für Texte oder Programme!).

Für solche Dateien erfolgt die Komprimierung in mehreren Stufen:

Digitalisierung: Kontinuierliche Signale (wie bei einer Videokamera) müssen als Bitfolgen dargestellt werden. Analoge Signale liefern viel zu viel Information.

Lossy Compression: Je nach Kanalkapazität, wird die Anzahl der Bits verkleinert (z.B. durch geringere Auflösung). Durch geschickte Methoden soll, trotz der Komprimierung, der Informationsinhalt der Bilder einigermaßen erhalten bleiben. Diese Stufe kann hierarchisch aufgebaut werden.

Klassiche Komprimierung und fehlertolerate Codierung: Zum Schluß werden normale Methoden (wie bei Texten) angewendet (dazu kann z.B. die Match-Heuristik folgendermaßen verändert werden: wir suchen ein Wort im Wörterbuch, das *fast* mit dem ausgewählten Eingabeblock übereinstimmt). Zum besseren Schutz gegenüber Kanalstörungen kann man eine fehlertolerante Codierung anwenden.

Motivation
Multimedia, interaktives Fernsehen, video-on-demand, ...
Über 90% der Information wird in Zukunft durch Bilder übertragen.

2.7.1 Lossy compression

Lossy Compression beduetet, daß die entkomprimierten Daten nicht identisch zu den ursprünglich komprimierten sein müssen.

Der Sinn von lossy compression besteht darin, die Daten so zu bearbeiten, daß alle wesentlichen Informationen erhalten bleiben aber gleichzeitig die Gröse der Datei verringert wird. Zum Beispiel sollte bei Bildern aus der komprimierten Datei ein Bild zurückgewonnen werden können, das für das menschliche Auge (fast) die gleiche Qualität wie das ursprüngliche Bild hat.

Einige Methoden für lossy compression:

- Vektorquantisierung
- Fraktale Codierung
- Interpolation (einzelne Teile oder Charakteristiken der Bilder können durch einige mathematische Funktionen approximiert werden; statt das Bild zu speichern, kann man die Parameter der approximierenden Funktion speichern).

2.7.2 Vektorquantisierung

Beispiel: Ein Bild mit 256 Graustufen soll auf 16 Graustufen komprimiert werden.

Vorschlag 1: Wir benutzen nur 16 Graustufen. Die Graustufen von $(i-1)\cdot 16+1$ bis $i \cdot 16$ werden als Graustufe $i = 1 \ldots 16$ dargestellt. Nachteil: Falls das Bild vorher nur aus wenigen, nah beieinander liegenden Graustufen bestanden hat, ist der Kontrast u.U. vollständig verloren gegangen, d.h. nach der Komprimierung kann man nichts mehr erkennen (vergleiche Abbildung 2.11).

Vorschlag 2: Man bestimmt zu jedem $j = 1 \ldots 256$, wieviele Pixel mit dieser Graustufe existieren. Danach werden die 256 Graustufen in 16 Intervalle unterteilt, so daß es in jedem Intervall ungefähr gleich viele Pixel gibt. Für jedes Intervall wird dann die mittlere Graustufe berechnet und alle Pixel mit einer Graustufe in dem Intervall mit der mittleren Graustufe dargestellt (vergleiche Abbildung 2.12). Achtung: Overhead in der Datei (also nur ungefähr Komprimierung auf ein viertel), zweimaliger Durchlauf durch die Datei.

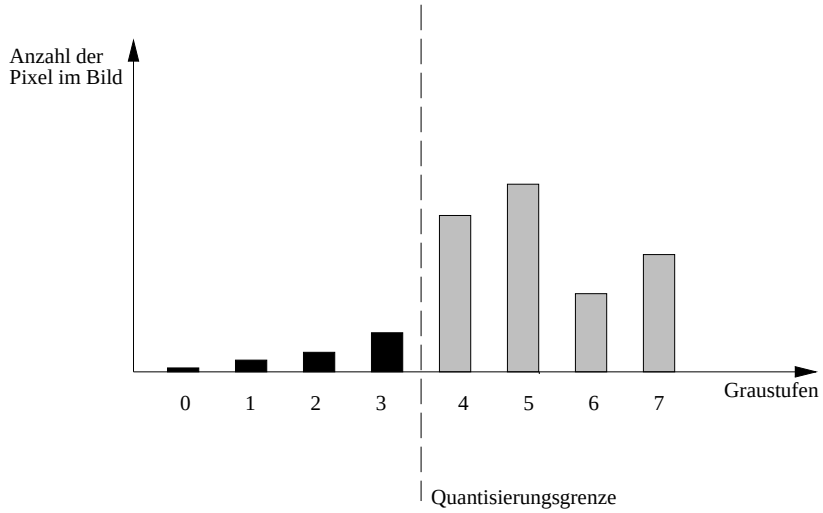


Abbildung 2.11: Reduktion von 8 auf 2 Stufen nach dem Vorschlag 1. Das Bild wird fast total in einer Farbe dargestellt.

Definition: Vektorquantisierung

Pixel (oder Pixelgruppen) werden durch verschiedene Parameter beschrieben. Zum Beispiel wird bei zwei Parametern jeder Pixel durch einen Punkt im 2-dimensionalen Raum beschrieben (vergleiche Abbildung 2.13). Vektorquantisierung beschränkt die Auswahl der Parameter auf nur wenige Punkte. Die Auswahl dieser Punkte ist für eine gute Komprimierung kritisch. Jedes Pixel in dem Bild wird durch ein Pixel ersetzt, dessen Parameter einem der vorgegebenen Punkte entspricht. Man wählt also den vorgegebenen Punkt, der dem Pixel am ähnlichsten ist.

Methoden zur Auswahl der vorgegebenen Punkte sind u.a.:

COVER: Die Punkten sind so gewählt worden, daß die maximale Entfernung von einem beliebigen Punkt zu einem vorgegebenen Punkt höchstens d ist.

LOG: Punkte sind Zeichenketten $a_1, \ldots, a_n$ und die vorgegebenen Punkte sind die Zeichenketten $b_1, \ldots, b_n$ für die $b_1 = \ldots = b_k = 0$, $b_{k+1} \neq 0$ und $b_{k+i} = \ldots = b_n = 0$ ist, für ein frei zu wählendes i (d.h. eine Normierung der Anzahl an signifikanten Stellen auf i). Motivation: z.B. Charakteristik der Empfindlichkeit von menschlichen Ohren oder Augen.

Kohonnen Algorithmus: Die Punkte werden durch einen Lernprozeß mit Hilfe von Neuronalen Netzen erzeugt (Heuristik!).

2.7.3 Fraktale Codierung

Idee Betrachten wir ein Photo von einem Baum. Auf vielen Regionen sind Blätter zu erkennen. Obwohl jedes Blatt anders ist (Größe, Position, Winkel, Farbe), sind sie alle ähnlich.
Jedes Blatt im Bild kann man ungefähr beschreiben durch ein Musterblatt und die Transformation, durch die man aus dem Musterblatt das spezielle Blatt bekommen kann. Die erlaubten Transformationen können zum Beispiel lineare Funktionen (Drehung, Spiegelung, Verschiebung, Skalierung und ihre Hintereinanderausführung) sein.

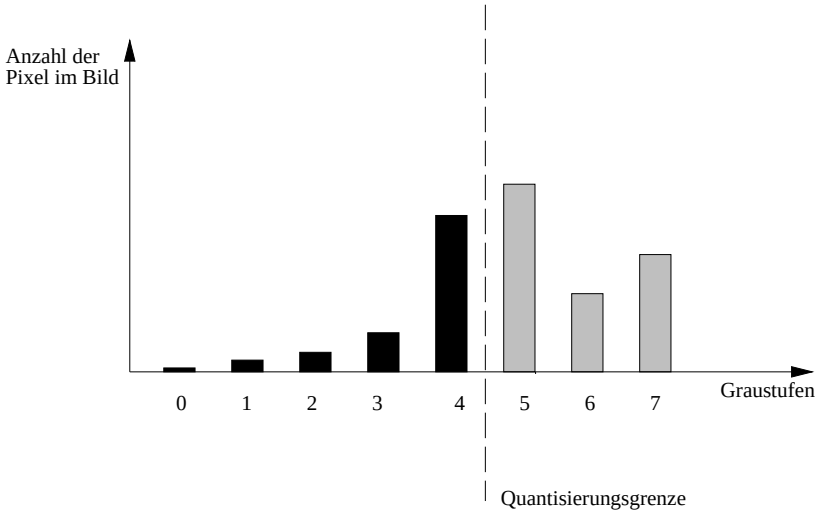
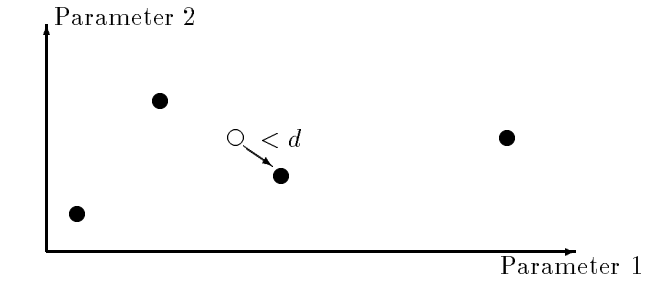


Abbildung 2.12: Reduktion von 8 auf 2 Stufen nach dem Vorschlag 2. Das Gleichgewicht zwischen den verschiedenen Pixeln ist besser.



● - Ausgewählte Punkte für Quantisierung

Abbildung 2.13: Auswahl der Punkten für lossy compression

Form des Fractal Codes

Das Bild wird in kleine $m \times n$ Rechtecke aufgeteilt. Zu jedem Rechteck werden zugewiesen:

- Codierung eines evtl. größeren Muster-Rechtecks,
- Transformation des Muster-Rechtecks (durch einige Parameter).

Um das Rechteck (x, y) zu rekonstruieren nimmt man das Muster für Position (x, y), wandelt es mittels der Transformation um, und kopiert es auf die Position (x, y).

Die Muster sind größer als die Zielrechtecke (z.B. 2-mal) und selbst die Elemente des Bildes (z.B. als $2m \times 2n$ Rechtecke, die um $m/2$ bzw. $n/2$ gegenüber dem Zielrechteck verschoben sind).

Die Rekonstruktion erfolgt stufenweise als Prozeß. Am Anfang nehmen wir ein beliebiges Bild und wenden alle Transformationen an. Das Ergebnis wird als Eingabe für die nächste Wiederholung benutzt.

2.7.4 Einige Techniken für Bilder

Lauflängencodierung: Das Bild wird als eindimensionale Folge von Pixeln codiert. Jede maximale Teilfolge von gleichen Pixeln wird durch das Paar (Pixel-Charakteristik, Teilfolgenlänge) codiert. (Diese Methode kann besonders gut sein, zum Beispiel für Zeichentrickfilme).

Interleaving: Das Bild wird in zwei Bilder untergeteilt. Ein Bild umfasst die geraden das andere die ungerade Zeilen. Die Bilder werden separat übertragen. Eine solche Codierung erlaubt es, die Originalbilder zu erkennen, wenn lokale Störungen auftreten.

Prädiktive Codierung: Codierer und Decodierer enthalten einen Schätzer. Ein Schätzer P versucht das nächste Pixel x_{n+1} an Hand der vorherigen Pixel $x_n, x_{n-1}, \ldots, x_{n-k}$ zu erraten. Dann wird x_{n+1} durch die Differenz zwischen x_{n+1} und $P(x_n, x_{n-1}, \ldots, x_{n-k})$ codiert.

PET-priority encoding transmission: In asynchronen Netzen werden die Bilder nicht als ganzes verschickt sondern auf Pakete aufgeteilt. Einige Pakete erreichen das Ziel schneller als andere, einige können verlorengehen.

Das Bild wird in verschiedene Komponenten aufgeteilt: wichtige Elemente (z.B. einige Konturen, die Positionen wo man anklicken kann, ...) erhalten hohe Priorität, weniger wichtige Elemente erhalten niedrige Priorität und können eventuell nicht zugestellt werden. Das Bild wird nach Prioritäten codiert. Die wichtigsten Elemente sind in fast jedem Paket enthalten. Deswegen kann das Skelett des Bildes mit hoher Wahrscheinlichkeit sehr schnell rekonstruiert werden.

Die Grundlage für PET sind *erasure codes*. Sie erlauben eine Nachricht so auf Pakete aufzuteilen, daß der Verlust von bis zu s Paketen (s ist ein Parameter) die vollständige Wiederherstellung der Datei ermöglicht.

Idee von erasure codes:

- sei $\mathbb{F}$ ein endlicher Körper, w eine Folge von Elementen aus $\mathbb{F}$ (als Polynom interpretiert), sei k der Grad von w, $k < |\mathbb{F}|$.
- Um w zu bestimmen kann man entweder
  - alle Koeffizienten von w angeben, oder
  - $k+1$ Werte $w(x_1), \ldots, w(x_{k+1})$ angeben, wobei $x_1, \ldots, x_{k+1}$ beliebige aber verschiedene Elemente aus F sind.
- um die Koeffizienten von w zu codieren nehmen wir Werte $w(x)$ für $k+1+s$ verschiedene $x \in \mathbb{F}$. Jedes $w(x)$ verschicken wir in einem eigenen Paket. Falls nur bis zu s Pakete verlorengehen, kann der Empfänger das Polynom w rekonstruieren. (Es ist egal welche Pakete überleben!)

Randomisierung: Ein Bild soll von vielen Graustufen auf schwarz-weiß projieziert werden. Mit Vektorquantisierung erhält man zufällige Konturen, die der Wahl der Projektionspunkte entsprechen.

Ausweg: Ein Pixel mit Graustufe i wird ersetzt durch ein Pixel mit Graustufe $\mathrm{Proj}(i + X)$, wobei X eine Zufallsvariable ist und Proj die Abbildung gemäß Vektorquantisierung bezeichnet. Auf diese Weise wird ein Bereich, der in der Mitte zwischen weiß und schwarz klassifiziert werden soll, durch einen Bereich mit gleich vielen schwarzen und weißen Flecken dargestellt.

Oder: Man kann eine Gruppe von Pixeln zusammen betrachten, die durchschnittliche Graustufe berechnen und dann die Gruppe entsprechend dieser Graustufe durch proportional viele weiße und schwarze Pixel darstellen.

Dadurch werden die Graustufen gut simuliert, aber der Kontrast des Bildes verschlechtet sich (Dithering).

...

2.8 Kolmogorov-Komplexität

Der Informationsinhalt einer Nachricht wurde durch die Entropie beschrieben und gilt für Quellen, wo die Symbole unabhängig erzeugt werden. Das ist aber bei off-line Komprimierungsverfahren nicht der Fall: Wir suchen eine sehr gut komprimierte Form einer Datei, die ganz spezifische Eigenschaften haben kann.

Fundamentale Frage: Wie weit läßt sich eine gegebene Datei komprimieren?

Um diese Frage zu beantworten brauchen wir ein Komplexitätsmaß, das die „Komprimierungsfähigkeit" einer Zeichekette beschreibt. Ein solches Maß ist die Kolmogorov Komplexität.

Definition: Kolmogorov-Komplexität

Sei x Zeichenkette und M eine Turingmaschine, die bei Eingabe ϵ die Zeichekette x erzeugt, wobei M unter all den Turingmaschinen, die diese Eigenschaft erfüllen, eine mit kürzestem Programm ist. Dann ist die Länge des Programms von M die **Kolmogorov-Komplexität** von x. Notation: $K(x)$ ist die Kolmogorov-Komplexität von x.
Bemerkung:

- Statt Turingmaschinen könnte man ein anderes Modell benutzen. Die entsprechende Komplexität wäre im wesentlichen äquivalent.
- Es ist egal, wie lange die Turingmaschine braucht, um die Zeichenkette x zu erzeugen. Wichtig ist nur die minimale Größe des Programms, das x erzeugt.

Beispiel: Sei x die binäre Darstellung von $2^{2^{500}}$, womit x aus 2^{500} Bits besteht, also mehr aus mehr Bits als die Anzahl der Atome im Weltraum. Andererseits es ist nicht schwer eine Turingmaschine zu entwerfen, die diese Zeichenkette erzeugt (unter der Annahme, daß die Maschine genug Zeit und Platz dafür hat).

Lemma 46
(1) $K(x) \leq |x| +$ *Konstante*
(2) Für fast alle Zeichenketten der Länge n *gilt* $K(x) \approx |x|$.

Beweis:
(1) die Maschine mit den folgenden Übergängen erzeugt $x_1 x_2 \ldots x_n$:

$$\delta(q_i, z) = (q_{i+1}, x_i, L)$$

für $i = 1 \ldots n$, q_1 =Startzustand, q_{n+1} =Haltezustand.

(2) Es gibt $\leq c^k$ Programme der Länge k und $\leq d^n$ Zeichenketten der Länge n. Für binär codierte Programme, binäre Zeichenketten und k (nicht viel) kleiner als n gibt es $2^n - 2^k \approx 2^n$ Zeichenketten, für die es keine Programme der Länge k gibt. □

Korollar 47 *Fast alle Zeichenketten sind nicht komprimierbar!*

Aber: Menschen und Maschinen erzeugen fast immer komprimierbare Zeichenketten. (es ist sogar technisch schwierig die Zeichenketten zu erzeugen, die hohe Kolmogorov-Komplexität haben).

2.9 Off-line Zeiger-Methoden

Fast alle Komprimierungsmethoden arbeiten nach dem folgenden Prinzip:
Man ersetzt eine Teilfolge in der Datei durch einen Zeiger auf die gleiche Teilfolge, die irgendwo anders steht (im Wörterbuch, woanders in der Datei, ...). Bei Dekomprimierung wird das Prozeß rückgängig gemacht.
Solche Methoden werden wir **Zeiger-Komprimierung** nennen.

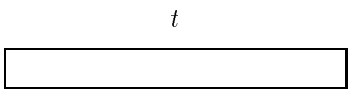
Problem: die Reihenfolge, in der wir die Zeiger durch Zielfolgen bei Dekomprimierung ersetzen, kann sehr wichtig sein.
Beispiel: Komprimierte Folge ist $\texttt{aa}_{[1,2][1,3]}$.
Berechnung 1: $\texttt{aa}_{[1,2][1,3]} \longrightarrow \texttt{aaaa}_{[1,3]} \longrightarrow \texttt{aaaaaaa}$
Berechnung 2: $\texttt{aa}_{[1,2][1,3]} \longrightarrow \texttt{aa}_{[1,2]}\texttt{aa}_{[1,2]} \longrightarrow \texttt{aaaaaa}_{[1,2]} \longrightarrow \texttt{aaaaaaaa}$
Das Problem liegt daran, daß die Zielfolgen selbst Zeiger enthalten können. Solche „rekursiven“ Zeiger sollten erlaubt sein, um eine bessere Komprimierung zu ermöglichen.

Jede komprimierte Datei enthält das **Wörterbuch** und das **Skelett**. Sie müssen nicht getrennt sein.

Skelett Ist eine Grundstruktur der komprimierten Datei. Es enthält unkomprimierte Teile und Zeiger zum Wörtebuch. Wenn die Zeiger durch Zielfolgen ersetzt werden, gewinnt man die ursprüngliche Datei zurück.

Wörterbuch Enthält alle Zielfolgen für die Zeiger aus dem Skelett. Das Wörterbuch kann selbst komprimiert sein.

Typen von Zeiger-Komprimierung:



Skelett und Wörterbuch

Abbildung 2.14: Internes Schema der Off-Line Komprimierung

- **interne versus externe Methoden:** bei einer externen Methode sind Wörterbuch und Skelett (z.B. durch das #-Zeichen) getrennt, bei einer internen Methode sind sie identisch.

  Beispiel: Der String $w = \texttt{aaBccDaacEaccFaaccGacac}$ könnte bei der externen Methode als $x = \texttt{aacc\#}_{[1,2]}\texttt{B}_{[3,2]}\texttt{D}_{[1,3]}\texttt{E}_{[2,3]}\texttt{F}_{[1,4]}\texttt{G}_{[2,2][2,2]}$ und bei der internen Methode als $y = \texttt{aaBccD}_{[1,2]}\texttt{cE}_{[7,2]}\texttt{F}_{[7,2]}\texttt{cGac}_{[15,2]}$ komprimiert werden.

- **rekursive Zeiger:** die Zielfolge, auf die ein Zeiger deutet, kann andere Zeiger enthalten.

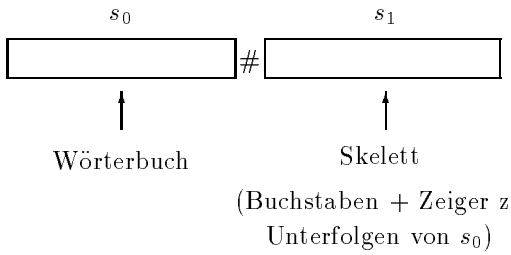


Abbildung 2.15: Externes Schema der Off-Line Komprimierung

- **Originalzeiger versus Wörterbuch-Zeiger:** Originalzeiger deuten auf Positionen in der unkomprimierten Datei. (Wörterbuch-Zeiger deuten hingegen (wie der Name schon sagt) auf Positionen im ursprünglichen Wörterbuch.) Beispiel für einen Originalzeiger: a[1,k] deutet auf eine Folge von $k+1$ Zeichen. [1,k] deutet dabei auf eine Teilfolge der Orginalfolge ab Position 1 bis Position k. Es ist leicht zu sehen, daß damit a[1,k] eindeutig die Folge a^{k+1} codiert.
  Mit diesem Verfahren würde der String w z.b. wie folgt komprimiert:
  $z =$ aaBccD[1,2]cEa[4,2]F[1,2][4,2]Gac[16,2].

In den obigen Beispielen wurden Zeiger nur auf vorherige Positionen gelegt. Hebt man diese Restriktion auf, so kann z.B. an Stelle von z der folgende komprimierte String erzeugt werden: [15,2]B[17,2]D[15,3]E[16,3]FaaccG[16,2][16,2].

Hierdurch können sich zyklische Verweise ergeben, wobei der komprimierte String jedoch *eindeutig* dekomprimierbar bleibt.
Beispiel: ab[5,2] a[1,3] kann eindeutig als abaaaaba dekomprimiert werden, jedoch verweist [5,2] auf die ersten beiden Zeichen von a[1,3] und [1,3] auf die ersten drei Zeichen von ab[5,2]. Treten solche Zyklen bei einer Komprimierung mit Originalzeigern nicht auf, so wird die Komprimierung **topologisch** genannt.

2.9.1 Schemata für Off-Line Komprimierung

- EPM (External Pointer Macro): Die komprimierte Datei t hat die Form $s_0 \# s_1$, wobei s_0 und s_1 Buchstaben und Zeiger zu Teilfolgen von s_0 enthalten.

  Dekomprimierung:
  while s_1 enthält Zeiger do
  ersetze Zeiger in s_1 durch die Zielfolge des Zeigers aus s_0

  Beispiel: Die Folge aa[1,2][1,3]#[1,4]b[1,2] wird wie folgt dekomprimiert:
  [1,4]b[1,2] → aa[1,2][1,3]b[1,2] → aaaa[1,3]baa → aaaaaa[1,2]baa → aaaaaaaabaa

- CPM (Compressed Pointer Macro): Die komprimierte Datei t enthält kein separates Wörterbuch, sondern nur Buchstaben und Zeiger auf t.

  Dekomprimierung: erfolgt wie EPM auf $t\#t$.

  Beispiel: Die Folge aa[1,2][1,3] wird wie folgt dekomprimiert:
  aa[1,2][1,3] → aaaa[1,3] → aaaaaa[1,2] → aaaaaaaa

- OPM (Original Pointer Macro): Sei t eine komprimierte Version der Originaldatei s. Dann enthält t Buchstaben und Zeiger auf s.

  Dekomprimierung:
  (1) Jeder Zeiger [n,m] wird durch die Zeiger [n,1][n+1,1] . . . [n+m-1,1] ersetzt. Dann

steht jeder Zeiger für einen Buchstaben.
(2) Auf das erhaltene Wort wird CPM angewendet,

Beispiel: Die Folge ab[5,2]a[1,3] wird wie folgt dekomprimiert:
(1) ab[5,2]a[1,3] → ab[5,1][6,1]a[1,1][2,1][3,1]
(2) Dann mit CPM: ab[5,1][6,1]a[1,1][2,1][3,1] →aba[1,1]aab[5,1] →abaaaaba

- OEPM (Original External Pointer Macro): Die komprimierte Datei t hat die Form $s_0\#s_1$.

  Dekomprimierung:

  1. s_0 wird durch OPM dekomprimiert. Sei r das Ergebnis.
  2. In s_1 wird jeder Zeiger durch die Zielfolge aus r ersetzt

  Beispiel: Die Folge a[1,4]#[1,5]b wird wie folgt dekomprimiert:
  a[1,4]#[1,5]b $\overset{\text{OPM}}{\rightarrow}$ aaaaa#[1,5]b → aaaaab

2.9.2 Codelänge bei off-line Komprimierungs-Schemata

Notation

Wir nehmen an, daß jeder Zeiger eine feste Anzahl von p Bits zur Darstellung benötigt. Sei $\Delta_R(s)$ die kürzeste Folge, die nach Schema R die Folge s darstellt, und $|\Delta_R(s)|$ die Länge von $\Delta_R(s)$.

Lemma 48 $|\Delta_R(s)| \geq |\Delta_R(\mathtt{a}^{|s|})|$ *für beliebige Folge* s*, Zeichen* a *aus Quellalphabet und Schema* R*.*

Beweis: Werden alle Buchstaben in s durch a ersetzt, dann stimmen die Zeiger auch weiterhin. Also erhält man aus einer komprimierten Version der Folge s eine komprimierte Version der Folge $\mathtt{a}^{|s|}$, die die gleiche Länge hat. □

Beispiele:

1. Wir nehmen an, daß keine Zeiger sich überlappen dürfen (zwei Zeiger [x,m] ≠ [y,n] überlappen, wenn sich die Intervalle (über den reelen Zahlen) $[x, x+m-1]$ und $[y, y+n-1]$ überlappen). Dann ist die beste Komprimierung für $\mathtt{a}^n$: $\mathtt{aaa}q_0q_0q_0q_1q_1q_1q_2q_2q_2\ldots q_Lq_Lq_Lq_{L+1}\#a^{r_0}q_0^{r_1}\ldots q_{L+1}^{r_L+2}$, wobei q_{i+1} ein Zeiger auf $q_iq_iq_i$ ist, L entsprechend gewählt ist und die r_j eine ternäre Darstellung von n bilden.

   Beispiel:
   Für $\mathtt{a}^{11}$: aaa[1,3][1,3][1,3]#aa[4,3]
   Für $\mathtt{a}^{24}$: aaa[1,3][1,3][1,3]#[1,3][1,3][4,3][4,3]

   Beweisidee: Die komprimierte Folge stellt eigentlich einen Baum dar, bei dem jeder Knoten k Söhne hat. Bei $\mathtt{a}^m$ wird die Folge vor dem Trennsymbol # durch ungefähr $k\log_k m = \frac{k}{\ln k}\ln m$ Zeiger gebildet. Ableitung nach k ergibt, daß an der Stelle $k_0 = e = 2.71\ldots$ die einzige Extremstelle liegt. Da $\frac{2}{\ln 2} = 2.88\ldots$, $\frac{e}{\ln e} = 2.71\ldots$ und $\frac{3}{\ln 3} = 2.73\ldots$ ist, folgt, daß die Stelle e **das** Minimum ist und daß die Stelle 3 das ganzzahlige Minimum ist (da die Funktion vor der Stelle e streng monoton fällt und nach der Stelle e streng monoton wächst).

2. Falls keine Rekursion und Überlappung erlaubt ist, dann ist die beste Komprimierung von $\mathtt{a}^n$: $\mathtt{a}^t\underbrace{[1,t]\ldots[1,t]}_{n/t-1}$, wobei $t = \sqrt{p\cdot n}$ ist.

   Beweisidee: Das komprimierte Wort enthält w, eine Teilfolge des zu komprimierenden Wortes. Zusätzlich braucht man $\frac{n}{|w|}$ Zeiger um einen String

der Länge n zu codieren. Der komprimierte Wort enthält dann $|w| + \frac{n}{|w|} \cdot p$ Buchstaben. Der minimaler Wert wird für $|w| = \sqrt{p \cdot n}$ erreicht.

Theorem 49
$|\Delta_{OPM}(s)| \leq |\Delta_{CPM}(s)|$. *Dabei kann* $|\Delta_{OPM}(s)|$ *wesenlich kleiner sein.*

Beweis:

- Jeder Zeiger entspricht bei CPM letztendlich einer Teilfolge der Originalfolge. Diesen Zeiger kann man durch einen Originalzeiger auf diese Teilfolge ersetzen. Beispiel: CPM-Codewort `aa`[1,2][1,3] entspricht dem String `aaaaaaaa`. Der Zeiger [1,2] verweist auf die Zielfolge `aa` (Positionen von 1 bis 2) und [1,3] hat letztendlich die Zielfolge `aaaa` (die Positionen von 1 bis 4). Das OPM-Codewort kann dann folgendermaßen aussehen: `aa`[1,2][1,4].

  Damit ist der erste Teil des Theorems bewiesen.

- $\Delta_{OPM}(\mathtt{a}^n) = \mathtt{a}$[1,n-1]. Andererseits kann jedes CPM-Codewort mit konstant vielen Zeiger nur einen String von konstanter Länge erzeugen (wenn es keine Zyklen gibt. Existieren Zyklen, so wird ein unendlich langer String erzeugt). Also wächst $|\Delta_{CPM}(\mathtt{a}^n)|$ unbeschränkt ($|\Delta_{CPM}(a^n)| = \Omega(p \cdot \log(n))$).

  Damit ist der zweite Teil des Theorems gezeigt. □

Theorem 50 *Für jede Folge s gilt :*

1. $\frac{2}{3}|\Delta_{CPM}(s)| < |\Delta_{EPM}(s)| \leq |\Delta_{CPM}(s)| + p + 1$
2. $\frac{1}{2}|\Delta_{OPM}(s)| < |\Delta_{OEPM}(s)| \leq |\Delta_{OPM}(s)| + p + 1$

(p ist die Zeigergröße in Bits).

Beweisidee:

- die rechten Ungleichungen sind trivial, z.B.: $\Delta_{CPM}(s)$ # [1,$|\Delta_{CPM}(s)|$] ist ein EPM-Codewort von s der Länge $|\Delta_{CPM}(s)| + p + 1$
- Aus einem EPM-Codewort $s_0 \# s_1$ macht man CPM-Codewort, indem die Folge s_0 wie folgt in s_1 eingebaut wird:
  - Entferne aus s_0 alle Zeichen, auf die nicht durch einen Zeiger von s_0 oder s_1 verwiesen wird, und modifiziere die anderen Zeiger entsprechend.

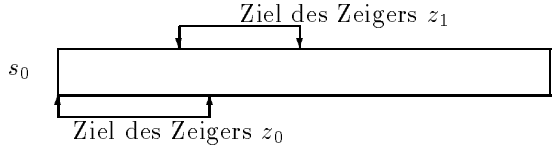


Abbildung 2.16: Umwandeln eines EPM-Codes in einen CPM-Code

  - Bestimme eine Folge $z_0, z_1, \ldots$ von Zeigern aus s_1, so daß:
    - die Zielfolge von z_0 der längste Präfix von s_0, auf die ein Zeiger von s_1 verweist, ist.
    - die Zielfolge von z_{i+1} der längste Präfix von s_0, auf die noch kein Zeiger $z_0, \ldots, z_i$ verweist (jedoch ein Zeiger von s_1), ist.

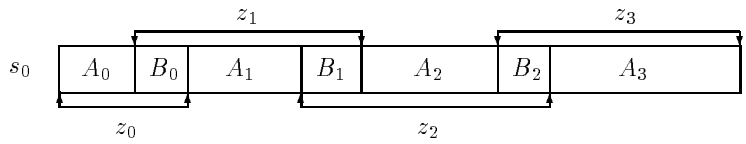


Abbildung 2.17: Umwandeln eines EPM-Codes in einen CPM-Code

– Bezeichnet A_i die Teilfolge von s_0, auf den nur z_i (jedoch nicht die anderen z_j) zeigt, und B_i die Teilfolge, auf die z_i und z_{i+1} gemeinsam verweisen, so wird in s_1 genau ein Zeiger z_i durch die Teilfolge A_iB_i ersetzt.

Durch die Wahl der Zeiger $z_0, z_1, \ldots$ ist es möglich jeden noch in s_1 vorhandenen Zeiger auf eine Teilfolge in s_0 durch höchstens 2 Zeiger zu ersetzen, da für keinen Zeiger p, dessen Zielfolge mit z_i überlappt, das rechte Ende der Zielfolge von p weiter rechts liegen kann als bei z_{i+1} (sonst hätte man p als z_{i+1} gewählt).

Es ist einfach zu überprüfen, daß das erhaltene CPM-Codewort die richtige Länge besitzt.

2.9.3 Komplexitätsfragen für off-line Komprimierung

Problem: Gegeben sei eine Datei x. Finde das kürzeste Codewort für x, z.B. mit CPM-Schema.

Leider läßt sich das Problem im allgemeinen nicht effizient lösen. Die Fragen nach einem optimalen Wörterbüch gehört zu der Klasse von NP-schweren Fragen.

Theorem 51 *Folgendes Problem ist NP-vollständig. Eingabe: Folge s, Zahl K.*

$$\textit{Frage: } |\Delta_{CPM}(s)| \leq K \ ?$$

(bzw. $|\Delta_{EPM}(s)| \leq K$?, $|\Delta_{OPM}(s)| \leq K$?, $|\Delta_{OEPM}(s)| \leq K$?)

Andererseits kann man bei fest vorgegebenem Wörterbuch das kürzeste EPM-Codewort durch dynamische Programmierung finden.

Beweisidee: Wir zeigen eine Reduktion unter den zusätzlichen Annahmen, daß die Zeiger nicht rekursiv sind und daß die Zielfolgen der Zeiger sich nicht überlappen (das ist eine sehr beschränkte Komprimierung, für die die Suche nach dem optimalen Wörterbuch nicht schwieriger sein sollte).

Wir reduzieren VERTEX COVER auf unser Wörterbuch-Problem. Eine Eingabe für VERTEX COVER ist ein Graph und eine Zahl k. Gefragt wird, ob es eine Menge von k Knoten gibt, so daß jede Kante mindestens ein Ende aus dieser Menge hat. Es ist bekannt, daß VERTEX COVER NP-vollständig ist. Dadurch wäre auch unser Problem NP-vollständig.

Sei $G = (V, E)$ ein Graph, $V = \{v_1, \ldots, v_n\}$, $E = \{e_1, \ldots, e_m\}$. Für alle Knoten v_i und Kanten $e_i = (v_s, v_t)$ von G definieren wir die folgende Wörter:

$$\vec{V}_i = \$ \underbrace{v_i \ldots v_i}_{p-1} \$ \qquad\qquad \vec{E}_i = \$ \underbrace{v_s \ldots v_s}_{p-1} \$ \underbrace{v_t \ldots v_t}_{p-1} \$$$

wobei p die Anzahl der Buchstaben ist, die einen Zeiger codieren (Annahmne: alle Zeiger brauchen genau p Buchstaben). Dann kann man den Graphen G durch folgendes Wort s_G repräsentieren:

$$\underbrace{\vec{V}_1@ \ldots \vec{V}_1@}_{p} \ldots \underbrace{\vec{V}_n@ \ldots \vec{V}_n@}_{p} \vec{E}_1@ \ldots \vec{E}_m@,$$

wobei die $pn + m$ vielen @-Zeichen paarweise *verschiedene* Symbole des Arbeitalphabets sind!

Behauptung:
G hat Vertex-Cover der Größe k genau dann, wenn $|\Delta_{CPM}(s_G)| \leq |s_G| + k - m$.

($\Rightarrow$) Sei X ein Vertex-Cover für G. Wir entwerfen ein CPM-Codewort für s_G: $s_0 \# s'$ wobei s' am Anfang s_G ist (d.h. wir ersetzten stufenweise einige Teilfolgen aus s_G durch Zeiger, letztendlich kriegen wir fertiges s') und

- s_0 die Wörter $\vec{V}_i$ für $v_i \in X$ enthält. Dafür benötigen wir insgesamt $\leq k(p+1)$ Buchstaben.
- bei s' wird im jedem $\vec{E}_j$ **ein** $\vec{V}_l$ durch einen Zeiger ersetzt. Dadurch sparen wir pro Kante genau einen Buchstaben, insgesamt also m Buchstaben.
- für $v_i \in X$ werden auch die $\vec{V}_i$ im ersten Teil von s_G durch die Zeiger ersetzt. Für jedes $v_i \in X$ sparen wir dadurch $p \cdot 1 = p$ Buchstaben. Insgesamt also $\leq kp$ Buchstaben.
- Die Gesamtfolge hat damit die Länge $\leq k(p+1) + 1 + |s_G| - m - kp = kp + k + 1 + |s_G| - m - kp = |s_G| + k + 1 - m$.

($\Leftarrow$) Jede komprimierte Version von s hat die folgende Eigenschaften:

(i) Die verschiedenen @-Zeichen gewährleisten, daß es keinen Sinn macht, diese in s_0 aufzunehmen, da sie nur genau einmal auftreten.

(ii) Zeiger, die auf mehr als die Hälfte von einem $\vec{E}_j$ verweisen, können nur einmal benutzt werden, da Kanten sich nicht wiederholen. Also macht es keinen Sinn, solche zu erstellen.

(iii) Verwendet man an Stelle $\vec{V}_j$ einen Zeiger, so läßt sich genau einen Buchstaben sparen.

Aus (i), (ii) und (iii) kann man herleiten, daß alle CPM-Codeworte die sinvoll sind, die gleiche Form wie aus Teil ($\Rightarrow$) haben. □

2.10 Superstring Problem

Definition: Superstring Problem

Gegeben: Worte $x_1, x_2, \ldots, x_n$.
Gesucht wird eine Folge x, so daß für jedes i die Folge x_i eine Unterfolge von x ist. So ein x heißt **Superstring** und bildet ein Wörterbuch für alle Wörter $x_1, x_2, \ldots, x_n$

Theorem 52 *Die Frage, ob für gegebene Wörter $x_1, x_2, \ldots, x_n$ und $k \in \mathbb{N}$ ein Superstring der Größe $\leq k$ existiert, ist NP-vollständig.*

Beweisidee: Wir zeigen, daß das Superstring Problem sehr stark mit dem TSP (Travelling Salesman Problem) verwandt ist.

Sei $x_1, \ldots, x_n$ und k eine Eingabe. Dann definieren wir einen gerichteten *Überlappungs-Graphen*:

- Die Knoten sind die Strings x_i.
- Für je zwei Knoten x_i, x_j definieren wir die gerichtete Kante (x_i, x_j) mit dem Gewicht Überlapp(x_i, x_j), wobei Überlapp(x_i, x_j) die Länge des längsten Suffix von x_i ist, der ein Prefix von x_j ist.

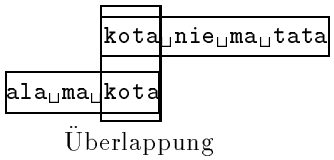


Abbildung 2.18: Überlappung von zwei Strings

Annahme 1: Kein x_i ist ein Teilstring von einem anderen x_j (so ein x_i kann offensichtlich aus dem Problemstellung eliminiert werden).
Annahme 2: Der Superstring fängt mit x_{i_1} und endet mit x_{i_2}. (Es gibt nur n^2 Möglichkeiten, die alle getest werden können.)

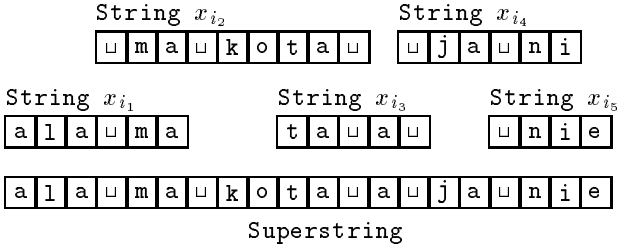


Abbildung 2.19: Eine Lösung fur Superstring Problem

Unter den Annahmen sieht eine Lösung wie in Abbildung 2.19 aus und entspricht einem hamiltonischen Weg in dem Überlappungs-Graphen. Die Länge des Weges ist die Summe von Überlappungen. Die Länge des Superstrings ist $\sum_i |x_i|$−(die Länge des Weges). □

2.10.1 Approximationsalgorithmen für Superstrings

Das Superstring Problem kann nicht optimal in polynomieller Zeit gelöst werden (unter der Annahme, daß P≠ NP!), aber man kann die optimale Lösung *approximieren*, daß heißt eine Lösung mit beweisbarer Qualität in vertretbarer Zeit finden.

Greedy-Algorithmus

Phase 1 : finde 2 Strings mit der größten Überlappung. Klebe sie so überlappend zusammen. Dadurch wird die Anzahl der Strings um 1 verkleinert.

Phase 2 : GOTO Phase 1

Am Ende erhalten wir ein Problem mit genau einem String. Das ist der produzierte Superstring.

Qualität der Lösung:

- Es wurde bewiesen, daß die Länge des Superstrings, den der Greedy Algorithmus produziert, höchstens vier mal so lang wie die optimale Lösung ist.
- Die Einsparungen durch Überlappungen sind mindestens 50% des möglichen.

Andererseits:

- Es gibt Beispiele, wo die Lösung des Greedy Algorithmus doppelt so lang wie die optimale Lösung ist.
- Es gibt eine Konstante $\delta > 1$, so daß kein polynomieller Algorithmus die Länge $\leq \delta \cdot$(die optimale Länge) immer erreichen kann (Annahme: P≠ NP).

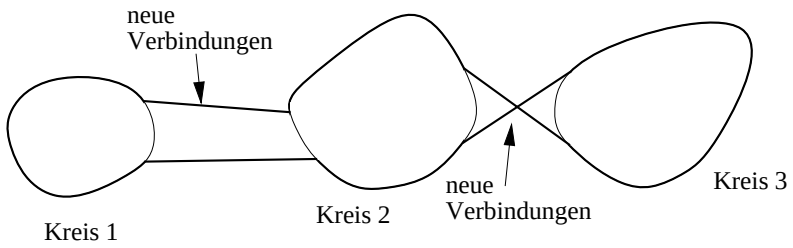


Abbildung 2.20: Bilden von Wegen aus Kreisen

Andere Algorithmen

Man kann Konstanten < 4 erreichen. Idee der Algorithmen:

- finde eine Überdeckung der Knoten des Überlappungsgraphen mit Zyklen, so daß das Gesamtgewicht der Kanten in den Zyklen maximal wird. Die Berechnung von einer solchen Überdeckung ist in polynomieller Zeit möglich (maximales Matching in dem bipartiten Graphen $H = (V_1 \uplus V_2, V_1 \times V_2)$, mit $V_1 = V_2 = V$ und Kanten $(v_i, v_j) \in V_1 \times V_2$ mit den ursprünglichen Gewichten aus G).
- Verbinde die Zyklen in neue Zyklen (siehe Abbildung 2.20). Dafür werden die leichtesten Verbindungen zwischen den Kreisen benutzt.

Kapitel 3

Kryptologische Codes

3.1 Einleitung

3.1.1 Zwecke der Codierung

Klartext und Kryptogramm

Aus der ursprünglichen Datei, die man **Klartext** nennt, erzeugt man eine codierte Version, die man **Kryptogramm** nennt. Bei der Codierung und Decodierung sind zusätzlich **Schlüssel** notwendig. Es gibt:

symmetrische Verfahren: der Schlüssel zum Codieren und der Schlüssel zum Decodieren sind gleich (oder leicht voneinander berechenbar).

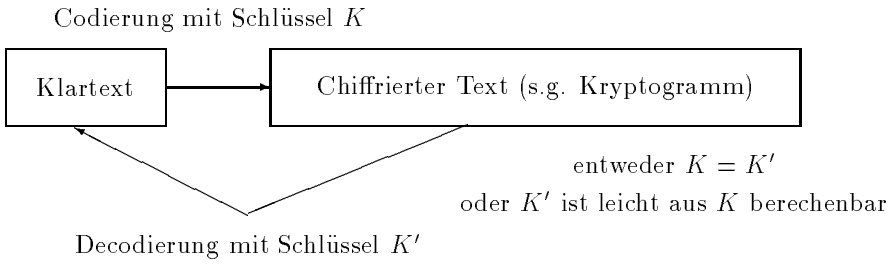


Abbildung 3.1: Chiffrieren mit symmetrischen Verfahren

asymmetrische Verfahren: (public key Verfahren) die Schlüssel zum Codieren und Decodieren sinde unterschiedlich (also werden die Schlüssel immer paarweise betrachtet); es soll praktisch unmöglich sein den einen Schlüssel von dem anderen herzuleiten.

Grundsätzlich soll das Decodieren mit einem falschen Schlüssel **keine** Informationen über den Klartext liefern.

Zwecke der Codierung – Beispiele

Geheimhaltung der Dateien: mit symmetrischen Verfahren.
Anwendungsgebiete:

- Sicherung der Dateien (sie werden nur in der verschlüsselten Form gespeichert). Nur der Besitzer des Schlüssels kann sie lesen.

- Sicherung der Kommunikation durch unsichere Leitungen (die man abhören kann). Zum Beispiel Austausch von Dateien zwischen zwei Filialen einer Firma über Telefon.

Authentifikation von Nachrichten: mit asymmetrischen Verfahren.

Man veröffentlicht einen von 2 passenden Schlüsseln (man nennt ihn *public key*). und hält den zweiten Schlüssel geheim (*private key*). Die Authentizität einer eigener Nachricht wird durch Codierung mit dem private key bestätigt, denn es gilt:

- Jeder kann mit dem public key eine lesbare Nachricht wiedergewinnen.
- Kein Kryptogramm, das mit einem anderen private key erzeugt wurde, läßt sich mit dem public key in eine lesbare Nachricht decodieren ($\Rightarrow$ niemand außer dem Besitzer des private key kann die Nachrichten auf diese Weise codieren).
- Die Entschlüsselung mit einem Schlüssel ungleich dem public key ergibt eine chaotische Folge von Buchstaben ($\Rightarrow$ man kann den Autor nicht verwechseln).

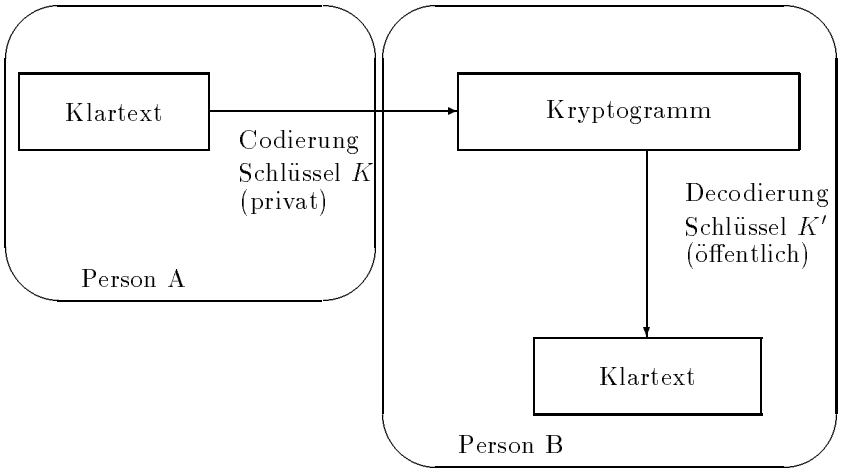


Abbildung 3.2: Authentifikation durch Chiffrieren

Geheimhaltung elektronischer Korrespondenz: mit public key Verfahren.

Jeder veröffentlicht seinen public key (z.B. durch persönliche WWW home page). Wenn man einen privaten Brief an Person C schicken will, dann

- holt man den public key K von C,
- verschlüsselt den Brief mit dem Schlüssel K,
- verschickt den codierten Brief zu C,
- entschlüsselt C den Brief mit seinem private key und kann ihn lesen.

Keine Vereinbarung der Schlüssel zwischen Absender und Empfänger ist notwendig. Keiner außer dem Empfänger kann den Brief lesen.

Elektronischer Notar:

Zwei der wichtigsten Anwendungen:

1. Nachweis, daß man im Besitz von einem Text ist ohne den Text selbst präsentieren zu müssen,

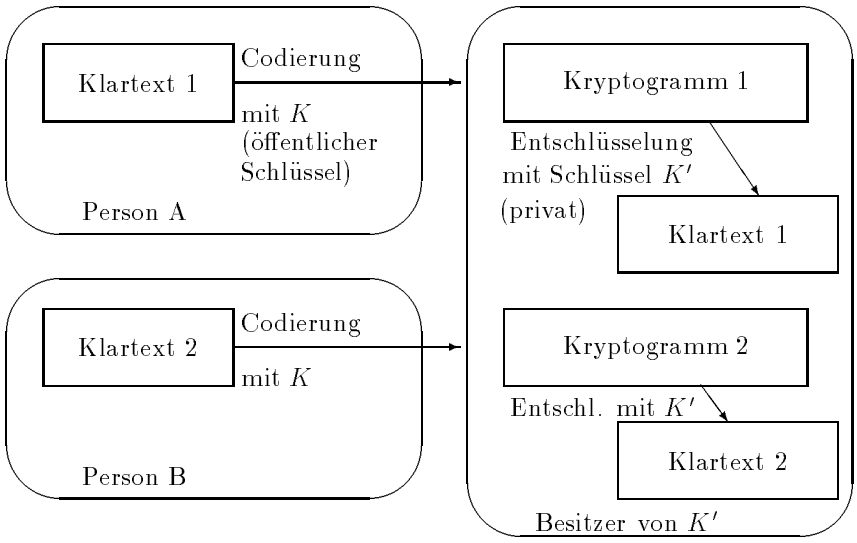


Abbildung 3.3: Verschicken von chiffrierten Briefen mit public key Verfahren

2. Schutz vor Änderungen in Dateien. Für jede Datei kann man einen kurzen Fingerprint berechnen, der praktisch eindeutig die Datei identifiziert (sogar ein extra Blank ändert den Fingerprint).

Abbildung 3.4: Einweg-Hashfunktionen

Dafür benötigt man *Einweg-Hashfuntionen*. Notwendige Eigenschaften einer Einweg-Hashfunktion F:

- Für jedes X ist $F(X)$ einfach zu berechnen.
- $F(X)$ hat eine feste Länge für alle X (damit die Länge der Originaldatei nicht verraten wird, und man die Fingerprints in der Zeitung als private Anzeige veröffentlichen kann).
- $F^{-1}(Y)$ ist praktisch nicht berechenbar (auch nicht durch vollständige Suche). $F(X)$ muß lang genug sein, damit die vollständige Suche hoffnungslos wird. Es gibt z.B. schon bei 128 Bits 2^{128} Möglichkeiten (und der Weltall ist nur $\approx 2^{61}$ Sekunden alt).

Bei Anwendung (1) veröffentlich man den Fingerprint (z.B. in der Zeitung). Um den Beweis durchzuführen, reicht es nachher, den Originaltext zu zeigen. Um Zweck (2) zu erfüllen, publiziert man den Fingerprint als Kontrollcodewort für die eigene Datei. Ändert jemand die Datei (z. B. im Programm die Lizenznummer oder versucht jemand, einen Viren einzubauen), ist dieses leicht zu erkennen.

... – und viele andere Anwendungen.

3.1.2 Kurzgeschichte der Kryptologie

- immer stark unter staatlicher Kontrolle (in einigen Länder ist die Publizierung auch heute noch verboten (Rußland) oder beschränkt (USA)). Auch die Forschung wird kontrolliert.
- Bis in die siebziger Jahre gab es nur symmetrische Verfahren.
- Große Fortschritte erst ab den siebziger Jahren durch „public domain"-Forschung an Universitäten
- wie beweist man heute die Sicherheit eines kryptographischen Verfahrens:
  1. man veröffentlicht das Verfahren,
  2. man läßt die Forscher aus den ganzen Welt versuchen, die Chiffren zu „knacken",
  3. Knackt nach langjährigen Versuchen niemand die Chiffren, so kann das Verfahren als sicher gelten.

3.2 Grundlegende Methoden der Chiffrierung

3.2.1 Substitution:

Sei $\pi : \Sigma \to \Sigma$ eine Permutation. Der Klartext $a_1 a_2 \dots a_n$ wird durch das Kryptogramm $\pi(a_1)\pi(a_2)\dots\pi(a_n)$ ersetzt.

Beispiel Cäesar-Chiffren: $\pi(i) := i \oplus_{26} 3$ für alle $i \in \{0, \dots, 25\} = \{\mathrm{a}, \dots, \mathrm{z}\}$.

Häufigkeitsanalyse

Die Schwäche von solchen Substitutionschiffren liegt daran, daß sie durch eine Häufigkeitsanalyse geknackt werden können. Zum Beispiel beträgt die durchschnittliche Häufigkeit von Buchstaben in englischen Texten:

E	-	12.31 %	L	-	4.03 %	B	-	1.62 %
T	-	9.59 %	D	-	...	G	-	...
A	-	...	C	-		V	-	
O	-		U	-		K	-	
N	-		P	-		Q	-	
I	-		F	-		X	-	
S	-		M	-		J	-	
R	-		W	-		Z	-	0.09 %
H	-	5.14 %	Y	-	1.88 %			

Man kann die Häufigkeiten aus dem Kryptogramm mit dieser Tabelle vergleichen und die Werte von z.B. $\pi(E)$, $\pi(T)$, $\pi(A)$ raten. Dann sucht man die typischen Wörter wie z.B. „the".

Auch eine Erweiterung von π auf Blöcke ($\pi : \Sigma^k \to \Sigma^k$) kann mit Hilfe der Häufigkeitsanalyse geknackt werden. Jedoch sind hierzu (in Abhängigkeit von k) längere Texte nötig.

Beispiel für Blöcke der Länge 2: Fairplay-System, siehe Abb. 3.5.

Korollar: Substitution allein ist zu schwach und kann nur als Hilfsmittel dienen.

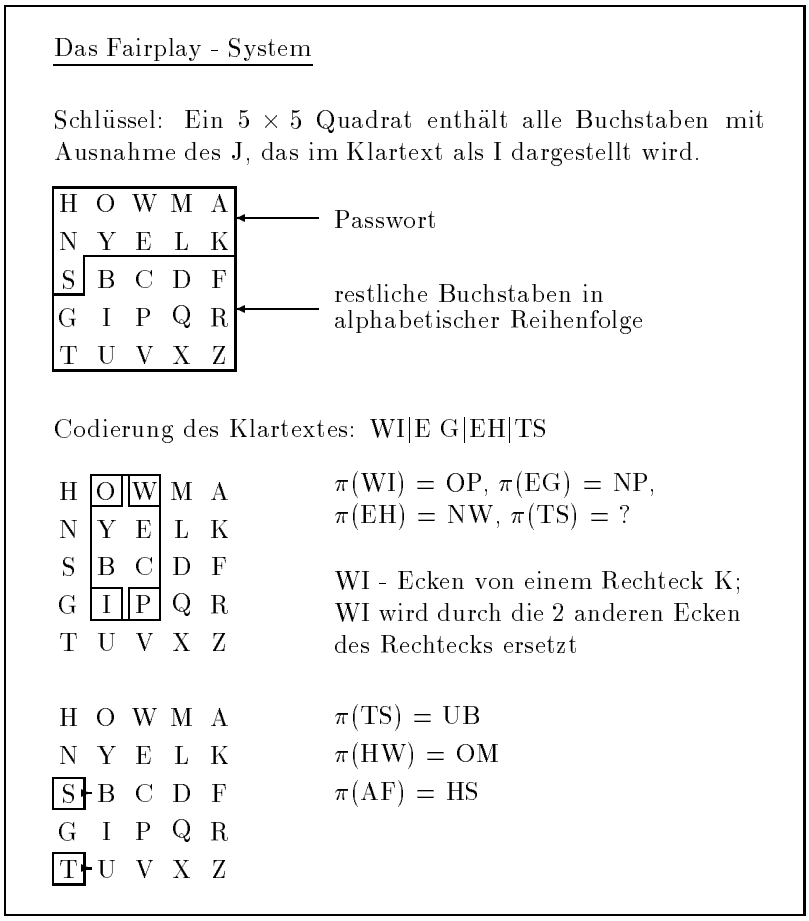


Abbildung 3.5: Fairplay-System

3.2.2 XOR, one-time-pad und Rotor Codes

Die Operation XOR ($\oplus$) wird über $\{0,1\}$ wie folgt definiert:

$$0 \oplus 0 = 1 \oplus 1 = 0, \qquad 1 \oplus 0 = 0 \oplus 1 = 1.$$

Chiffrieren mir XOR: Sei der Klartext eine Bitfolge $a_0 \dots a_{n-1}$ und der Schlüssel eine Bitfolge $k_0 \dots k_{n-1}$. Dann ist das Kryptogramm die Bitfolge $c_0 \dots c_{n-1}$ mit $c_i := a_i \oplus k_i$ für $i = 0, \dots, n-1$.

Diese Methode hat einige Anwendungen.

One-time-pad

Wähle $k_1, \dots, k_n$ zufällig und verwende den Schlüssel nur für **einen** Klartext. Eigenschaften:

- Das Kryptogramm ist auch eine zufällige Folge von n Bits.
- Ohne Schlüssel kann man one-time-pad Codes überhaupt nicht knacken.

  Beweis: Für jeden Klartext der Länge n gibt es einen Schlüssel der Länge n, der das gegebene Kryptogramm erzeugt! □

Anwendung von one-time-pad: für kurze aber sehr wichtige Informationen, wie z.B.: „seit heute morgen befinden wir uns im Krieg“. Man sollte nicht durch einen Hacker einen Krieg anfangen!

Problem: Der Schlüssel
- muß im voraus vereinbart sein.
- muß zufällig gewählt werden.
- muß sicher aufbewahrt sein.
- muß mindestens so lang sein wie die Nachricht, die verschlüsselt werden soll.

XOR mit kurzem Schlüssel

Vorschlag 1 Jeder Block wird mit dem gleichen one-time-pad verschlüsselt:

$$c_i = a_i \oplus k_{i \bmod n}.$$

Für das Kryptogramm $c_1, \ldots, c_{2n}$ ergibt sich für jedes $j \in \{0, \ldots, n-1\}$:

$$c_j \oplus c_{n+j} = a_j \oplus k_j \oplus a_{n+j} \oplus k_j = a_j \oplus a_{n+j}$$

Mit diesen Informationen läßt sich der Klartext erraten und somit auch der Schlüssel. Die gleiche Begründung zeigt, warum es gefährlich ist, denselben Schlüssel mehrmals für one-time-pad zu verwenden.

Vorschlag 2 (Rotor Codes)

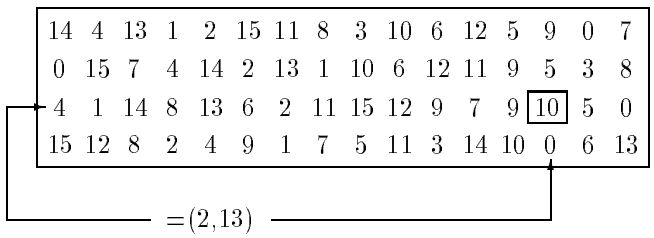
Eine Rotor Maschine besteht aus aus t Rotoren, die auf einer Achse befestigt sind. Auf jeder der beiden kreisförmigen Seiten eines Rotors sind jeweils 26 elektronische Kontakte gleichmäßig angebracht. Innerhalb des Rotors sind die 2×26 Kontakt miteinander verbunden, so daß eine Permutation entsteht. Legt man also an einen freien Kontakt am ersten Rotor eine Spannung an, so wird diese durch den ersten Rotor geschleust, über den Kontakt an der anderen Seite an den zweiten Rotor übergeben, usw., bis die Spannung an genau einem freien Kontakt des t-ten Rotors angekommen ist. Hierdurch hat man ein Zeichen codiert, die Rotoren werden gedreht (z.B. wie bei einem Kilometerzähler) und das nächste Zeichen codiert. Die maximal mögliche und auch praktisch erreichbare Periode der angewandten Schlüssel beträgt 26^t.

In einer Haglin Maschine sind die t Rotoren in p_i, $1 \leq i \leq t$ Segmente unterteilt, wobei die p_i zueinander paarweise prim sind und die Segmente wahlweise (initial) mit einer 0 oder einer 1 bestückt werden können. Die vorne abzulesene t-Bitfolge ist der erste Schlüssel. Danach werden *alle* Rotoren um eine Position weitergedreht und man kann vorne den zweiten Schlüssel ablesen. Da die p_i paarweise prim sind, tritt die gleiche Konfiguration der Rotorscheiben erst nach $\prod p_i$ Schlüsseln auf (hat man Rotoren mit 17, 19, 21, 23, 25 und 26 Segmenten, so hat man eine Periodenlänge von über 10^8).

Während des zweiten Weltkriegs wurde dieser Ansatz von den Amerikanern zur Verschlüsselung mit der C-36 benutzt. Dazu wird eine Lug-Matrix L verwendet – eine 6×27 Matrix mit Einträgen aus $\{0, 1\}$, wobei in jeder Spalte maximal zwei Stellen ungleich 0 sind –, die mit dem s-ten gewonnen 6-Bit Schlüssel K_s multipliziert wird. Dieses ergibt einen Vektor $v_s = K_s \cdot L$ mit 27 Einträgen aus $\{0, 1, 2\}$. Innerhalb dieses Vektors wird die Anzahl h_s an Einträgen ungleich 0 bestimmt – $h_s \in \{0 \ldots, 27\}$ – die auch Trefferanzahl (Hitnumber) genannt wird. Ein Buchstabe $\alpha \in \{1, \ldots, 26\}$ wird dann als $\gamma = h_s - \alpha - 1$ codiert. Da hiermit auch $\alpha = h_s - \alpha - 1$ ist, kann mit der gleichen Maschine mit demselben Verfahren dechiffriert werden.

3.2.3 S-Boxen

Die S-Box $\in M(4, 16; \{0, \ldots, 15\})$ ist die wichtigste Komponente von DES, dem **D**ata **E**ncryption **S**tandard.



Eingabe (10, 1101), Ausgabe= $10_{10} = 1010_2$

Abbildung 3.6: Verschlüsseln mit S-Boxen

Eingabe: 6 Bits = eine 2-Bit Zahl und eine 4-Bit Zahl
Ausgabe: 4-Bit Zahl

Verlauf einer Berechnung:

- 2 Bits der Eingabe bestimmen eine Zeile,
- 4 Bits der Eingabe bestimmen eine Spalte,
- der Schnittpunkt der Spalte und der Zeile enthält eine Zahl x, die ausgegeben wird.

Die Auswahl der Zahlen in der S-Box ist kritisch für gute Codes. Es sind keine klaren und einfachen Methoden zur Auswahl der Zahlen (öffentlich) jedoch einige Eigenschaften, die S-Boxen erfüllen sollten, bekannt.

Lawieneneffekt: Die Änderung eines Bits im Klartext soll sehr viele Bits im Kryptogramm ändern, um die Kryptoanalyse zu erschweren (damit ähnliche Klartexte unähnliche Kryptogramme haben).

Mit Hilfe von S-Boxen kann der Lawineneffekt gut erreicht werden. Erst kann man eine Erweiterungs-Permutation anwenden, damit die Anzahl der Bits durch die S-Box nicht reduziert wird:

$$(x, y, z, w) \to (y, w, z, x, w, z)$$

Dann bewirkt z.B. die Änderung des Bits w:
yw = Nummer der S-Box Zeile, $zxwz$ = Nummer der S-Box Spalte.
Deswegen gibt es keinen Grund, warum die Ausgabe zur urprünglichen ähnlich sein sollte.
Die S-Boxen werden iterativ mehrmals benutzt, damit die Änderung eines Eingabebits eine Lawine von Änderungen im ganzen Kryptogramm verursacht.

3.3 DES (Data Encryption Standard)

- Das Verfahren ist symmetrisch, d.h. es wird der gleiche Schlüssel für Chiffrierung und Dechiffrierung benutzt.
- Der Algorithmus ist veröffentlicht. Im Gegensatz hierzu kann ein nicht veröffentlicher Algorithmus eine **Trap-Door** (**Falltür**) enthalten, d.h. einen Weg die Information ohne die Kenntnis des Schlüssels zu dechiffrieren. Dadurch kann der Autor (bzw. die Herstellerfirma) alle Kryptogramme dechiffrieren und die geheimen Informationen ausnutzen oder verkaufen.)

- DES ist wesentlich besser in Hardware als in Software (Software-Lösungen sind weniger sicher). DES-Chips sind sehr schnell (bis 1 GB/Sek.); wohingegen schnelle Software-Lösungen $\approx$ 320KB/Sek. auf einem 80486 erreichen.
- DES chiffriert Blöcke von 64 Bits, normalerweise 8 Zeichen mit 8 Kontrollbits, d.h. 56 Informationsbits, wobei die Schlüssel aus 64 Bits bestehen. Im Schlüssel sind 8 Kontrollbits vorhanden. Es wird vermutet, daß die Schlüssel zu kurz sind. Durch vollständige Suche kann man mit $\approx$ 10 Millionen \$ Kosten einen Code knacken.
- DES ist als Standard für nicht militärische Zwecke empfohlen; entwickelt durch IBM in Yorktown Heights, modifiziert wobei die NSA (National Security Agency) mitgespielt hat. Es bestand immer der Verdacht, daß die CIA eine Trap-Door kennt.
- Mathematische Grundlagen der Konstruktion sind nicht veröffentlicht. 20 Jahre akademische Forschung zeigen, daß die Konstruktion von DES sehr raffiniert ist. Es ist keine entscheidende Verbesserung der Kryptoanalyse im Vergleich zur vollständigen Suche gefunden worden. Die vereinfachten Versionen von DES scheinen alle viel schwächer zu sein. Dagegen bringen die komplizierteren Vorschläge (auch mit längeren Schlüsseln) bis jetzt keine Verbesserung der Sicherheit.

DES ist ein iteratives Verfahren und besteht aus 16 Runden (zu einer genauen Spezifikation der erwähnten Permutationen und S-Boxen siehe Anhang B). Jede Runde i benutzt einen eigenen Runden-Schlüssel K_i. Diese Schlüssel hängen von Chiffrierungsschlüsseln ab.

3.3.1 Erzeugung von Runden-Schlüsseln

1. 8 Parity-Bits werden entfernt, die verbleibenden 56 Bits werden permutiert und in zwei Hälften aufgeteilt,
2. for $j = 1$ to 16
   jede Hälfte wird um 1 (für j=1,2,9,16) oder
   um 2 (alle anderen j's) zyklisch geshiftet,
3. Der Runden-Schlüssel K_i wird aus dem Zwischenergebnis in der Schleife für $j = i$ generiert: Aus den 56 Bits werden 48 Bits gewählt, und zwar die Bits $14, 17, 11, 24, \ldots, 29, 32$ (das ist die sogenannte Compression Permutation).

3.3.2 Preprocessing, Postprocessing

Am Anfang werden die Bits des Klartextes permutiert. Das hat keinen kryptographischen Grund. Die Permutation ist einfach durch Drähte in VLSI zu implementieren. Dagegen ist eine Software-Simulation sehr aufwendig. Am Ende des Chiffrierens wird die entsprechende inverse Permutation durchgeführt.

3.3.3 DES Runde

Die Eingabe einer Runde $i+1$ wird in zwei Folgen L_{i+1}, R_{i+1} mit je 32 Bits geteilt. Nach Runde $i+1$ gilt:

$$\begin{aligned} L_{i+1} &= R_i \\ R_{i+1} &= L_i \oplus f(R_i, K_{i+1}) \end{aligned} \tag{3.1}$$

f ist eine Funktion, die die eigentliche Verschlüsselung realisiert. Die Berechnung von f wird durch Abbildung 3.7 dargestellt und erfolgt folgendermaßen:

1. Durch eine Erweiterung-Permutation erhält man eine 48-Bit Folge aus den 32 Bits von R_i (einige Bits werden wiederholt und dann wird permutiert).
2. Die 48 Bits werden mittels XOR mit dem Runden-Schlüssel K_{i+1} kombiniert.
3. Die 48 resultierenden Bits werden auf 8 Gruppen mit 6 Bits aufgeteilt. Für jede Gruppe wird eine von 8 DES S-Boxen verwendet.
4. Die gewonnenen 32 Bits werden zum Abschluß noch einmal permutiert.

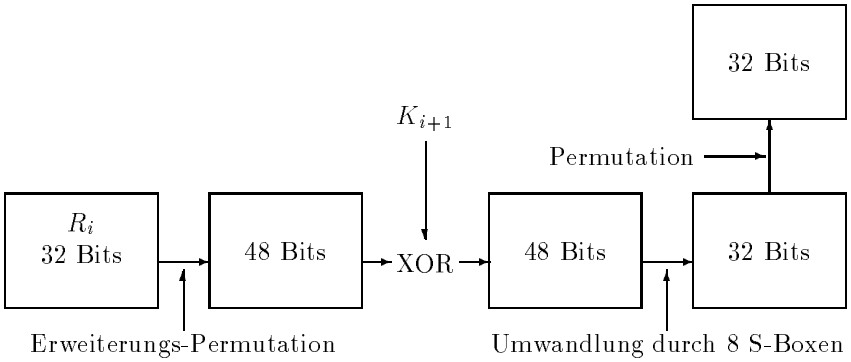


Abbildung 3.7: DES Funktion f

3.3.4 Dechiffrierung

Um ein Kryptogramm zu entschlüsseln muß man normalerweise die ganze Berechnung rückgängig machen. Bei S-Boxen ist dies aber nicht möglich! Deswegen brauchen wir einen Trick:

Falls L_i und R_i bekannt sind, dann gilt nach Gleichung (3.1):

$$\begin{aligned} R_{i-1} &= L_i \\ L_{i-1} &= R_i \oplus f(R_{i-1}, K_i) = R_i \oplus f(L_i, K_i) \end{aligned}$$

Also: zum Dechiffrieren braucht man nicht f^{-1} zu berechnen! Es ist leicht zu sehen, daß zum Entschlüsseln dieselbe Hardware wie zum Chiffrieren benutzt werden kann (nur die Reihenfolge der Runden-Schlüssel wird geändert).

3.4 Chiffrierung von langen Nachrichten

DES chiffriert nur sehr kurze Nachrichten (8 Buchstaben!). Um die Methode nutzbar zu machen, muß man einen Weg finden, Nachrichten beliebger Länge zu chiffrieren. Drei allgemeine Methoden werden im folgenden vorgestellt (welche bei jedem Codierungsverfahren funktionieren, das Blöcke fester Länge chiffriert).

3.4.1 Elektronisches Codebuch

Das **Elektronisches Codebuch** (ECB) (s. Abbildung 3.8) funktioniert wie folgt: jede Nachricht wird auf Blöcke aufgeteilt, die dann separat chiffriert werden. Für jeden Block verwenden wir den gleichen Schlüssel.

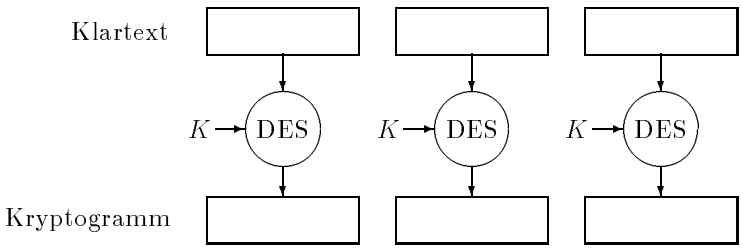


Abbildung 3.8: Elektonisches Codebuch (ECB)

Das ECB hat den Vorteil, daß die Beschädigung oder der Verlust einzelner Teile keine Auswirkung auf andere Teile des Kryptogramms hat. Das ECB ist jedoch nicht allzu sicher (auch wenn K ungeknackt bleibt).

Beispiel: Wir nehmen an, daß die Kommunikation zwischen Banken für Geldüberweisungen mit ECB codiert ist:

Geldüberweisung =	BLZ1	BLZ2	Empf	änger	Konto	Betrag
Code =	Block 1	Block 2	Block 3	Block 4	Block 5	Block 6

Jetzt ist der folgende Betrug möglich:

1. Der Verbrecher tätigt in verschiedenen Zeitintervallen mehrere Überweisungen von Bank 1 zu Bank 2 und identifiziert den Code seines Namens und seines Kontos.
2. Er ändert mehrere Nachrichten, die über die Leitung gehen, so daß er als Empfänger angegeben wird (ohne Ahnung wieviel Geld dadurch auf seinem Konto landet).
3. In einem günstigen Moment hebt er das Geld ab und verschwindet.

3.4.2 Cipher Block Chaining

Das **Cipher Block Chaining** (CBC) verknüpft alle Blöcke miteinander:
Besteht der Klartext aus den Blöcken $P_1, P_2, P_3, \ldots$, so werden die Kryptogramme $C_1, C_2, C_3, \ldots$ definiert durch:

$$\begin{aligned} C_1 &= \text{Code}(P_1 \oplus I) \\ C_i &= \text{Code}(P_i \oplus C_{i-1}) \end{aligned}$$

Der zufällig erzeugte Vektor I wird unchiffriert gesendet.
Entschlüsseln ist einfach:

$$\begin{aligned} P_1 &= \text{Decode}(C_1) \oplus I \\ P_i &= \text{Decode}(C_i) \oplus C_{i-1} \end{aligned}$$

CBC hat die folgende Eigenschaften:

- Vorteil: Gleiche Blöcke werden in der Regel durch verschiedene Codes repräsentiert (ein Angriff wie bei ECB ist damit nicht mehr möglich).

- Nachteil: Man kann keinen Block aus der Kette nach dem Chiffrieren entfernen (sehr schlecht für Datenbanken), ähnliche Probleme gibt es bei Insert-Operation,
- Nachteil: Die Aufteilung auf Blöcke muß sicher sein (ein zusätzliches Bit oder ein fehlendes Bit machen die Nachricht unlesbar),
- Es gibt nur geringe Probleme bei Fehlern innerhalb eines Blocks (ohne Änderung der Anzahl der Bits in einem Block). Zum Beispiel bei einem Fehler in Block C_s wird die Folge der Decodieroperationen Decode$(C_i) \oplus C_{i-1}$ nur für $i = s, s+1$ beeinflußt.

3.4.3 Cypher Feedback Mode

Der Cypher Feedback Mode (CFM) dient z.B. zur Kommunikation zwischen Terminal und Server. Dabei werden einzelne Bytes codiert und verschickt. Abbildung 3.9 zeigt eine allgemeine Lösung, bei der DES verwendet wird. Dabei wird das Queue-Register zufällig initialisiert (die zufällige Folge wird in Klartext gesendet) und danach der Inhalt jeweils um 8 Bit verschoben, d.h. es werden 8 Bits von der linken Seite entfernt, die Folge um 8 Bit verschoben und von rechts mit den 8 Bit aus der Operation $P_t \oplus Z_t$ aufgefüllt.

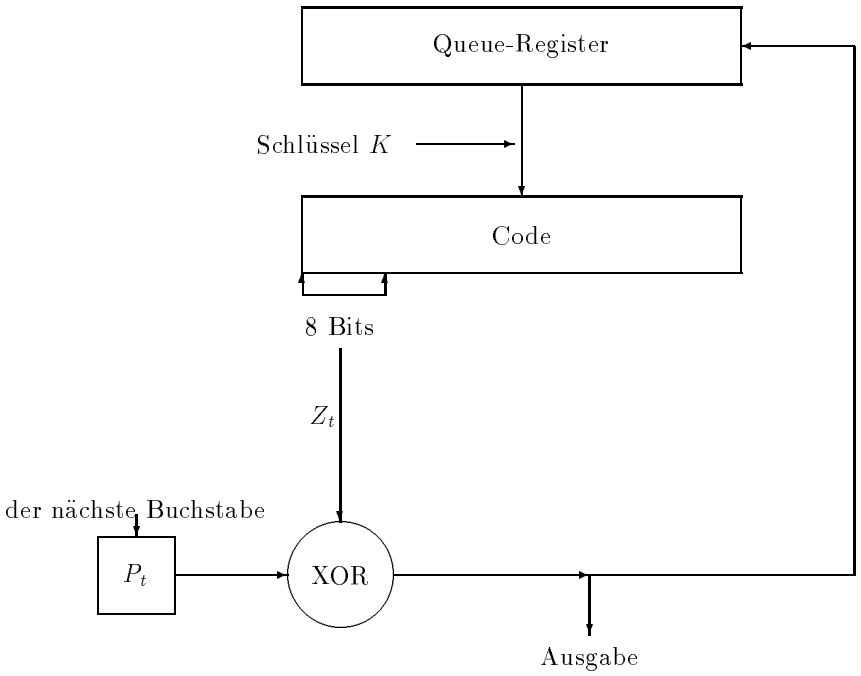


Abbildung 3.9: Cypher Feedback Mode (CFM)

3.5 IDEA

Grundlage: Die Schlüssel bei DES sind möglicherweise zu kurz. Deshalb werden neue Algorithmen gesucht. Neben vielen anderen Codes wurde IDEA (International Data Encryption Standard) vorgeschlagen.

Eigenschaften von IDEA:

- kostenlos für nichtkommerzielle Zwecke
- nicht alt – eingeführt in 1992 (solche Chiffren werden als gefährlich betrachtet – zahlreiche neue Verfahren wurden nur wenige Jahre später geknackt)
- eine Komponente vom PGP (populäres public-domain kryptographisches Paket)
- die Schlüssel sind 128 Bits lang und somit zu lang, um eine vollständige Suche zu ermöglichen,
- trotz der Größe der Schlüssel sind die IDEA-Programme nicht langsamer als die von DES

3.5.1 Komponenten von IDEA

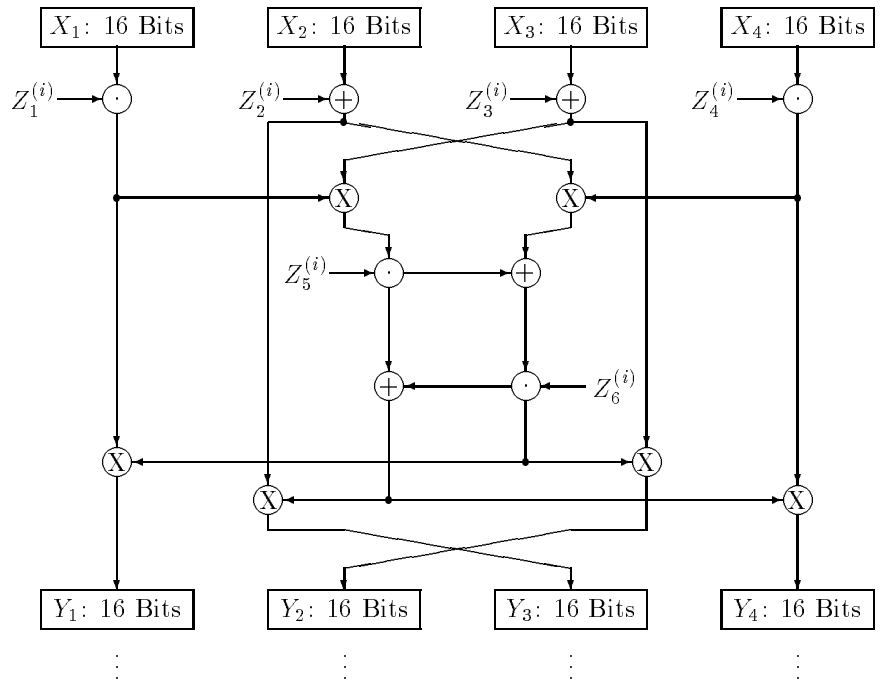
- Das Chiffrieren erfolgt in 8 Runden. Eine Runde wird schematisch in der Abb. 3.10 beschrieben. Zusätzlich wird nach der letzten Runde (aus technischen Gründen notwendig $\rightarrow$ siehe Dechiffrierung) – siehe Abbildung 3.11 – ein Postprocessing durchgeführt.
- Jede Runde führt verschiedene Operationen auf 16-Bit Blöcken durch. 3 Operationen werden verwendet:

  bitweise XOR , Addition mod 2^{16}, Multiplikation mod $(2^{16}+1)$
- Der Schlüssel enthält 128 Bits. Für die Runde i werden Runden-Schlüssel $Z_1^{(i)}, \ldots Z_6^{(i)}$ gebraucht.
- Ein Klartext enthält 64 Bits.
- Eine Runde wird durch Abbildung 3.10 dargestellt.

3.5.2 Dechiffrierung von IDEA

Wie bei DES muß man eine schlaue Methode anwenden, damit nicht jede Operation der Verschlüsselung direkt rückgängig gemacht werden muß.

Trick 1: Sei $A = X_1 \cdot Z_1^{(i)}, B = X_2 + Z_2^{(i)}, C = X_3 + Z_3^{(i)}, D = X_4 \cdot Z_4^{(i)}, E = (A \oplus C) \cdot Z_5^{(i)}$ und $F = (E + (B \oplus D)) \cdot Z_6^{(i)}$, also die Ausgänge der Operationen, in die die Schlüssel $Z_k^{(i)}$ einfließen. Es gilt (siehe Abbildung 3.10), daß $Y_3 \oplus Y_4 = B \oplus D$ ist. Deswegen ist $B \oplus D$ bekannt. Ähnlich kann man $A \oplus C$ ausrechnen. Außerdem gilt, daß $B \oplus D$ und $A \oplus C$ Ausgangswerte der zwei XOR-Gatter im mittleren Teil des Schaltkreises sind. Kennt man also die Schlüssel $Z_5^{(i)}$ und $Z_6^{(i)}$, so kann man E und F berechnen, mit deren Hilfe man die folgenden Berechnungen durchführt: $A = Y_1 \oplus F$, $B = Y_3 \oplus (E + F)$, $C = Y_2 \oplus F$, $D = Y_4 \oplus (E + F)$. Aus A, B, C, D und den Schlüsseln $Z_1^{(i)}$ bis $Z_4^{(i)}$ kann man zum Schluß $X_1, \ldots, X_4$ ableiten.
Auf diese Weise haben wir eine Runde rückgängig gemacht. Man kann dies für alle Runden iterativ durchführen, wenn wir das Kryptogramm und die Runden-Schlüssel kennen.



⊙ Multiplikation

⊕ Addition

Ⓧ XOR

$Z_1(i), Z_2(i), \ldots$ - Rundenschlüssel von Runde i

Abbildung 3.10: Eine Runde beim IDEA Verfahren

Trick 2: Wir haben soeben gesehen, daß Dechiffrierung möglich ist. Falls wir die Operationen genau auflisten, kann man beobachten, daß sie genau den Operationen bei der IDEA-Chifrierung entsprechen (dazu ist aber ein Postprocessing notwendig). Die Rundenschlüssel fließen in der umgekehrten Reihenfolge ein und müssen invertiert (Multiplikation) oder negiert (Addition) werden.

Also kann bei IDEA derselbe Chip zum Chiffrieren und Dechiffrieren verwendet werden!

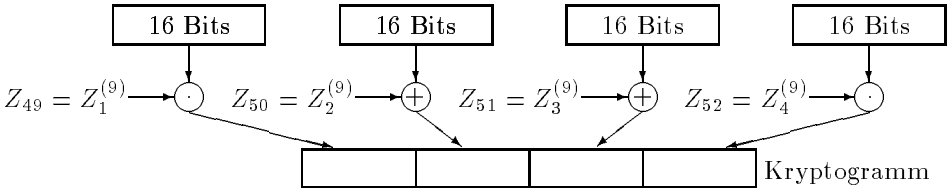


Abbildung 3.11: Postprocessing beim IDEA Verfahren

3.5.3 Erzeugung von Runden-Schlüsseln

IDEA benötigt $6 \cdot 8 + 4$ Schlüssel (wobei 8 die Rundenzahl ist und 4 Schlüssel für das Postprocessing benötigt werden), die wie folgt erzeugt werden:

- Die 128 Bits des geheimen Schlüssels werden in 16-Bit Blöcke aufgeteilt. Das ergibt 8 Runden-Schlüssel (6 für die erste Runde, 2 für die zweite Runde)
- Der geheime Schlüssel wird nun um 25 Positionen zyklisch geshiftet und neu aufgeteilt. Das ergibt weitere 8 Runden-Schlüssel (die 4 restlichen für die zweite Runde, 4 für die dritte Runde)
- ... (wiederholt geshiftet und geteilt)

3.6 Public key Verfahren

Ein Chiffrierverfahren heißt **public key** oder asymmetrisch, falls

- die Schlüssel in Paaren vorkommen (ein Schlüssel zum Chiffrieren und einer zum Dechiffrieren).
- die Veröffentlichung des Chiffrier-Schlüssels (bzw. Dechiffrier-Schlüssel) verrät nichts über den Dechiffrier-Schlüssel (bzw. Chiffrier-Schlüssel), auch wenn man sehr aufwendige Berechnungen machen darf.

3.6.1 Komplexitätstheorie versus Kryptologie

Für die Konstruktion von guten Codes möchte man auf Probleme zurückgreifen, die beweisbar schwierig sind. Zuerst zeigen wir, daß das Knacken von public key Codes *nur* ein Problem aus der Klasse NP ist.

Kryptoanalyseproblem

Eingabe: Kryptogramm + public key (zum Chiffrieren)
Ausgabe: ein passender Klartext und ein Schlüssel zum Dechiffrieren

Lemma 53 *Das Kryptoanalyseproblem liegt in NP.*

Beweis: Man findet leicht einen NP-Algorithmus für das Kryptoanalyseproblem:

1. rate den Schlüssel zum Dechiffrieren,
2. dechiffriere das Kryptogramm mit dem geratenen Schlüssel,
3. überprüfe, ob der Klartext dem Kryptogramm entschpricht. □

Es folgt daraus, daß ein public key System nicht schwieriger als NP-vollständig sein kann. (Dagegen haben symmetrische Verfahren keine derartige Beschränkung der Komplexität. Ein Kryptogramm allein reicht nicht zur Verifizierung daß der durch den NP-Algorithmus geratene Schlüssel korrekt ist.) Bedeutet dies, daß public key Verfahren sehr schwach sind? Nicht ganz:

- Kryptologie arbeitet mit natürlichen Zahlen, die auch in der Wirklichkeit (im Rechner) als Binärzahl dargestellt werden. Für die Kryptologie ist z.B. 2^{500} eigentlich schon $+\infty$ (es gibt angeblich ca. 2^{280} Atome im Weltall). Demgegenüber untersucht die Komplexitätstheorie die asymptotische Komplexität der Probleme. Das Verhalten bei kurzen Eingaben kann beliebig schwanken. Dieser Effekt fließt in die Komplexitätsbetrachtung nicht ein.

- Komplexitätstheorie arbeitet im Rahmen des O-Kalküls. Ist die Komplexität einer Funktion genau $2^{500}n$, dann wird sie als $\Theta(n)$ klassifiziert (Linearzeit!), obwohl sie *praktisch* unberechenbar ist. Dagegen läßt sich für eine Funktion mit der exponentiellen Komplexität $2^{n/100}$ für viele kurze (und damit vielleicht die *praktisch* relevanten) Eingabefolgen das Ergebnis berechnen.
- für die Kryptologie braucht man die Komplexität im besten Fall (oder zumindest im mittleren Fall als Hinweis; dann sollte die Abweichung vom Mittelwert klein sein). Komplexitätstheorie betrachtet hingegen immer den schlechtesten Fall.

Trotzdem kann man versuchen die Ergebnisse der Komplexitätstheorie ausnutzen.

Idee: Man sollte möglichst schwierige Probleme aus NP benutzen, z.B. NP-vollständige Probleme, damit die Kryptoanalyse NP-vollständig wird.

3.6.2 Knapsack Codes

Knapsackproblem

Gegeben: natürliche Zahlen $(a_1, \ldots, a_n), k$
Aufgabe: finde $i_1, \ldots, i_n \in \{0, 1\}$, so daß $\sum i_j \cdot a_j = k$
Es ist bekannt, daß:

Theorem 54 *Das Knapsackproblem ist NP-vollständig.*

Es gibt mehrere Knapsack-Codes (die meisten davon wurden geknackt). Wir lernen einen einfachen Knapsack Code kennen (geknackt durch Rabin, siehe z.B. das Buch „Public key cryptography“ von A.Salomaa).

Chiffrieren mit Knapsack-Vektoren

Ein Vektor $(i_1, \ldots, i_n)$ wird durch $\sum_{j=1}^{n} i_j \cdot a_j$ codiert.
Probleme:

1. Solche Codes sind nicht unbedingt eindeutig, d.h. es können zwei Vektoren $(i_1, \ldots, i_n)$ und $(k_i, \ldots, k_n)$ existieren mit $\sum i_j \cdot a_j = \sum k_j \cdot a_j$.
2. Das Knacken von solchen Codes ist offensichtlich nicht leicht (NP-Vollständigkeit von Knapsack-Problem), aber man kennt auch keine Methode, mit dem geheimen Schlüssel zu decodieren.

Beide Probleme werden durch eine Beschränkung der Knapsack-Vektoren gelöst.

Super Increasing Vektoren

Ein Vektor $(a_1, \ldots, a_n)$ heißt **super increasing**, falls

$$\forall i \leq n : \quad \sum_{j<i} a_j < a_i$$

Lemma 55 *Für einen super increasing Vektor gilt*

1. *die Codes sind eindeutig*
2. *aus $k = \sum_j i_j \cdot a_j$ und $a_1, \ldots, a_n$ kann man $i_1, \ldots, i_n$ in linearer Zeit berechnen.*

Beweis: der erste Teil folgt aus den ersten zweiten.
Beachte, daß $k \geq a_n \Leftrightarrow i_n = 1$ (Die Summe von allen anderen a_i reicht nicht aus um a_n zu überschreiten). Nach Bestimmung von i_n betrachten wir $k' = k - i_n \cdot a_n$ und den super increasing Vektor $a_1, \ldots, a_{n-1}$ und machen rekursiv weiter. □

Problem: Bei super increasing Vektoren sind die Codes eindeutig, aber jeder kann die Codes leicht dechiffrieren.

Lösungsvorschlag: Geheim wählt man:

1. $M > \sum a_i$
2. $W < M, W > 1$, so daß W und M teilerfremd sind (W soll auch nicht klein sein, $W > M/2$ ist ok).

Sei $a'_i = a_i \cdot W \bmod M$. Dann bildet die (eventuell permutierte) Folge $a'_1, \ldots, a'_n$ (die jetzt nicht mehr super increasing ist!) den public key und $a_1, \ldots, a_n, M, W$ den private key.

Chiffrieren mit dem speziellen Knapsack Code

Der Code von $i_1, \ldots i_n$ ist $\sum_{j=1}^{n} i_j \cdot a'_j$

Dechiffrieren von dem speziellen Knapsack Code

Sei $y = i_1 \cdot a'_1 + \ldots + i_n \cdot a'_n$ die Nachricht, gesucht werden $i_1, \ldots, i_n$:

- da W und M teilerfremd sind: $\exists r, s : r \cdot W + s \cdot M = 1$, bzw. $r \cdot W = 1 \bmod M$. Also ist $r = W^{-1} \bmod M$ leicht mit dem euklidischen Algorithmus zu bestimmen.
- Es gilt:
  $W^{-1} \cdot y = W^{-1}(i_1 \cdot a'_1 + \ldots + i_n \cdot a'_n) = W^{-1}(i_1 \cdot a_1 \cdot W + \ldots + i_n \cdot a_n \cdot W) = i_1 \cdot a_1 + \ldots + i_n \cdot a_n \bmod M$
- Es ist also noch das Knapsack-Problem für $W^{-1}y$ und den super increasing Vektoren $a_1, \ldots, a_n$ zu lösen (die Laufzeit ist linear in n).

Leider kann man diese Codes nicht verwenden, weil eine erfolgreiche Kryptoanalyse für sie existiert (siehe zum Beispiel das Buch von Salomaa). Es bedeutet nicht, daß dadurch das Knapsack Problem gelöst würde. Durch die zusätlichen Informationen über das spezielle Knapsackproblem (der Vektor war vorher super increasing, ...) kann man das spezielle Problem effizient lösen. Für beliebige Eingaben (Knapsack-Vektor + die Summe) kennt man keine effiziente Methode.

3.7 RSA

RSA wurde in 1978 von Rivest, Shamir, Adleman vorgeschlagen.

- Es ist ein Standard-Algorithmus für public key Verfahren.
- Die Länge des Schlüssels kann angepaßt werden (obwohl die minimale Länge ca. 100 Ziffern beträgt).
- RSA hat massive kryptoanalytische Angriffe bestanden (obwohl man zur Sicherheit die Schlüssellänge anpassen mußte).

- RSA ist *kommutativ*, d.h. falls E die Chiffrierungs-, D die Dechiffrierungsfunktion, k_1, k_2 ein Paar von Schlüssel für E, bzw. D ist, dann gilt:

$$\forall x: \quad D(E(x,k_1),k_2) = x \tag{3.2}$$

$$\forall x: \quad E(D(x,k_2),k_1) = x. \tag{3.3}$$

  Bei RSA sind sogar die Dechiffrierungs- und Chiffrierungsfunktion gleich.

3.7.1 Anwendungen von RSA

Beide Gleichungen (3.2) und (3.3) sind nutzbar:
Alice hält k_1 geheim und veröffentlicht k_2.

- *Geheimhaltung der Nachrichten:*
  Bob schickt eine geheime Nachricht M an Alice:
  Bob codiert dazu M als $D(M,k_2)$. Alice empfängt $D(M,k_2)$ und berechnet $E(D(M,k_2),k_1) = M$. Ohne k_1 kann man M nicht berechnen, deswegen kann nur Alice den Brief lesen.

- *Authentifikation:*
  Alice schickt einen öffentlichen Brief M an alle. Alice codiert M als $E(M,k_1)$. Jeder Empfänger entschlüsselt die Nachricht mit k_2: $D(E(M,k_1),k_2)$.

  Nur Alice kann der Autor sein, da die Verschlüsselung mit einem anderen Schlüssel k_3 bei der Decodierung eine Folge von Bits ohne Bedeutung ergibt.

RSA benötigt viele arithmetische Operationen, deswegen ist die Geschwindigkeit beim Chiffrieren und Dechiffrieren erheblich geringer als bei DES (ungefähr Faktor 100 bis 1000). Deswegen sind die Anwendungen auf typische public key Bereiche beschränkt. So vereinbart man z.B. mit RSA einen session key für eine Video-on-demand Übertragung. Die eigentliche Übertragung wird dann durch ein symmetrisches Verfahren mit dem vereinbarten session key codiert.
RSA ist im PGP-Paket enthalten.

3.7.2 RSA Algorithmus

Die Auswahl der Schlüssel und die Chiffrierung

Die Auswahl der Schlüssel:

1. Es werden zufällig zwei große Primzahlen p, q gewählt *(wie das realisiert ist wird später erklärt).*

2. Es wird zufällig eine Zahl e gewählt, so daß e und $(p-1)(q-1)$ teilerfremd sind *(es wird einfach irgendein e gewählt und mit dem euklidischen Algorithmus $ggT(e,(p-1)(q-1))$ berechnet; wurde e falsch geraten dann versucht man noch mal).*

3. Durch den euklidischen Algorithmus wird ein d gefunden, so daß $e \cdot d = 1 \bmod (p-1)(q-1)$ gilt.

4. Man setzt $n := p \cdot q$,

5. Man löscht die Zahlen p, q, so daß keine Spur hinterlassen wird.

Damit werden e und n zu den öffentliche Schlüssel und d der Geheimschlüssel.

Chiffrieren: verschlüsselt werden Zahlen $m < n$: $E(m,[e,n]) = m^e \bmod n$

Dechiffrieren: $D(c,[d,n]) = c^d \bmod n$

3.7.3 Begründung der Konstruktion

Lemma 56 *Für RSA-Schlüssel n, e, d (wie oben definiert) gilt:*

$$\forall m < n \quad D(E(m,[e,n]),[d,n]) = m$$

Für den Beweis brauchen wir folgende Mittel aus Algebra:

- Sei $\varphi(n)$ = Anzahl der Zahlen $x < n$, die mit n teilerfremd sind.

  Offensichtlich $\varphi(p) = p - 1$ für eine Primzahl p.
  Für 2 Primzahlen p, q, $p \neq q$, gilt $\varphi(p \cdot q) = (p-1)(q-1)$. (Man kann die Zahlen, die mit pq nicht teilerfremd sind leicht auflisten: $p, 2p, \ldots, p(q-1)$, $q, 2q, \ldots, q(p-1)$ und 1).

- der Satz von Euler sagt, daß $\quad \mathrm{ggT}(x,n) = 1 \Rightarrow x^{\varphi(n)} = 1 \bmod n$.

Beweis von Lemma 56 Sei m der Klartext (d.h. m ist eine Zahl $< n$).
Fall 1 $\mathrm{ggT}(m,n) = 1$:

$$D(E(m,[e,n]),[d,n]) = D(m^e \bmod n,[d,n]) = (m^e \bmod n)^d \bmod n = m^{ed} \bmod n.$$

Außerdem

$$e \cdot d = 1 + x \cdot (p-1)(q-1) = 1 + x \cdot \varphi(n).$$

Deswegen

$$m^{ed} = m \cdot m^{x \cdot \varphi(n)} = m \cdot \left(m^{\varphi(n)}\right)^x = m \cdot 1^x \bmod n$$

also $m^{ed} = m \bmod n$.

Fall 2 $\mathrm{ggT}(m,n) > 1$:
Eine solche Nachrichte sollte nicht verschickt werden, da ein nicht-trivialer Teiler von n die Faktorisierung von n verrät, und dann d gefunden werden kann. Die Wahrscheinlichkeit einer solchen Nachricht beträgt $\frac{p \cdot q - \varphi(pq)}{p \cdot q} \approx \frac{1}{p} + \frac{1}{q}$. Bei $p, q \approx 2^{100}$ ist das $2^{-99} \approx 0$.
Trotzdem (um das Lemma zu beweisen): Ohne Einschränkung der Allgemeinheit gelte $p|m$. Dann $m^{q-1} = 1 \bmod q$, $m^{\varphi(n)} = 1 \bmod q$, $m^{x \cdot \varphi(n)} = 1 \bmod q$. Deswegen $m^{ed} = m^{1 + x \cdot \varphi(n)} = m \bmod q$. Andererseits gilt $p|m$. Also

$$m^{ed} = 0 \bmod p \quad \text{d.h.} \quad m^{ed} = m \bmod p.$$

Nach dem Chinesischen Restklassensatz gilt somit[1]

$$m^{ed} = m \bmod p \cdot q.$$

□

3.7.4 Kommentare zu RSA

- durch Faktorisierung von n kann man d finden. Ist RSA sicher??
  - Das Faktorisierungsproblem ist in NP, aber es ist nicht bekannt, daß es NP-vollständig ist.
  - die besten bekannten Faktorisierungsalgorithmen haben große Laufzeiten, wie z.B. beim Number Field Sieve mit einer erwarteten Laufzeit von:

$$\approx 2^{c \cdot \log n^{\frac{1}{3}} \cdot \log\log n^{\frac{2}{3}}} \quad \text{mit } c \approx 1.9$$

[1] **Chinesicher Restklassensatz**: $q_1, \ldots, q_s$-teilerfremd, $a_1, \ldots, a_s$-beliebige Zahlen. Dann gibt es genau eine Zahl $x \leq q_1 \cdot \ldots \cdot q_s$, so daß $\forall i \leq s : x = a_i \bmod q_i$.

- um Schlüssel zu erzeugen, muß man irgendwie (also z.B. zufällig) zwei große Primzahlen finden. Wie ist das möglich?
  - Es gibt viele Primzahlen: für alle x ist die Anzahl der Primzahlen im Intervall $[1, \ldots, x]$ ungefähr $x/\ln x$. Also beträgt bei 100 Ziffern die Chance, eine Primzahl zu treffen, ungefähr 1/115.
  - Wie kann man bestimmen, ob eine Zahl prim ist? Faktorisierungversuche (s.o.) dauern zu lange. Es gibt jedoch schnelle probabilistische Tests, die schnell und mit hoher Wahrscheinlichkeit die korrekte Antwort geben (Man beachte, daß dies nicht das Faktorisierungsproblem löst!).

3.7.5 Primzahltests

Fermat Primzahltest

Nach dem Satz von Euler (genannt auch Fermat Satz für Primzahlen p) gilt für alle Primzahlen p und alle w mit $0 < w < p$, daß $w^{p-1} = 1 \bmod p$.

Definition: $w < n$ ist ein **Zeuge** für „n ist eine Primzahl“ $\Leftrightarrow w^{n-1} = 1 \bmod n$.

Lemma 57 *Seien w und n teilerfremd. Entweder sind alle Zahlen oder höchstens die Hälfte aller Zahlen $w < n$ Zeugen für n.*

Beweis: Sei $w_1, \ldots, w_t$ die Liste aller Zeugen für n und $w \notin \{w_1, \ldots, w_t\}$, wobei gilt, daß w und n teilerfremd sind. Sei $u_i := w \cdot w_i \bmod n$ für $i \leq t$. Dann gilt:

- $u_i \neq u_j$ für $i \neq j$. Angenommen: Es gilt $i \neq j$ und $u_i = u_j \Rightarrow n | ww_i - ww_j$ $\Rightarrow n | w \cdot (w_i - w_j) \Rightarrow n | (w_i - w_j)$. Da $-n < w_i - w_j < n$ gilt, folgt $w_i = w_j$. Widerspruch zu $i \neq j$.
- ${u_i}^{n-1} = (ww_i)^{n-1} = w^{n-1} \cdot {w_i}^{n-1} = w^{n-1} \neq 1 \bmod n$

Also: für t Zeugen haben wir t nicht-Zeugen gefunden. □

Der Algorithmus - Fermat Primzahltest

Wir führen k-mal den folgenden Test unabhängig voneinander aus:

1. Wähle zufällig ein $w < n$ und berechne $\mathrm{ggT}(w, n)$.

   Ist $\mathrm{ggT}(w, n) > 1$, so ist n *keine* Primzahl.

2. Berechne $u = w^{n-1} \bmod n$.

   Ist $u \neq 1$, so ist n *keine* Primzahl.

Besteht eine Zahl n alle k Tests, so sagt man, daß n prim sei.

Eine Einschränkung: Jede Primzahl besteht diesen Test. Aber es gibt zusammengesetzte Zahlen[2], die den zweiten Teil des Tests ebenfalls immer bestehen (es ist nicht bekannt, wieviele solche Zahlen es gibt!). Trotzdem wird der Fermat Test in der Praxis verwendet (PGP).

[2] Carmichael Zahlen [nach R.D. Carmichael, 1879–1967]; die kleinste ist 561.

Echte Primzahltests

Solovay-Strassen Primzahltest
Er ist ähnlich zum Fermat Test, jedoch wird nicht nur $w^{n-1} \bmod n$ getestet, sondern auch $u = w^{\frac{n-1}{2}} \bmod n$ für das jeweils zufällig gewählte $w < n$.

- Für Primzahlen ist $u \in \{1, -1\}$ und läßt sich u als *Jakobi-Symbol* effizient berechnen.
- Für zusammengesetzte Zahlen ist die Wahrscheinlichkeit, daß $u \in \{1, -1\}$ gilt, höchstens 0.5.

Deswegen ist bei bei einer zusammengesetzten Zahl n und k Versuchen die Wahrscheinlichkeit, daß wir keinen Teiler und keinen nicht-Zeugen finden, höchstens $1/2^k$. Neben den kurz dargestellten gibt es auch noch andere interessante Primzahltests, wie z.B. die von Miller-Rabin, Goldwasser-Kilian oder Adleman-Huang (mit elliptischen Kurven).

3.8 Sicherheitsfragen bei RSA

3.8.1 Sicherheit bei Schlüsseln mit dem gleichen n

Regel: Man sollte nicht zwei private Schlüssel mit demselben Parameter n benutzen.

Theorem 58 *Seien e_A, d_A und e_B, d_B zwei Paare von RSA-Schlüssel, die denselben Parameter n verwenden. Dann kann bei gegebenem e_A, e_B und d_B der fehlende Schlüssel d_A in polynomieller Zeit gefunden werden.*

Beweis:

- Nach der Konstruktion ist $e_B d_B = 1 \bmod \varphi(n)$, also $e_B d_B - 1 = k\varphi(n)$ für irgendein k.
- Finde die maximale Zahl t, die
  - $e_B d_B - 1$ teilt und
  - nur solche Primteiler wie e_A hat.

  t kann ohne Faktorisierung berechnet werden:
  1. Sei $g_0 = e_B d_B - 1$. Berechne $h_0 = \mathrm{ggT}(g_0, e_A)$. h_0 erfüllt die obigen Eigenschaften für t, ist aber noch nicht maximal.
  2. Berechne induktiv: $g_{i+1} = g_i/h_i, h_{i+1} = \mathrm{ggT}(g_{i+1}, e_A)$. Brich die Iteration ab, wenn für ein $i = l$ gilt, daß $\mathrm{ggT}(g_{l+1}, e_A) = 1$. Dann ist $t = \prod h_i$.
- Sei $\alpha = (e_B d_B - 1)/t$. Nach den Eigenschaften von t sind somit α und e_A teilerfremd.
- Berechne mit dem euklidischen Algorithmus Zahlen a und $b > 0$, so daß
$$a \cdot \alpha + b \cdot e_A = 1 \tag{3.4}$$
- e_A und $\varphi(n)$ sind teilerfremd $\Rightarrow$ t und $\varphi(n)$ sind teilerfremd.
  Andererseits gilt: $\varphi(n) | e_B d_B - 1$ und $e_B d_B - 1 = \alpha t$ Also $\varphi(n) | \alpha$. Durch (3.4) erhält man
$$b \cdot e_A = 1 \bmod \varphi(n).$$
  Damit kann b als Dechiffrierungsschlüssel d_A benutzt werden. □

3.8.2 Knacken von RSA versus Faktorisierung

Theorem 59 *Annahmen: sei A ein Algorithmus, der für gegebene Zahl n und public key e einen passenden private key d berechnet. Dann läßt sich aus A ein randomisierter Algorithmus B ableiten, der n faktorisiert und nur wenig zusätzliche Schritte und Speicherplatz benötigt.*

Praktisch bedeutet dies: kann man RSA private keys knacken, so kann man n faktorisieren. Offensichtlich gilt: kann man n faktorisieren, so kann man RSA private keys knacken.

Beweis z. B. im Buch von Salomaa.

Bemerkung: Vielleicht läßt sich der Klartext auch ohne den private key ausrechnen?! Der Satz garantiert also keine endgültige Sicherheit.

3.8.3 Hard-Bit bei RSA

Klasse P und effizient berechenbare Funktionen

In der Komplexitätstheorie gelten die Sprachen aus der Klasse P als *effizient entscheidbare Mengen*. Alles außerhalb P gilt als nicht effizient entscheidbar (berechenbar). Diese weit verbreitete Meinung ist teilweise falsch.
In polynomieller Zeit kann man noch randomisierte Algorithmen anwenden. Eine Fehlerwahrscheinlichkeit von z.B. 2^{-100} ist normalerweise wesentlich kleiner als die Wahrscheinlichkeit eines Erdbebens, das den Rechner zerstört. Wir kümmern uns nicht um Erdbeben, warum sollten wir uns um Fehler des Algorithmus kümmern?

Hard-core Bit

Sei $b : \{0,1\}^* \longrightarrow \{0,1\}$ eine Abbildung. Dann heißt b **Hard-core Bit**, falls für jede randomisierte Turingmaschine M, die in polynomieller Zeit hält, und jedes beliebige Polynom p gilt:

$$\exists\, l_p : \forall x, |x| > l_p : \text{Prob}\left(b(x) = f_M(x)\right) \leq \frac{1}{2} + \frac{1}{p(|x|)}$$

Kommentare zur Definition: Falls M als Alternative einfache eine Münze wirft und nach diesem Ergebnis für 1 oder 0 entscheidet, dann ist die oben genannte Wahrscheinlichkeit schon 0.5. Die Definition sagt, daß es keine wesentlich bessere Maschine gibt.

Kann man zeigen, daß zum Beispiel die Funktion

$$\text{(RSA-Kryptogramm, public key)} \longrightarrow \text{letztes Bit des Klartextes}$$

ein Hard-core Bit bildet, dann wäre die Sicherheit des letzten Bits perfekt. Leider wurde dies noch nicht bewiesen. Man kann trotzdem Aussagen über die Sicherheit des letzten Bits in Zusammenhang mit der Sicherheit von RSA sagen.

Orakel-Maschine

Eine Orakel-Maschine ist eine normale (Turing- oder RAM-) Maschine, die zusätzlich ein *Orakel* besitzt. Das Orakel, das über eine Sprache L entscheidet, funktioniert folgendermaßen: für eine Folge x, die dem Orakel präsentiert wird, entscheidet das Orakel sofort ob $x \in L$ ist und gibt die Antwort zurück.

Die Orakel-Maschine ist nur eine theoretische Konstruktion, die für Überlegungen *„wie wäre es wenn ein effizienter Algorithmus für L existieren würde“* dient. Da das Orakel sofort antwortet, werden bei Orakel-Maschinen nur die Schritte für die Laufzeit betrachtet, die nicht zur Berechnung von L durchgeführt werden.

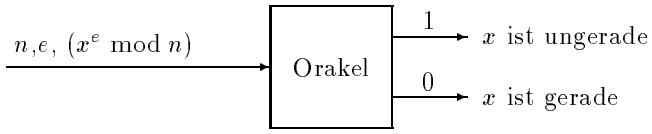


Abbildung 3.12: Orakel für das letzte Bit von RSA

Hard-core Bit bei RSA

Theorem 60 *Es gibt eine Orakel-Turingmaschine M mit dem Orakel für das letzte Bit des Klartextes, die auf Basis des Kryptogramms und des public keys den ganzen Klartext in polynomieller Zeit berechnet.*

Bedeutung des Satzes: Das letzte Bit des Klartextes zu gewinnen, ist genauso schwierig wie das Knacken von RSA. (Man kann auch sagen: Es reicht aus, nur das letzte Bit effizient bestimmen zu können.)

Beweisidee für Satz 60: Wir konstruieren eine Orakel-Maschine, die den Klartext herausfindet. Sei x der Klartext, e der public key und $x^e \bmod n$ das Kryptogramm. Seinen $e, n, (x^e \bmod n)$ bekannt und x gesucht. Annahme: der Parameter n ist ungerade (sonst wäre n sofort faktorisierbar).

Beobachtung 1: Für jeden Klartext y ist es gleichbedeutend die letzten k Bits von y oder oder von $n - y$ herauszufinden ($y = n - (-y) \bmod n$).

Beobachtung 2: Da 2 und n teilerfremd sind, gibt es ein Element $\frac{1}{2}$ im Ring $\mathbb{Z}_n$ (d.h. ein Element a, so daß $2 \cdot a = 1 \bmod n$).

Falls y ($y < n$) gerade ist, dann hat $y/2$ dieselbe binäre Darstellung wie y mit Ausnahme der letzten Null, die gestrichen ist.

Um also die k letzten Bits von y herauszufinden reicht es, die $k-1$ letzten Bits von $y/2$ herauszufinden.

Beobachtung 3: Falls $y < n$ und y ungerade ist, dann ist $n - y$ gerade. Folglich können die letzten k Bits von y aus den k letzten Bits von $n - y$ bzw. $k - 1$ letzten Bits von $(n - y)/2 \bmod n$ berechnet werden.

Beobachtung 4: Aus dem Kryptogramm $y^e \bmod n$ läßt sich das Kryptogramm für $(y/2)$ herausfinden:

$$(y/2)^e = (\tfrac{1}{2} \cdot y)^e = \tfrac{1}{2}^e \cdot y^e.$$

Auf analoge Weise ist das Kryptogramm für $n - y$:

$$(n - y)^e = (-1)^e \cdot y^e = n - (y^e \bmod n) \bmod n.$$

Aufgrund der vier Beobachtungen und mit Hilfe des Orakels für das letzte Klartext-Bit läßt x wie folgt Berechnen:

Orakel-Algorithmus:
Wir haben die Aufgabe alle $N = \log n + 1$ Bits von x zu bestimmen.
Eingabe: e, n, x^e, N.

1. Wenn $N = 0$ ist, dann gib das leere Wort als Ergebnis zurück.

2. Entscheide mit dem Orakel, ob das letzte Bit u von x gleich 1 ist,

3. Falls $u = 0$ ist, dann rufe den Algorithmus rekursiv mit den folgenden Parametern auf: $e, n, x^e/2^e, N-1$.
   Als Ergebnis erhalten wir eine binäre Zahl $d_{N-2} \ldots d_0$. Dann ist $d_{N-2} \ldots d_0 0$ die binäre Darstellung von x.

4. falls $u = 1$ ist, dann rufe den Algorithmus rekursiv mit den folgenden Parametern auf: $e, n, -x^e/2^e, N-1$.
   Als Ergebnis erhalten wir eine binäre Zahl $d_{N-2} \ldots d_0$. Dann ist $d_{N-2} \ldots d_0 0$ die binäre Darstellung von $n - x$, aus der wir $x = n - (n - x)$ berechnen.

□

3.9 Einweg-Funktionen

Eine Funktion $f : \{0,1\}^* \longrightarrow \{0,1\}^*$ heißt **Einweg-Funktion** (zusätzliche Annahme: für jedes x gilt $|f(x)| \approx |x|$) falls:

- f in polynomieller Zeit (deterministisch) berechnet werden kann und
- f schwierig zu invertieren ist, d.h. für jeden randomisierten Algorithmus A und jedes Polynom p existiert ein l_p (eine Konstante, die von p abhängen kann), so daß für alle $n > l_p$:

$$\text{Prob}(f(A(z)) = z) < \frac{1}{p(n)}$$

  wobei z eine Zufallsvariable ist, die uniform in der Menge $\{f(x) : x \in \{0,1\}^n\}$ verteilt ist.

Kommentare zur Definition: Es ist nicht ausgeschlossen, daß $f(A(z)) = z$. Aber die Wahrscheinlichkeit von diesem Ereignis ist sehr sehr klein (eigentlich fast Null). Es ist auch nicht ausgeschlossen, daß für einige Folgen z das Invertieren leicht sein kann. Jedoch für ein zufällig gewähltes z ist das Invertieren schwer!
Es wurde bisher noch von keiner expliziten Funktion bewiesen, daß sie eine Einweg-Funktion ist. Trotzdem gibt es Vermutungen, daß bestimmte Funktionen diese Eigenschaft besitzen. Es kann aber auch passieren, daß eine Einweg-Funktion praktisch unbrauchbar ist, da die Konstante l_p zu groß sein kann.

Anwendungen von Einweg-Funktionen:

- Hashfunktionen,
- Generatoren für zufällige Folgen von Bits,
- Chiffrierfunktionen (es muß noch die Möglichkeit gegeben sein, mit einem Schlüssel zu invertieren).

Diese Konstruktionen haben eine theoretische Begründung (siehe das Buch von Goldreich), sind aber nicht immer praktisch.

3.9.1 Kandidaten für Einweg-Funktionen

Produkt von Primzahlen

Für zwei Primzahlen p, q ist $f(p, q) = p \cdot q$.
Damit entspricht das Invertieren von f der Faktorisierung von $p \cdot q$.

Decodierung von zufälligen linearen Codes

Es kann gezeigt werden, daß fast alle Matrizen (ab einer gewissen Größe) Generatormatrizen von d-fehlerkorrigieren Codierungen sind. Für gegebene Generatormatrix G und Nachricht x berechne den Codewort $f_G(x) = x^T \cdot G$.
Im allgemeinen ist das Invertieren sehr schwierig.

Knapsack-Funktion

$f(x_1, \ldots, x_n, I) = (x_1, \ldots, x_n, \sum_{i \in I} x_i)$.
Die Funktion f zu invertieren scheint schwierig zu sein. (Die effiziente Kryptoanalyse von Knapsack-Chiffren widerspricht nicht dieser Vermutung: Knapsack-Chiffren benutzen nur sehr spezielle Folgen $x_1, \ldots, x_n$.)

RSA-Funktionen

Für zwei zufällig gewählte Primzahlen p, q, $n = p \cdot q$ und e teilerfremd mit $\varphi(n)$ ist $f_{e,n}(x) = x^e \bmod n$ leicht zu berechnen.
Die inverse Funktion $x^e \mapsto x$ gilt als schwierig.
(Da hier eine Familie von Funktionen auftaucht, muß die Definition von Einweg-Funktionen etwas umformuliert werden.)

Rabin Funktion

n soll so gewählt werden wie bei RSA-Funktion. Dann ist $f_n(x) = x^2 \bmod n$.
Es kann gezeigt werden, daß durch Berechnung der quadratischen Wurzeln n faktorisiert werden kann, was als schwierig gilt.

Diskrete Logarithmen

Einfache Funktion: $\exp(x, g, p) \stackrel{\text{def}}{=} g^x \bmod p$
Schwierige Funktion: $(z, g, p) \mapsto x$, so daß $\exp(x, g, p) = z$.

3.10 Einweg-Hashfunktionen

Zur Definition und zu einigen Anwendungen von Einweg-Hashfunktionen siehe auch Kapitel 3.1.1.

Weitere Anwendungen:

Passwörter und Hashfunktionen: Unter UNIX werden die Passwörter als solches nicht gespeichert (zu riskant), sondern nur ihre Hashwerte. Ein login wird folgendermaßen durchgeführt:

1. Der Benutzer A gibt sein Passwort x ein.
2. Unix berechnet $H(x)$ und vergleicht mit dem Wert, der für A gespeichert ist (x wird nicht gespeichert!).

Hashwerte als Beweisstück: Hashwerte werden in einigen US-Bundesstaaten als Beweisstücke vom Gericht anerkannt. Man kann zum Beipiel, um Autorenrechte für ein Programm zu schützen, den Quelltext hashen und den Hashwert in einer Zeitung als private Anzeige veröffentlichen.

Schutz vor Änderungen: Um Dateien nach der Kanalübertragung überprüfen zu können, vergleichen der Empfänger und der Absender die Hashwerte per Telefon (auch als Option bei PGP vorhanden).

3.10.1 Angriffe gegen Hashfunktionen

Wörterbuch-Angriff (dictionary attack)

Das ist eine Methode um zum Beispiel Passwörter in Rechnersystemen zu knacken. Man geht folgendermaßen vor:

1. Berechne für ein großes Wörterbuch (alle deutschen, englischen, ..., hebräischen, ... Wörter, geographische Namen, ... — auch mit kleinen Abweichungen: z.B. Padeborn statt Paderborn) alle Hashwerte (solche Datenbanken können z.B. auf CDROM gespeichert werden).

2. kopiere das File mit den Hashwerten der Paßwörter auf Diskette oder den eigenen Rechner.

3. suche (zu Hause auf dem PC) gleiche Einträge auf der Diskette und auf der CD.

So kann man *schwache* Passwörter finden und dadurch freien Zugang zum System bekommen. Besonders schwache Paßwörter (Geburtstagdatum, obszöne Wörter, ...) wählen die Anfänger.

Verteidigung von UNIX: Die Hashtabelle von Passwörtern enthält für jeden Benutzer zwei Werte:

- eine zufällige Folge r von z.B. 12 Bits und
- ein Hashwert y.

Ist x das richtige Passwort, dann gilt $H(x \circ r) = y$.
Beim Wörterbuch-Angriff muß man die zusätzlichen 12 Bits bei *jedem* Eintrag in Betracht ziehen. Dadurch wächst die Hash-Liste um den Faktor 2^{12}.

Geburtstag-Angriff (birthday attack)

Beispiel: Alice und Bob sollen einen Vertrag elektronisch unterschreiben. Wegen der Textgröße wird nur der Hashwert des Vertrages unterschrieben (so funktionieren digitale Unterschriften). Alice bereitet einen Betrug vor:

1. Alice bereitet erst zwei Versionen des Vertrages vor: Eine Version V_1, die Alice zum Unterschreiben präsentiert, und eine Version V_2, die für Alice profitabel ist.

2. Durch kleine Änderungen in V_1 und V_2 (Zeilentrennungen, Abstände, etc.) ändert Alice die Werte $H(V_1)$ **und** $H(V_2)$. Nach einiger Zeit findet sie V_1' und V_2', entsprechende Versionen von V_1 und V_2, so daß $H(V_1') = H(V_2')$,

3. Bob unterschreibt V_1', d.h. er unterschreibt elektronisch $H(V_1')$,

4. nach einiger Zeit geht Alice zum Gericht mit der Unterschrift von V_2' und verlangt Geld. Obwohl Bob auch einen passenden Text präsentieren kann, kann man nicht herausfinden, wer betrogen hat.

Der Name „Geburtstag-Angriff“ stammt von folgender Situation: die Wahrscheinlichkeit, daß in der Gruppe von k zufälligen Personen jemand den Geburtstag am, sagen wir, 11. Dezember hat, beträgt $k/365$. Dagegen die Wahrscheinlichkeit, daß irgendwelche 2 Personen denselben Geburtstag haben, beträgt

$$1 - \tfrac{364}{365} \cdot \tfrac{363}{365} \cdot \ldots \cdot \tfrac{365-k+1}{365}.$$

Für $k = 20$ beträgt die erste Warscheinlichkeit ca. 0.06 und die zweite ca. 0.44. Ähnliche Berechnungen zeigen, daß Alice viele Versuche spart, wenn sie beide Versionen des Vertrages modifiziert und nicht nur nach einem zu V_1 passenden Vertrag sucht.

Korollar: Die minimale Länge der Hashwerte muß nach Laufzeit des Geburtstag-Angriffs orientieren und nicht nach der für eine vollständige Suche.
Heutzutage verwenden praktische Systeme meistens 128 Bits.

3.10.2 Hashing von langen Nachrichten

Für Hashing werden in der Praxis

- allgemeine Verschlüsselungsverfahren (bzw. die entsprechende Software oder Hardware),
- speziell entworfene Algorithmen (siehe Kapitel 3.10.3)

benutzt. In beiden Fällen ist die Eingabelänge fest! Hashing soll aber für Eingaben beliebiger Länge anwendbar sein.

Lösung: Verkettung von Hashwerten für einzelne Blöcke. Angenommen wir haben eine Funktion $E : \Sigma^{2k} \to \Sigma^k$ (ein Beispiel für E ist der DES-Verschlüsselungalgorithmus mit $\Sigma = \{0, 1\}$ und $k = 64$; die ersten k Bits sind die Schlüsselbits, die restlichen k Bits sind die Nachrichtenbits). Eine lange Nachricht M wird auf Blöcke der Länge k geteilt: Sei $M = M_1 M_2 \ldots M_t$ (M wird zusätzlich um die Länge von M und einige padding-Bits erweitert.)

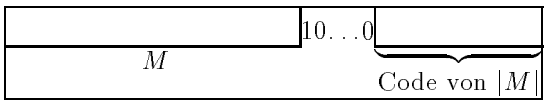


Abbildung 3.13: Erweiterung von M

Rabin Schema

Sei:

$$\begin{aligned} H_0 &= IV \text{ (=Initialwert)} \\ H_i &= E(M_i, H_{i-1}), \quad \forall i \leq t \\ H_t &= \text{Hashwert}(M) \end{aligned}$$

Dabei ist IV eine Folge, die irgendwie initialisiert werden muß (und nicht geheim gehalten wird).

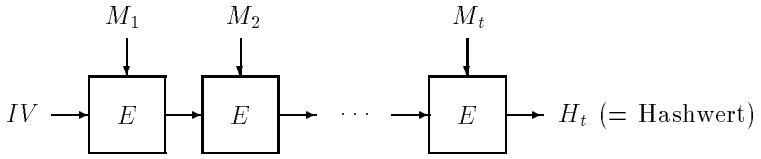


Abbildung 3.14: Rabin Schema

Meet in the middle-Angriff

Annahme: die Funktion E ist eine Chiffrierfunktion. Der folgende Angriff heißt *meet in the middle-Angriff*, dessen Ziel es ist, eine Folge mit einem beliebig ausgewählten Hashwert x zu erzeugen.

Vorgehensweise:

1. für jede Nachricht N_1 der Länge $n/2$ berechnen wir den Hashwert, der nach dem Rabin Schema aus dem letzten Block von N_1 herauskommt. Sei Z_1 die Menge aller berechneten Hashwerte.

2. für jede Nachricht N_2 der Länge $n/2$, die aus Blöcken $Y_1, \ldots, Y_l$ besteht, berechnen wir induktiv:

$$\begin{aligned} H_l &= x \\ H_{i-1} &= D(Y_i, H_i), \quad \forall i \leq l \end{aligned}$$

   wobei D die Dechiffrierungsfunktion ist, die E entspricht. Sei Z_2 die Menge aller berechneten Hashwerte H_0.

3. Wir suchen einen gemeinsamen Element in der Mengen Z_1 and Z_2. Falls $y \in Z_1 \cap Z_2$ ist, können wir die Folgen N_1 und N_2, die y entsprechen, konkatenieren. Die konstruierte Folge hat den Hashwert x.

Stärkeres Schema

Man kann das Schema von Rabin etwas modifizieren, damit das Zurückrechnen von Hashwerten erschwert wird:

$$\begin{aligned} H_0 &= IV \text{ (=Initialwert)} \\ H_i &= E(M_i, H_{i-1}) \oplus H_{i-1}, \quad \forall i \leq t \\ H_t &= \text{Hashwert}(M) \end{aligned}$$

Wieder muß IV initialisiert werden.

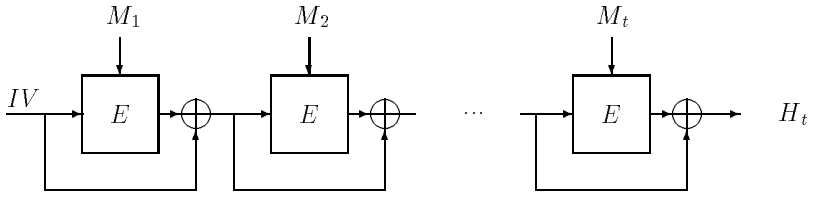


Abbildung 3.15: ein verbessertes Schema gegen meet in the middle-Angriff

Noch ein Schema – das Matyas-Meyer-Oseas Schema –, das als Standard anerkannt wurde und als sicherer gilt:

$$\begin{aligned} H_0 &= IV \text{ (=Initialwert)} \\ H_i &= E(s(H_{i-1}), M_i) \oplus M_i, \quad \forall i \leq t \\ H_t &= \text{Hashwert}(M) \end{aligned}$$

dabei ist s eine Funktion, die aus den Kryptogrammen die Schlüssel erzeugt, wobei s dabei keine kryptographische Rolle spielt.

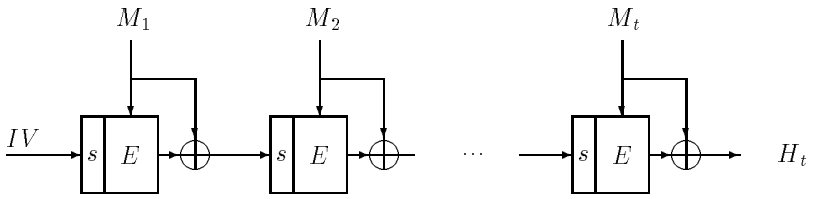


Abbildung 3.16: Matyas-Meyer-Oseas Schema

3.10.3 MD5 Hashfunktion

Eigenschaften von MD5 Hashfunktion:

- Autor: Ron Rivest
- MD5 : $\{0,1\}^{512} = \{0,1\}^{16 \cdot 32} \to \{0,1\}^{128} = \{0,1\}^{4 \cdot 32}$
- MD5 ist einfach zu implementieren, da nur grundlegende Operationen auf 32-Bit Blöcken durchgeführt werden.
- MD5 hat bis jetzt (zahlreiche) kryptoanalytische Angriffe bestanden.
- Der Secure Hash Algorithm – ein amerikanischer Hashfunktion-Standard – ist sehr ähnlich zu MD5.
- MD5 ist im PGP-Paket enthalten.

MD5 besteht aus 4 ähnlichen Runden. Bei jeder Runde werden die folgende Grundoperationen auf 32-Bit Blöcken bitweise verwendet:

- $F(X,Y,Z) = X \cdot Y \text{ OR } (\neg X) \cdot Z$
- $G(X,Y,Z) = X \cdot Z \text{ OR } Y \cdot (\neg Z)$
- $H(X,Y,Z) = X \oplus Y \oplus Z$
- $I(X,Y,Z) = Y \oplus (X \text{ OR } \neg Z)$

Auf Basis der Funktionen F, G, H definiert man die Prozeduren FF, GG, HH und II:

$$FF(a,b,c,d,M_j,s,t_i) \equiv a := b + \text{Shift}_s(F(b,c,d) + M_j + t_i)$$

Für GG, HH, II ersetzt man F entsprechend durch G, H und I.
Sei M_j der j-te Block der Eingabe und

$$t_i = \lfloor 2^{32} \cdot |\sin i| \rfloor \text{ binär dargestellt}$$

Der Algorithmus verwendet die Register a, b, c, d. Sie werden mit den Hexadezimalzahlen (32 Bit)

- a=`0X01234567`
- b=`0X89ABCDEF`
- c=`0XFEDCBA98`
- d=`0X76543210`

initialisiert.

Runde 1: Sequentiell werden die folgenden Operationen durchgeführt

$$\left.\begin{array}{l} FF(a,b,c,d,M_0\ ,\ 7,t_1\ ) \\ FF(d,a,b,c,M_1\ ,12,t_2\ ) \\ FF(c,d,a,b,M_2\ ,17,t_2\ ) \\ \ldots \\ FF(b,c,d,a,M_{15},22,t_{15}) \end{array}\right\} \text{16 Operationen}$$

Dabei wird die Reihenfolge der ersten vier Argumente (a,b,c,d) immer um eine Position rotiert und die Shifts (vorletztes Argument) bilden eine periodische Folge der Länge 4 mit $7,12,17,22$.

Runde 2: Die Prozedur GG findet (anstatt FF) Anwendung und die periodische Folge der Länge 4 wird durch $5,9,14,20$ gebildet. Außerdem fließt in der i-ten Runde, $i = 0..15$ der Eingabeblock $M_{1+5i \bmod 16}$ anstelle von M_i ein, sowie der Wert t_{16+i} anstelle von t_i.

Runde 3: HH und $4,11,16,23$ sowie $M_{5+3i \bmod 16}$ und t_{32+i}

Runde 4: II und $6,10,15,21$ sowie $M_{0+7i \bmod 16}$ und t_{48+i}

Der Hashwert ist dann $a \circ b \circ c \circ d$.

3.11 Pseudozufällige Folgen

Zufällige Folgen werden für viele kryptographische Zwecke benötigt. Einige Beispiele:

- Erzeugung von RSA Schlüsseln,
- Erzeugung von digitalen Unterschriften (siehe Kapitel 3.13),
- Randomisierung in vielen Protokolle (siehe z. B. Kapitel 3.15,3.12),
- Fluß-Chiffren (siehe unten).

Da die schnelle Erzeugung von zufälligen Folgen technisch schwierig ist, verwendet man *pseudozufällige Folgen*. Jede solche Folge ist durch einen Parameter, den so genannter *seed*, definiert. Dann ist das i-te Element s_i in der pseudozufälligen Folge definiert als Funktion von dem seed x, dem Index i und den Werten $s_1, \ldots, s_{i-1}$. Da der seed eine kurze Folge ist, kann er einfach und zuverlässig durch Messen von Zeitintervalen zwischen Tastatur-Anschlägen erzeugt werden.

3.11.1 Fluß-Chiffren

Bei Fluß-Chiffren werden nicht größere Blöcke sonder die Bits einzeln on-line chiffriert und unverzüglich gesendet.
Um gute Fluß-Chiffren zu erzegen, kann man (außer beim Cypher Feedback Mode) pseudozufällige Folgen benutzen:

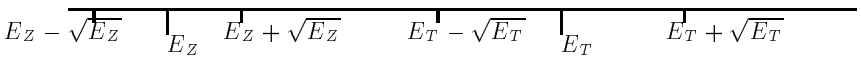
1. für gegebenen Schlüssel x erzeugt man die pseudozufällige Bitfolge $s_1, s_2, s_3 \ldots$ mit dem seed x,
2. für eine Nachricht $a_1a_2a_3 \ldots$ verschickt man die Bits $s_1 \oplus a_1, s_2 \oplus a_2, s_3 \oplus a_3, \ldots$.

Die zufälligen Bits werden dabei nach Bedarf on-line erzegut. Falls das Bit a_i zu verschicken ist, so berechnet der Sender unverzüglich s_i und sendet $s_i \oplus a_i$.

3.11.2 Anforderungen an pseudozufällige Folgen

- Ohne das Wissen von x läßt sich s_i (praktisch) nicht von $s_1, \ldots, s_{i-1}$ ableiten.
- Die obige Anforderung kann man mehr streng mathematisch formulieren:
  Es gibt keine randomisierte Turing Maschine, die in polynomieller Zeit mit Erfolgswahrscheinlichkeit mindestens $0.5 + 1/p(i)$ ($p()$ ist ein Polynom) entscheiden kann, ob die Eingabefolge $s_1 \ldots s_i$ durch einen echten Zufallsgenerator oder einen gegebenen Algorithmus erzeugt wurde (mit einem unbekannten seed).

  Motivation: Bei $n = p(i)^4$ Versuchen, ist der Erwartungswert an richtigen Entscheidungen der Turingmaschine $E_T = n \cdot (0.5 + 1/p(i)) = 0.5 \cdot n + n/p(i)$. Im Gegensatz hierzu ist der Erwartungswert einer rein zufälligen Entscheidung $E_Z = 0.5 \cdot n$. Da beide Versuche Bernoulii-Experimente sind, ist mit hoher Wahrscheinlichkeit die beobachtete Anzahl im Bereich Wurzel des Erwartungswertes um den Erwartungswert herum. Somit kann mit hoher Wahrscheinlichkeit bei einer Beobachtung von n Versuchen das richtige Ergebnis ermittelt werden, da $\sqrt{E_T} + \sqrt{E_Z} \leq 2\sqrt{p(i)^4} = 2p(i)^2 \leq p(i)^3 = n/p(i)$.



Pseudozufällige Folgen, die man außerhalb der Kryptographie benutzt, besitzen normalerweise diese Eigenschaften nicht. Insbesondere die Generatoren, die auf Kongruenzen basieren (z.B. $s_i = a \cdot s_{i-1} + b \bmod m$) sind für kryptographische Zwecke unbrauchbar.

3.11.3 Erzeugung von pseudozufälligen Folgen

Linear Feedback Shift Register

Das *Linear feedback shift Register* (kurz: LFSR) wird im folgenden schematisch dargestellt:

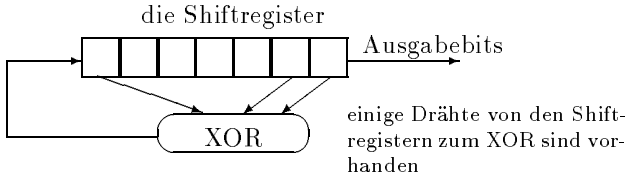


Abbildung 3.17: Erzeugung von pseudozufälligen Folgen mit Shiftregistern

- Zuerst wird der Schlüssel im Shiftregister gespeichert.
- Bei jedem Schritt wird der Wert von XOR berechnet und zum linkesten Register verschickt, um Platz zu machen werden alle Werte im Schiftregister um eine Position nach rechts geshiftet. Das rechteste Bit ist die Ausgabe der entsprechenden Iteration.

LSFR sind sehr schnell, oft benutzt, aber kryptographisch miserabel. Ein Ausweg ist, den Feedback nicht durch XOR sondern durch kompliziertere Funktionen zu berechnen; den Feedback nicht nur auf die linkeste Position zu schicken sondern mehrere LSFR gleichzeitig anzuwenden (hierarchisch, einige dienen als Switches, usw.). Viele solche Generatoren wurden vorgeschlagen und danach geknackt.

Blum-Micali Generator

Der Generator basiert auf der Vermutung, daß diskrete Logarithmen vermutlich sehr schwierig zu berechnen sind. Das heißt, die Funktion $x \mapsto a^x \bmod p$ für zwei Primzahlen a und p ist vermutlich eine Einweg-Funktion.
Sei a und p zwei Primzahlen, wobei p ungerade und groß genug ist. Sei $x_0 < p$ eine zufällig gewählte Zahl, und $x_{i+1} = a^{x_i} \bmod p$ für beliebiges i. Dann setze

$$s_i = \begin{cases} 1 & \text{wenn } x_i > (p-1)/2 \\ 0 & \text{sonst} \end{cases} .$$

RSA Generator

Der RSA Generator basiert auf der Vermutung, daß das letzte Bit von RSA-Codes ein Hard-Bit ist. Betrachte also einen RSA-Code mit Schlüsseln n, e, d. Sei $x_0 < n$ zufällig gewählt und $x_{i+1} = x_i^e \bmod n$ Dann ist s_i das rechteste Bit von x_i.

Blum-Blum-Shub Generator

Der Blum-Blum-Shub Generator ist eine Vereinfachung des RSA-Generator. Wir nehmen $n = p \cdot q$, wobei p, q Primzahlen sind und $p = q = 3 \bmod 4$. Sei x_0 teilerfremd zu n und $x_{i+1} = x_i^2 \bmod n$. Wiederum ist s_i das rechteste Bit von x_{i+1}. Durch die Einschränkungen ist dieser Generator jedoch viel schneller als der RSA-Generator. Andererseits kann man zeigen, daß Blum-Blum-Shub Generatoren so sicher sind wie RSA-Generatoren.

Generatoren durch Chiffrieralgorithmen

Man kann durch gute Chiffrieralgorithmen pseudozufällige Zeugen erzeugen. Zum Beispiel kann man mit einem DES-Chip schnell lange Folgen erzeugen: wir nehmen ein File, das am Anfang ein paar zufällige Zeichen enthält und sonst Blanks. Dann chiffriert man das File durch CBC mit einem Geheimschlüssel, der als seed dient.

3.12 Authentifikation

Neben Passwörtern gibt es auch andere Methoden zur Authentifikation in Rechnersystemen.

3.12.1 Chipkarten

Die Chipkarten dienen zu Benutzerauthentifikation. Dazu benötigt man

- auf der Chipkarte einen Schaltkreis zur Berechnung einer Einweg-Funktion f,
- einen im Chip gespeicherten Geheimschlüssel k,
- eine im Chip gespeicherte Kundennummer und
- einen nicht gespeicherten PIN-Schlüssel.

Die Authentifikation erfolgt mit dem „challenge and response“-Protokoll:

- die Zentrale (der Rechner, Geldautomat, ...) kennt den Schlüssel k von der Chipkarte.
- Schritt 1: der Benutzer gibt den PIN-Schlüssel ein. Nur ein passender PIN aktiviert die Karte.

- Schritt 2: Die Kundennummer des Benutzer wird zu Zentrale übermittelt (aber **nicht** der PIN oder der Geheimschlüssel aus dem Chip).
- Schritt 3: die Zentrale generiert eine zufällige Folge r und verschickt sie an den Benutzer.
- Schritt 4: die Chipkarte des Benutzers berechnet $f(k, r)$ und schickt das Ergebnis an die Zentrale.
- Schritt 5: die Zentrale ermittelt gleichfalls $f(k, r)$. Sind die Ergebnisse gleich, dann wird der Benutzer anerkannt.

Enthält die Karte einen falschen Schlüssel k', dann ist mit sehr großer Wahrscheinlichkeit $f(k, r) \neq f(k', r)$.

Nachteile:

- Heutzutage sind die Chips auf Chipkarten (noch) zu klein, um kompliziertere Funktionen auszurechnen zu können.
- Die Karte und die Zentrale verfügen über denselben Schlüssel (wenig Sicherheit bei unehrlichen Banken, da Duplikate der Karte ohne Wissen des Kunden hergestellt werden können).

Vorteile:

- Keine Übermittlung von Geheimdaten zwischen dem Benutzer und der Zentrale.
- Der Geheimschlüssel kann nicht so einfach abgelesen werden wie bei Magnetkarten.

Chips, die public key-Kryptogramme erstellen könnten, wären viel besser geeignet (die Zentrale würde nur die öffentlichen Schlüssel speichern).

3.12.2 Interaktive Beweise und Zero-Knowledge-Protokolle

Interaktive Beweise sollen ein Analogon aus dem „wirklichen Leben“ wiederspiegeln: Es ist normalerweise einfacher sich mit jemandem über seinen Beweis zu einer Sache zu unterhalten (und damit den Beweis zu akzeptieren), als sich einen aufgeschriebenen Beweis durchzulesen, der ja auf *alle eventuell* auftretenden Fragen schon eine Lösung parat haben muß.

Somit kommt man zu dem folgenden formalen Gerüst: Gegeben sind zwei Turingmaschinen V (Verifier) und P (Proofer). Die Turingmaschinen haben ein gemeinsames Read-Only Eingabeband, jeweils ein Write-Only Ausgabeband, daß jedoch vom anderen gelesen werden kann (auf diese Art und Weise können sich die beiden Turingmaschinen unterhalten) und jeweils ein privates Arbeitsband. Weiterhin sind die folgenden Einschränkungen gegeben:

- Der Verifier V ist eine randomisierte polynomialzeit-Turingmaschine
- Der Proofer P ist eine zeit- und platzunbeschränkte Turingmaschine
- P und V wechseln sich bei Berechnung ab (d.h. nicht beide Turingmaschinen sind gleichzeitig aktiv), wobei V die Berechnung startet (d.h. als erstes aktiv ist)
- Die Länge und Anzahl der ausgetauschten Nachrichten ist polynomiell in der Eingabelänge beschränkt

- V kann jederzeit seine Berechnung in einem akzeptierenden oder verwerfendem Zustand beenden (dann wird P auch nicht mehr aktiv)

Eine Sprache L erlaubt einen interaktiven Beweis, wenn ein Verifier V existiert, so daß:

1. Ein Proover P^* exisiert, so daß (V, P^*) alle Worte aus der Sprache (mit hoher Wahrscheinlichkeit) akzeptiert
2. Für alle Proover P gilt, däs (V, P) alle Worte, die nicht in der Sprache sind, (mit hoher Wahrscheinlichkeit) verwirft; d.h. kein Proofer den Verifier davon überzeugen kann, daß ein Wort, was in Wirklichkeit *nicht* in der Sprache ist, *doch* in der Sprache liegt.

Beispiel: Jede Sprache aus NP erlaubt einen interaktiven Beweis, z.B. Graphisomorphismus. Bei Eingabe G_1 und G_2 erwartet der Verifier V vom Proofer P eine Permutation π der Knoten von G_1. Erhält er keine Permutation, so verwirft der Verifier. Ist $\pi(G_1) = G_2$, so akzeptiert der Verifier, andernfalls verwirft er.
Es existiert ein Proofer P^*, der den Verifier V für zwei isomporphe Graphen davon überzeugen kann, daß sie isomorph sind, indem er die Permutation der Knoten berechnet (es gibt dafür einen exponentiellen jedoch keinen (bekannten) polynomiellen Algorithmus). Andererseits kann jeder Proofer bei Eingabe von zwei nichtisomorphen Graphen dem Verifier schicken was er will, der Verifier wird nie davon zu überzeugen sein, daß die Graphen isomorph sind, d.h. der Verifier wird die Eingabe niemals akzeptieren,

Es ist manchmal notwendig zu beweisen, daß man über geheime Informationen verfügt ohne sie zu veröffentlichen. Übertragen auf interaktive Beweise bedeutet dies, daß der Verifier von einer Tatsache überzeugt wird (aus Wahrscheinlichkeitsargumenten), jedoch nicht in die Lage ist selbst als Proofer zu fungieren, nachdem er überzeugt wurde (der Verifier ist nach der Überzeugungsarbeit genauso schlau wie vorher). Eine solche Art der Beweisführung wird zero-knowledge Beweis genannt.

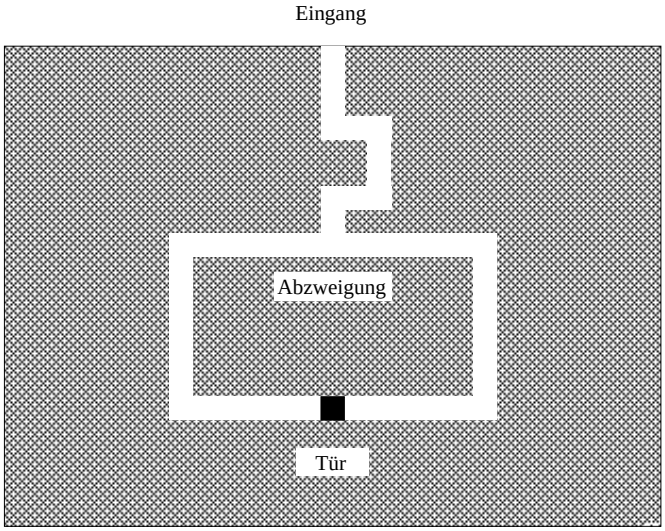


Abbildung 3.18: Die Idee von zero-knowledge Beweisen

Die häufigste Idee des zero-knowledge Beweises ist durch Abbildung 3.18 illustriert. In dem Labyrinth gibt es eine Tür, die Alice vermutlich immer öffnen und abschliessen kann. Folgendermaßen kann Alice beweisen, daß das so ist:

Der Beweis enthält k Phasen. In der Phase i stehen Alice und Bob am Eingang des Labyrinths. Alice geht hinein und geht zur rechten oder linken Seite der Tür (ohne das Bob weiß, wie Alice sich entschieden hat). Dann kommt Bob zu dem Abzweig, wirft eine Münze und verlagt je nach dem Ergebnis des Münzwurfes, daß Alice von der rechten oder linken Seite aus dem Labyrinth herauskommt.
Hat Alice die Möglichkeit (kennt sie die Methode) die Tür zu passieren, dann ist es kein Problem den Anweisungen von Bob zu folgen. Falls Alice keine solche Möglichkeit hat, dann ist die Wahrscheinlichkeit, einer Anweisung folgen zu können, genau 50%. Die Wahrscheinlichkeit, daß es bei k unabhängigen Versuchen jedesmal gelingt, den Anweisungen zu folgen, ohne die Möglichkeit zu haben, die Tür zu passieren, beträgt $1/2^k$. Es reicht also z.B. 30 Tests zu machen, um (absolut) sicher zu sein.

Zero-knowledge Beweis für diskrete Logarithmen

Sei p eine Primzahl, x teilerfremd zu p, und $a^x = b \bmod p$. Alice und Bob kennen p, a, b. Zusätzlich kennt Alice die Zahl x und will Bob überzeugen, daß sie x kennt. Die Berechnung von diskreten Logarithmen gilt als schwierig, weswegen Alice x nicht im Laufe des Beweises einfach ausrechnen kann. Sie muß x schon vorher kennen.

Protokoll:

1. Alice generiert t zufällige Zahlen $r_1, \ldots r_t$.
2. Alice berechnet $h_1 = a^{r_1} \bmod p, \ldots, h_t = a^{r_t} \bmod p$ und präsentiert Bob diese Zahlen.
3. Alice und Bob werfen zusammen t mal die Münze und generieren dadurch t Bits $b_1, \ldots, b_t$.
4. Sei $j := \max\{1, \ldots, t \mid b_j = 1\}$. Für $i \leq t$ übermittelt Alice
$$y_i = \begin{cases} r_i & \text{falls } b_i = 0 \\ r_i - r_j & \text{falls } b_i = 1 \end{cases} .$$
5. Für jedes i überprüft Bob die übermittelte Zahle y_i:
$$a^{y_i} = \begin{cases} h_i \bmod p & \text{falls } b_i = 0 \\ h_i / h_j \bmod p & \text{falls } b_i = 1 \end{cases} .$$
6. Für jedes i mit $b_i = 1$ muß Alice $z_i = x - r_i \bmod p - 1$ übermitteln.
7. Bob überprüft für diese i, ob $a^{z_i} = b/h_i \bmod p$ ist.

Die letzte Gleichung muß gelten, falls Alice nicht betrügt: $a^{z_i} = a^{l \cdot \varphi(p) + x - r_i} = a^x / a^{r_i} = b/h_i \bmod p$. Durch die Schritte 4 und 5 wird gewährleistet, daß Alice die h_i nach dem Protokoll erzeugen muß. Der Schrift 6 kann nur dann korrekt ausgeführt werden, wenn Alice x kennt (falls Alice die Zahl x nicht kennen würde, dann wäre die Berechnung von z_i gleichzeitig eine Berechnung von x). Aus z_i kann Bob x gewinnen, falls er r_i kennt. Aber er kennt nur h_i und die Berechnung von r_i aus h_i ist wieder eine Berechnug eines diskreten Logarithmus. Deswegen wird das Geheimnis nicht verraten!

3.12.3 Fiat-Shamir Protokoll

Beim Fiat-Shamir Protokoll authentifiziert sich Alice gegenüber Bob über einen elektronischen Kanal. Eigenschaften des Protokolls:

- Basiert auf der vermuteten Schwierigkeit quadratische Wurzel modulo n zu berechnen (falls $n = p \cdot q$ für zwei unbekannte Primzahlen p und q).
- Alice verfügt über mehr Informationen als Bob, wodurch dieses Protokoll gleichzeitig in beiden Richtungen benutzt werden kann: der Bank überzeugt sich, daß der Kunde tatsächlich vor dem Bankautomat steht; der Kunde überzeugt sich, daß das tatsächlich ein echter Geldautomat ist (und kein Dummy, der nur PINs und Codes aus den Karten abliest).

Schlüsselerzeugung:

- Die Schlüsselzentrale wählt zwei große Primzahlen p und q.
- Die Zentrale veröffentlicht $n = p \cdot q$.
- Alice erhält von der Zentrale eine geheime Zahl s.
- Die Zentrale veröffentlicht die Zahl $v = s^2 \bmod n$.

Alice muß wenn nötig beweisen können, daß sie s kennt ohne s zu verraten. (Ein Beweis, bei dem Alice s verrät, kann nur einmal durchgeführt werden, danach kann jeder s präsentieren.)

Authentifikation Die folgende Schritte werden mehrmals wiederholt:

Schritt 1: Alice wählt zufällig eine Zahl r (teilerfremd zu n), berechnet $x = r^2 \bmod n$ und schickt x an Bob.

Schritt 2: Bob wählt zufällig $b \in \{0, 1\}$ und sendet b an Alice.

Schritt 3: Alice muß folgende Antwort schicken:

$$y = \begin{cases} r \cdot s \bmod n & \text{falls } b = 1 \\ r \bmod n & \text{falls } b = 0 \end{cases} .$$

Schritt 4: Bob überprüft, ob

$$y^2 = \begin{cases} x \cdot v \bmod n & \text{falls } b = 1 \\ x \bmod n & \text{falls } b = 0 \end{cases} .$$

Sicherheits des Protokolls: Falls Alice die Zahl s nicht kennt, kann sie $r \cdot s$ nicht erzeugen. Aber die Wurzel von $x \cdot v$ ist $r \cdot s$. Bei $b = 1$ würde man einen Betrüger entlarven können.
Der Betrüger kann sich jedoch wehren und x geziehlt wählen: $x = y^2/v \bmod n$. Mit $b = 1$ würde der Betrüger den Test überstehen. Bei $b = 0$ müßte er jedoch die Wurzel von x präsentieren. Dies ist genauso schwierig, wie die Wurzel von v zu bestimmen, also den Schlüssel s zu knacken.
Bei einem einzelnen Test hat ein Betrüger eine Chance von 50%, den Test zu bestehen. Bei 20 unabhängigen Tests beträgt diese Wahrscheinlichkeit $1/2^{20} \approx 0$.

3.13 Digitale Unterschriften

Was ist eine *digitale Unterschrift*?

- Alice hat einen private key k_E (geheim) und public key k_D (zugänglich).
- die Nachricht M soll unterschrieben werden: Alice erzeugt aus M und k_E eine Kryptogramm, welches sie an den Empfänger weitergibt.
- Der Empfänger entchiffriert das Kryptogramm mit k_D. Das Ergebnis muß beweisen, daß Alice die Nachricht M unterschrieben hat.

Die Originalnachricht kann entweder im Klartext übermittelt werden oder auch geheim gehalten werden.

Lösungsvorschläge:

- Durch Einweg-Hashfunktionen. Dadurch können die Nachrichten im Klartext für alle zugänglich sein.
- Durch public key-Kryptogramme (z.B. durch RSA).

3.13.1 ElGamal digitale Unterschriften

ElGamal hat folgendes Protokoll vorgeschlagen:

Vorbereitung: Alice wählt eine Primzahl p, und zwei zufällige Zahlen $g, x < p$. Alice berechnet $y = g^x \bmod p$. Dann veröffentlicht sie g, p, y als public key.

Unterschreiben: Um eine Nachricht M zu unterschreiben, geht Alice folgendermaßen vor:

1. Alice wählt ein k, das teilerfremd zu $p-1$ ist.
2. Alice berechnet $a = g^k \bmod p$.
3. Da k und $p-1$ teilerfremd sind, gibt es t, so daß $k \cdot t = 1 \bmod p-1$. Somit gibt es auch ein b, so daß $M - xa = kb \bmod p-1$. Alice berechnet zuerst t mit dem euklidischen Algorithmus und damit dann b.
4. Alice präsentiert a und b als Unterschrift von M.

Verifikation der Unterschrift: Beachte, daß $y^a \cdot a^b = g^{ax} \cdot g^{kb} = g^M \bmod p$. Bob überprüft, ob $y^a \cdot a^b = g^M \bmod p$.

3.13.2 DSA

DSA ist ein amerikanischer Public Key Standard for Digital Signature (**D**igital **S**ignature **A**lgorithm, auch DSS genannt – Digital Signature Standard). Einige Eigenschaften:

- USA Standard, aber RSA ist populärer und älter.
- Verdächtig (Falltüren könnten durch die NSA eingebaut worden sein).
- DSA kann nur für digitale Unterschriften benutzt werden.
- Fortschritte bei Berechnungen von diskreten Logarithmen können DSA gefährden (die Schlüsselgröße ist vermutlich zu klein).
- DSA ist eine Variante von anderen Algorithmen (gemeinsame Elemente mit ElGamal sind deutlich erkennbar).

- DSA unterschreibt nicht die Nachricht, sondern nur den Hashwert davon. Der Autor muß nicht sofort verraten (oder nicht allen) was sie/er unterschrieben hat.
- DSA benutzt als Unterprozedur eine Einweg-Hashfunktion H, die auch standardisiert ist.

Schlüsselerzeugung

Man wählt zufällig folgende Zahlen:

- p, eine Primzahl mit L Bits, $512 \leq L \leq 1024$ und $64|L$,
- q, einen Primfaktor von $p-1$ mit 160 Bits,
- h, $0 < h \leq p-1$, so daß $g = h^{\frac{p-1}{q}} \neq 1 \bmod p$,
- x, $x < q$, wodurch $y = g^x \bmod p$ definiert ist.

Dann ist der private key x, der public key y. p, q, g sind allgemein bekannte Parameter.

Unterschreiben einer Nachricht m

1. A wählt eine zufällige Zahl $k < q$.
2. Sei $r := (g^k \bmod p) \bmod q$ und $s := \frac{H(m)+x\cdot r}{k} \bmod q$.
3. Dann ist (s, r) die Unterschrift.

Überprüfung der Unterschrift

1. $w := \frac{1}{s} \bmod q$
2. $u_1 := H(m) \cdot w \bmod q$
3. $u_2 := r \cdot w \bmod q$
4. $v := ((g^{u_1} \cdot y^{u_2}) \bmod p) \bmod q$
5. Falls $v = r$ ist, dann ist die Unterschrift korrekt.

Begründung der Konstruktion

Nach dem Satz von Euler gilt $g^q = h^{p-1} = 1 \bmod p$ und somit $g^z = g^{z \bmod q} \bmod p$. Die folgenden Rechnungen sind modulo p durchgeführt:

$$g^{u_1} \cdot y^{u_2} = g^{H(m)\cdot w} \cdot g^{x\cdot r\cdot w} = g^{(H(m)+x\cdot r)\cdot w} = g^{(H(m)+x\cdot r)\cdot \frac{k}{H(m)+x\cdot r}} = g^k$$

Also ist $(g^{u_1} \cdot y^{u_2} \bmod p) \bmod q = (g^k \bmod p) \bmod q = r$.

3.13.3 Blinde Unterschriften

Die Idee ist folgende: Alice kommt mit einem Dokument zu dem Notar, damit er bestätigt, daß Alice im Besitz des Dokuments ist. Alice will jedoch dem Notar das Dokument nicht zeigen. Sie steckt deshalb das Dokument in einem geschlosenem Umschlag zusammen mit einem Stück Blaupapier. Dann unterschreibt der Notar *auf dem Umschlag*. Nachher kann Alice den Umschlag öffnen und die Unterschrift *auf dem Dokument* zeigen.
Zur Erzeugung von blinden elektronischen Unterschriften kann man das folgende auf RSA basierende Protokoll verwenden. Alice will, daß Bob eine Nachricht m blind unterschreibt. Bob hat public key e und private key d zur RSA-Codierung modulo n. Ablauf des Protokolls:

1. Alice *blinds* die Nachricht m: sie wählt eine zufällige Zahl $k \leq n$ und berechnet $t = m \cdot k^e \bmod n$. Dann sendet sie t an Bob.
2. Bob chiffriert t mit seinem private key: $s = t^d \bmod n$.
3. Da $t^d = (m \cdot k^e)^d = m^d \cdot k^{ed} = m^d \cdot k \bmod n$ ist, kann Alice leicht aus s die signierte Nachricht m^d berechnen.

3.13.4 Subliminal Channel

Digitale Unterschriften enthalten eine zufällige Komponente, damit man ein Dokument mehrmals unterschreiben kann (erhöht die Sicherheit). Andererseits kann diese zufällige Komponente dazu benutzt werden, unbemerkt geheime Nachrichten zu senden. Diese Möglichkeit nennt man **subliminal channel**.
Beispiel: Ein Agent verschickt streng geheime Informationen und versteckt sie dazu in Telebanking-Aufträgen. Solch eine Möglichkeit läßt sich nicht vermeiden, da Kryptogramme und zufällige Folgen sich nicht unterscheiden lassen.
Es gibt Methoden subliminal channel in DSA und ElGamal zu realisieren.

ElGamal Protokol für subliminal channel

Bob möchte unbemerkt Alice wichtige Informationen übermitteln. Bob sendet harmlose Briefe, die mit ElGamal unterschrieben worden sind. Alice hat von Bob einen weiteren Geheimschlüssel, mit dem sie die geheime Nachricht aus den Unterschriften entziffern kann.

Verlauf des Protokolls:

1. Wie gewöhnlich wählt Bob eine Primzahl p und zufällige Zahlen $g, r < p$. Bob berechnet $K = g^r \bmod p$ und veröffentlicht K, g, p.
2. Bob verrät Alice die Zahl r.
3. Um eine Nachricht M versteckt in der Unterschrift von M' zu senden, macht Bob folgendes (notwendig ist daß M, M', p und $M, p-1$ teilerfremd sind):
   (a) Bob berechnet $A = g^M \bmod p$
   (b) Bob findet ein B so daß gilt
   $$M' = r \cdot A + M \cdot B \bmod p - 1$$
   (c) Bob sendet die ElGamal Unterschrift A, B an Alice.
   (d) Alice berechnet nach der Überprüfung, daß die Unterschrift echt ist (nach ElGamal-Schema),
   $$M = (M' - r \cdot A)/B.$$

3.14 Schlüssel

3.14.1 Schlüsselaustausch

Diffie-Hellmans Protokoll zum Schlüsselaustausch

Zwei Personen bestimmen einen Schlüssel über eine unsichere Kommunikationsleitung:

1. gegeben:

   Primzahl p (hunderte von Bits lang)

   erzeugendes Element α für $(\mathbb{Z}_p \setminus \{0\}, \cdot)$

2. Person A wählt Z_A, Person B wählt Z_B
   $Z_A, Z_B \leq p-2$, Z_A und Z_B werden geheim gehalten.

3. A schickt $C_A = \alpha^{Z_A}$, B schickt $C_B = \alpha^{Z_B}$

4. als Schlüssel wird $k = \alpha^{Z_A \cdot Z_B} = (C_B)^{Z_A} = (C_A)^{Z_B}$ benutzt.

Um den Schlüssel zu knacken muß man entweder: Z_A aus C_A oder Z_B aus C_B berechnen. *Aber:* Es ist kein schneller Algorithmus zur Berechnung von diskreten Logarithmen (d. h. in $\mathbb{Z}_p$) bekannt.

Schlüsselaustausch mit public key Verfahren

Alice und Bob können Schlüsselhälften generieren und sich die Schlüsselhälften verschlüsselt und unterschrieben gegenseitig verschicken.

Das ist die wichtigste Anwendung von public key Verfahren.

3.14.2 Schlüsselverwaltung

Hardware-Lösung

Ein Rechner verfügt über eine **Kryptographische Einheit**. Das ist ein extra Chip mit folgenden Eigenschaften:

- Der Chip steckt in einer „tamper box“, d.h. in einem Umschag, den man aus technologischen Gründen nicht öffnen kann (beim Öffnen wird der Chip vernichtet).
- Der Chip gibt keine elektromagnetische Strahlung ab (die man analysieren könnte).
- Der Chip enthält einen Schlüssel (Master Key), der nur einmal eingegeben aber nicht geändert oder gar gelesen werden kann.

Die gesamte Kommunikation zwischen Prozessor und der Außenwelt erfolgt über den Chip (Abbildung 3.19). Der Chip holt die verschiedenen anderen (Primär- und Sekundär-) Schlüssel (siehe nächster Abschnitt) und codiert oder decodiert die Nachrichten entsprechend.

Dieses Verfahren findet sogar auf PC-Ebene (in den USA) Anwendung. Die Sicherheit, die erreicht wird, ist sehr groß. Es gibt jedoch Probleme, falls der Chip kaputt geht und der Master Key nirgendwo anders gespeichert ist. Alle chiffrierten Files werden dann unlesbar.

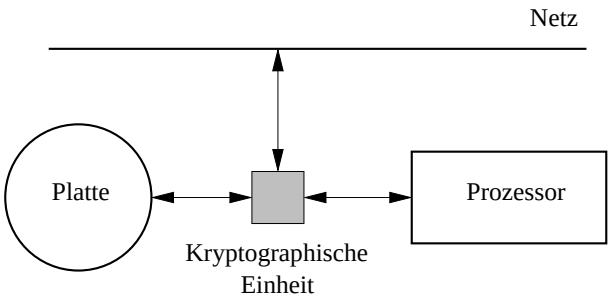


Abbildung 3.19: Kryptographische Einheit

Hierarchie der Schlüssel

Es gibt verschiedene Schlüssel, die andere Aufgaben haben und anders geschützt werden:

Master Key: Er wird nie geändert und ist nicht lesbar. Er existiert nur in einer kryptographischen Einheit oder im Gedächtnis und auf einer ID-Karte.

Primärschlüssel: Die Schlüssel, die zur Kommunikation und Speicherung von Dateien verwendet werden.

Sekundärschlüssel: Schlüssel zur Übertragung und Vereinbarung von Schlüsseln. Sie werden selten geändert.

Primärschlüssel werden oft geändert (um eine Kryptoanalyse zu erschweren). Sekundärschlüssel werden selten geändert – häufige Änderungen sind nicht notwendig, da man nicht viel Text mit Sekundärschlüsseln chiffriert.

Speicherung von Schlüsseln

Am bestem werden die Schlüssel mit dem Master Key verschlüsselt und dann gespeichert (wegen der Schlüssellänge muß man die Schlüssel auf einer Platte speichern. Denke an RSA-Schlüssel!)
Vorgehen beim Verschlüsseln mit einem Primär- oder Sekundärschlüssel K:

1. Der Code von K wird in die kryptographische Einheit geholt.
2. Mit dem Master Key wird K aus dem Code berechnet.
3. Der Klartext wird (stückweise) in die kryptographische Einheit geholt, mit K chiffriert und in dieser Form nach draußen geschickt.
4. Am Ende wird K in der kryptographischen Einheit vernichtet.

Beispiel PGP: Der Master Key und der Primärschlüssel werden im Gedächtnis gehalten (es handelt sich um ein symmetrisches Verfahren, also müssen die Schlüssel nicht lang sein). Der Sekundärschlüssel befindet sich (verschlüsselt) auf der Platte und die public keys in **key rings** (Die Idee ist die folgende: Man verschickt die eigenen public keys an vielen Personen. Diese Personen nehmen die Schlüssel in den eigenen Key Ring auf und bestätigen dadurch die Authentizität der Schlüssel. Also reichen nicht die root-Rechte in einem System, um die public keys einer Person zu ändern).

Beschaffung von Schlüsseln

- Einige Schlüssel sind lang und müssen somit durch Programme erzeugt werden.
- Man braucht normalerweise viele Schlüssel (z.B. zur Kommunikation mit verchiedenen Personen).
- Die Erzeugung von Schlüsseln mit Hilfe von Generatoren für Zufallszahlen ist gefährlich: man weiß nie ob Mallet das Programm so umgebaut hat, daß die erzeugten Folgen auch an ihn verschickt werden.
- **Schlüsselverteilung-Zentrale** (SVZ)– ist ein speziell geschützter Server, der die Schlüssel erzeugt. Voraussgesetzt wird, daß die Kommunikation zwischen SVZ und dem Benutzer gesichert ist (durch eine digitale Unterschrift und einen exclusiven Schlüssel für den Benutzer). Um die Gefahr eines Einbruchs in eine SVZ zu verringern, holt man die Schlüssel von vielen SVZ und kombiniert sie (z.B. durch Konkatenation)
- Eine elegante Lösung mit einer kryptographischen Einheit ist die folgende Erzeugung des i-ten Schlüssels: der i-te Schlüssel ist DES(Zeit+i,Master Key).

Schlüsselverlust

Der Schlüsselverlust ist ein ernstes Problem, denn:

- die Dateien können verlorengehen.
- ein Mitarbeiter, der alleine einen Schlüssel zu wichtigen Dateien besitzt, kann seine Firma erpressen.

Lösung: durch *secret sharing*. Ein Schlüssel wird in codierter Form auf n Personen verteilt. Es reicht die Codes von k Personen zu kennen um den Orginalschlüssel zu rekonstruieren. Weniger als k Teile reichen dagegen nicht aus. (Siehe Kapitel 3.15.3)

3.15 Einige kryptographische Protokolle

3.15.1 Man-in-the-middle Angriff

Sichere Chiffrierungen reichen manchmal nicht aus.

Beispiel : Alice und Bob wollen geheime Informationen austauschen. Sie machen das so:

1. Bob schickt an Alice seinen public key.
2. Alice wählt eine zufällige Folge x als Schlüssel für DES.
3. Alice chiffriert x mit dem public key von Bob und schickt dieses an Bob.
4. Bob entschlüsselt die Nachricht und erhält x.
5. Alice und Bob benutzen den Schlüssel x für ihre weitere Kommunikation.

Mallet ist der böse „man in the middle“. Das Protokoll von oben kann von Mallet unbemerkt beeinflußt werden:

1. Bob schickt an Alice seinen public key.
   Mallet fängt den Brief von Bob ab und schickt seinen public key an Alice.
2. Alice wählt eine zufällige Folge x als Schlüssel für DES.
3. Alice chiffriert x mit dem public key von Mallet und schickt dieses an Bob.
   Mallet fängt den Brief mit dem codierten x ab, entschlüsselt mit seinem private key, chiffriert mit Bobs public key und schickt es weiter an Bob.
4. Bob entschlüsselt die Nachricht und hat x.
5. Alice und Bob benutzen x für ihre weitere Kommunikation
   `und ahnen nicht, daß` **Mallet die Nachrichten entschlüsseln und sogar austauschen kann**.

Interlock-Protokoll (Ein Protokoll gegen den man-in-the-middle Angriff)

1. Bob schickt an Alice seinen public key.
2. Alice schickt an Bob ihren public key.
3. Bob verschlüsselt seine erste Nachricht mit dem public key von Alice, er sendet **die erste Hälfte des Kryptogramms** an Alice.
4. Erst nachdem Alice die erste Hälfte des Kryptogramms erhalten hat, chiffriert sie ihre Nachricht mit dem public key von Bob, und sendet die erste Hälfte des Kryptogramms an Bob.
5. Bob erhält die erste Hälfte von Alice und schickt seine zweite Hälfte.
6. Alice erhält die zweite Hälfte der Nachricht von Bob, entschlüsselt sie und überprüft ob diese Nachricht sinvoll ist (also keine chaotische Folge der Bits); falls alles in Ordnung ist, dann schickt Alice die zweite Hälfte der Nachricht an Bob.
7. Bob entschlüsselt die Nachricht von Alice und überprüft ob sie in Ordnung ist; falls ja, dann fängt Bob dasselbe Spiel mit der nächsten Nachricht an.

Jetzt kann Mallet weiterhin Bob und Alice seinen public key weitergeben. Er kann jedoch nicht die Nachrichtenhälften entsprechend Alices oder Bobs public key weiterleiten, da Mallet z.B. nach Abfangen der ersten Hälfte der Nachricht von Bob noch kein Bit dieser Nachricht entschlüsseln kann. Dadurch kann er nicht die erste Hälfte der Nachricht mit dem public key von Alice chiffrieren. Die einzige Möglichkeit für Mallet wäre seine eigene Nachricht zu verschicken. Dann kann aber Alice merken, daß die Nachrichten, die angeblich von Bob kommen, nicht ihren Erwartungen entsprechen.

3.15.2 Kerberos

Kerberos ist ein Protokoll, das die Kommunikation in unsicheren Netzwerken sichert. Anwendug: große Systeme mit mehreren Servern und vielen Workstations. Entwickelt am MIT (Public Domain).

Sicherheitsaspekte:

- Jede Kommunikation zwischen dem Benutzer und seinem Server erfolgt chiffriert.
- Jeder Zugriff zu Systemkomponenten erfolgt mit dem Ticket, das eine zeitbeschränkte Gültigkeit hat.
- Kritische Teile des Protokolls werden auf speziell geschützten Servern ausgeführt.
- Kerberos löst keine Probleme, die nicht durch die Kommunikation entstehen.

Teile des Kerberos-Systems:

Kerberos-Server: Verifiziert die Identität des Benutzers und vergibt das Ticket für den TGS.

Ticket-Granting-Server (TGS): Mit der Zustimmung des Kerberos Servers vergibt er Tickets für seine Server im Netzwerk.

(Dienst)Server: Realisiert die Dienste, die mit gültigen Tickets angefordert werden.

Client: der Benutzer.

Ablauf einer Kerberos-Session (siehe Abbildungen 3.20,3.21,3.22):

- Ein Benutzer identifiziert sich beim Kerberos-Server und verlangt den Zugriff zu TGS.
- Der Kerberos-Server antwortet mit einer Nachricht, die ein Ticket zu TGS und den Session-Key S enthält. Die Nachricht ist mit dem Schlüssel C des Benutzers chiffriert (dadurch kann nur der Benutzer *den Umschlag öffnen* und das Ticket bekommen). Das Ticket ist mit dem Schlüssel von TGS chiffriert.
- Der Benutzer sendet dem TGS gleichzeitig:
  1. das Ticket und
  2. seine Anforderung für den Dienstserver (verschlüsselt mit dem Session Key S).
- TGS enschlüsselt das Ticket des Kerberos-Servers, erhält dadurch den Session Key S und dechiffriert die Anforderung (paßt die Anforderung zu dem Session Key nicht $\Rightarrow$ Betrüger im Netz!).
- TGS antwortet mit einem Ticket für den Dienstserver und einem Session Key X für die Kommunikation mit dem Dienstserver (Verschlüsselt mit Session Key S). Das Ticket ist ein Kryptogramm chiffriert mit dem Schlüssel W vom Dienstserver.
- Der Benutzer entschlüsselt die Nachricht und erhält dadurch X und das Ticket für den Dienstserver.
- Der Benutzer sendet gleichzeitig das Ticket und seine Anforderung (verschlüsselt mit X) zu dem Dienstserver.

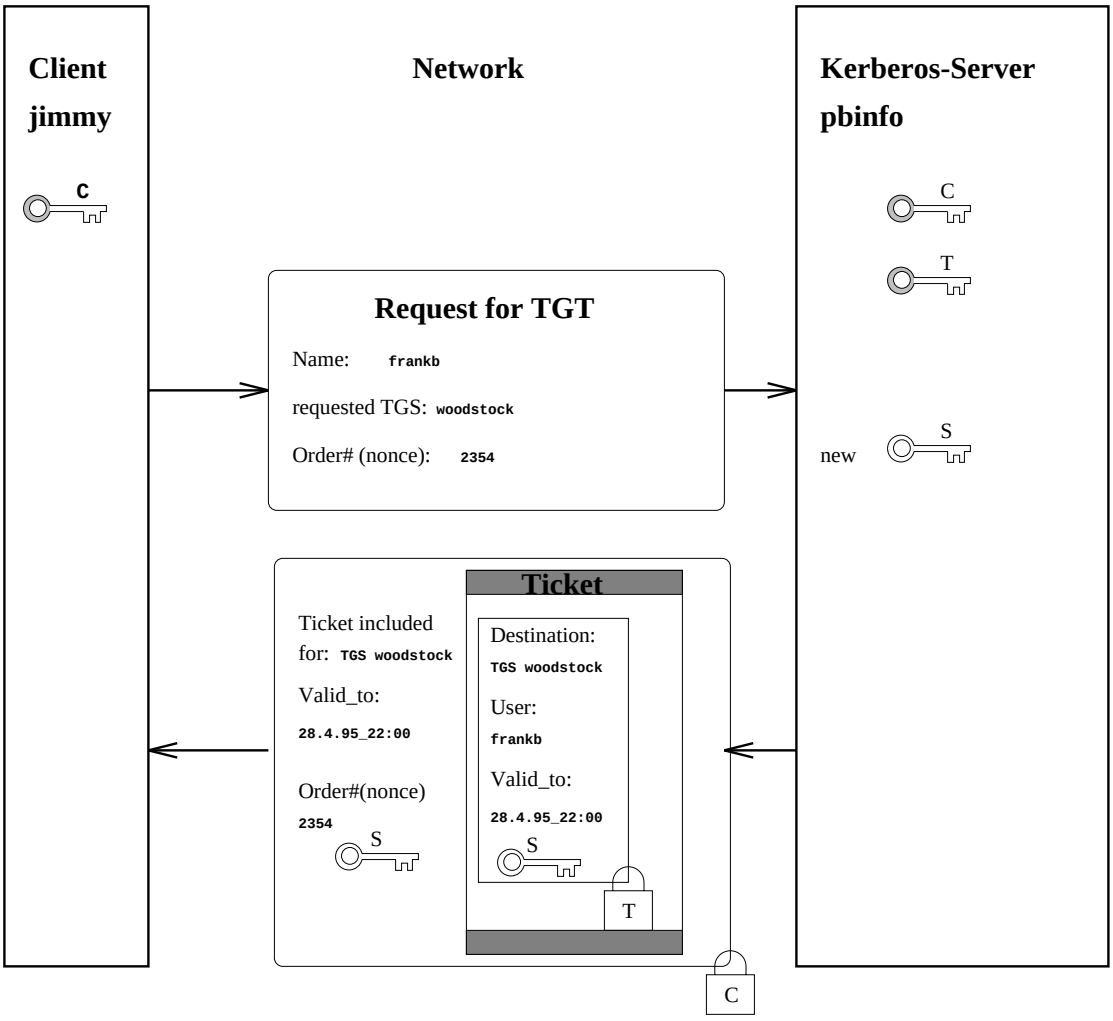


Abbildung 3.20: Kommunikation mit Kerberos Server

- Der Dienstserver entschlüsselt das Ticket, erhält dadurch den Session Key X und entschlüsselt mit X die Anforderung (gleichzeitig wird dadurch die Gültigkeit überprüft).
- Jede weitere Kommunikation zwischen dem Benutzer und dem Dienstserver wird mit X chiffriert.

Alles erfolgt vollautomatisch (man braucht nur entsprechende Kerberos-Versionen von login, rshell, ... zu instalieren) und sehr schnell.

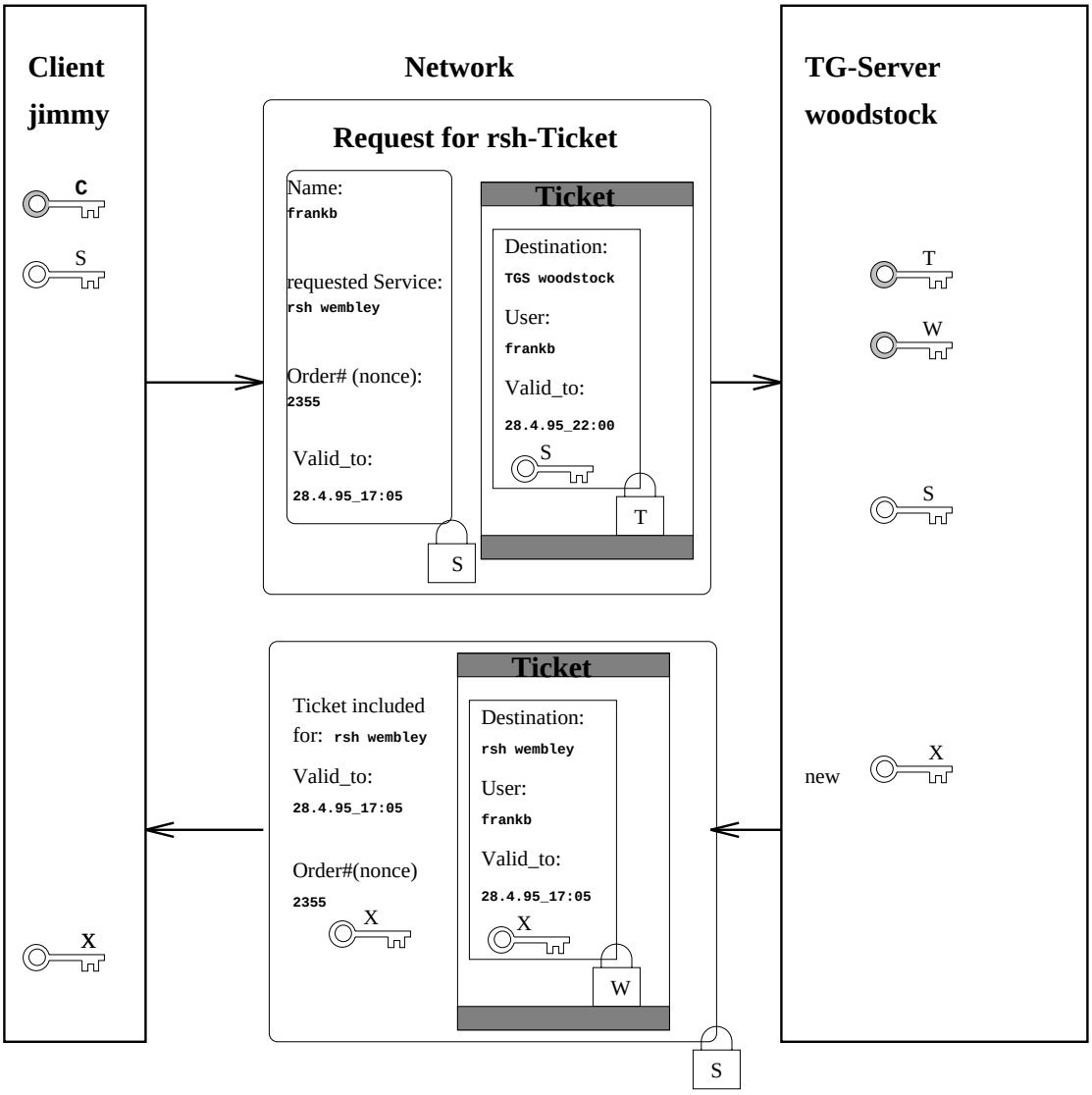


Abbildung 3.21: Kommunikation mit TGS von Kerberos

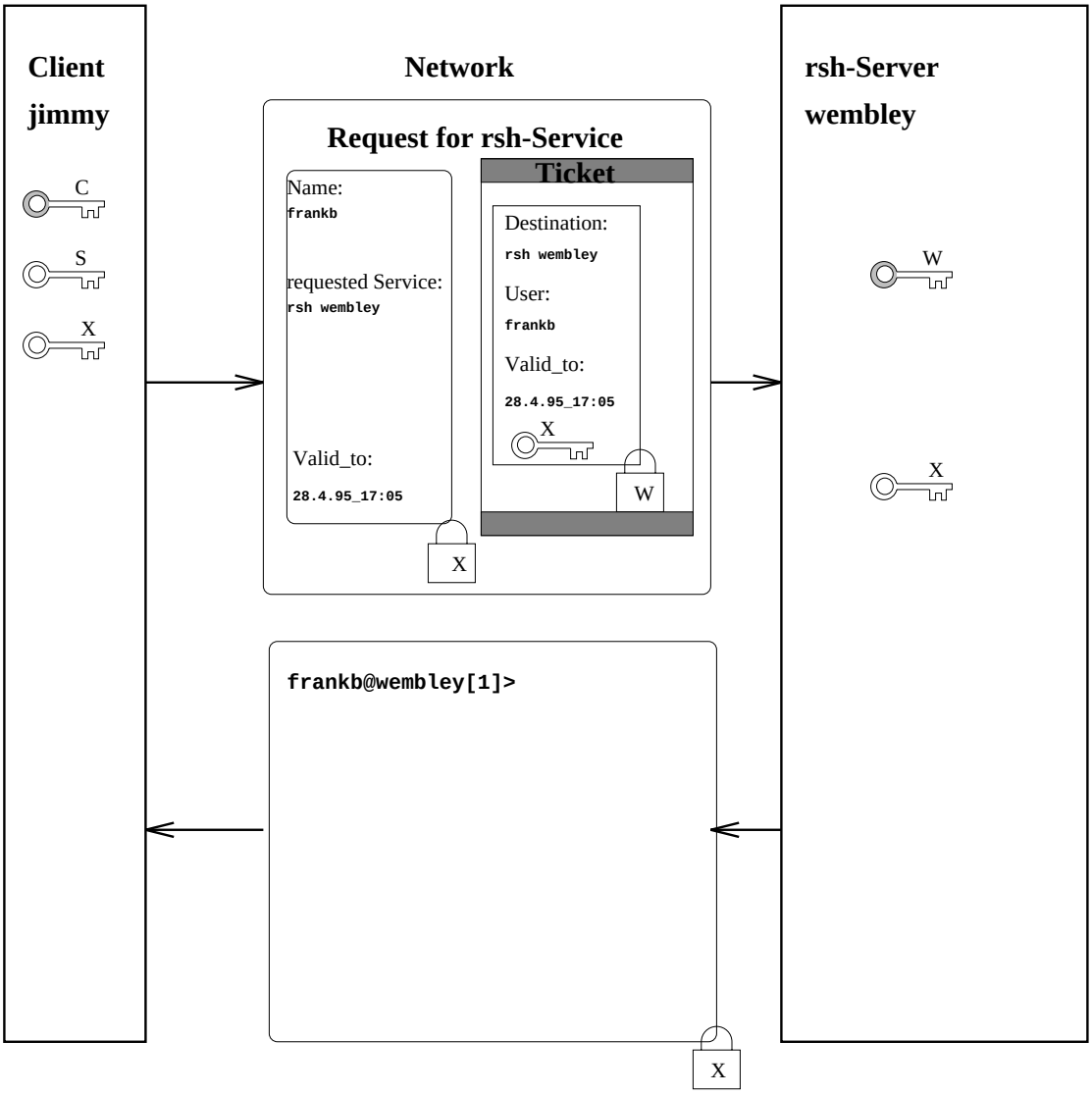


Abbildung 3.22: Kommunikation mit dem Server beim Kerberos-Protokoll

3.15.3 Secret Sharing

Secret sharing ist zum Beispiel in folgender Situation notwendig:
In einer Firma gibt es wichtige Pläne, die eine neue erfolgreiche Technologie beschreiben. Der Zugang zu den Plänen wird dadurch geschützt, daß der Safe sich nur dann öffnet, wenn alle führenden Personen in der Firma ihre Schlüssel gleichzeitig in den Safe stecken.

Secret sharing mit XOR

Annahmen: ein Text M muß so codiert werden, daß die Personen $A_1, \ldots, A_k$ gemeinsam M rekonstruieren können, aber weniger als k von diesen Personen können M nicht rekonstruieren.

Algorithmus:

1. man erzeugt $k-1$ zufällige Bitfolgen $R_1, \ldots, R_{k-1}$, die so lang sind wie M,
2. sei $R = M \oplus R_1 \oplus \ldots \oplus R_{k-1}$ (alle XOR bitweise durchgeführt),
3. für $i = 1, \ldots, k-1$ erhält Person A_i die Folge R_i; die Person A_k erhält R.

Alle Personen zusammen können M berechnen:

$$M = R \oplus R_1 \oplus \ldots \oplus R_{k-1}.$$

Weniger als k Personen können dagegen M nicht berechnen. Zum Beispiel können die Personen $A_2, \ldots, A_k$ nur

$$M \oplus R_1 = R \oplus R_2 \oplus \ldots \oplus R_{k-1}$$

berechnen. Aber weil R_1 eine zufällige Folge ist, ist $M \oplus R_1$ auch völlig zufällig.

Vorteile und Nachteile:

- perfekte Sicherheit; $k-1$ Personen erhalten überhaupt keine Informationen über das Geheimnis M.
- Der Verlust eines Teils ist gleichzeitig der Verlust von M.

Secret sharing mit Schwellen

Das letzte Protokoll wird so abgeändert, daß k Personen ihre Teile des Geheimnisses bekommen, aber je m Personen können das Geheimnis rekonstruieren. ($k-m$ Personen können „streiken“ und die Rekonstruktion des Geheimnisses doch nicht verhindern). Ein Protokoll, das so was garantiert, heißt *Secret sharing mit Schwellen*.

Ein Protokoll mit Schwellen:

1. Eine ausreichend große Primzahl p wird gewählt (p muß größer als das Geheimnis und die Anzahl der Teile sein).
2. Man wählt zufällig $m-1$ Zahlen $w_{m-1}, w_{m-2}, \ldots, w_1 < p$.
3. Sei $w(x) = N + \sum_{i=1}^{m-1} w_i \cdot x^i$, wobei N die Nachricht ist, die wir verheimlichen wollen.

4. Im Körper $\mathbb{Z}_p$ berechnet man die Werte $w(1), w(2), \ldots, w(k)$. Die i-te Person erhält $w(i)$.

Es ist bekannt, daß ein Polynom vom Grad $m-1$ durch m Werte eindeutig bestimmt wird.

3.15.4 Bit Commitment

In vielen Situationen müssen zwei Personen „die Münze werfen" um ein zufälliges Bit zu erzeugen. Ohne Münze geht es auch: jede Person wählt ein Bit; sie sagen **gleichzeitig** was sie gewählt haben; das XOR von beiden Bits ist das Ergebnis. Eine Elektronische Version dieses Protokolls ist nicht realisierbar: die Personen können nicht gleichzeitig ihre Bits verraten. Ohne diese Forderung kann jedoch die Person, die später seine Wahl verraten hat, beliebig das Ergebnis manipulieren.

Bit commitment ist ein Protokoll, der gewährleistet, daß

- Alice ein Bit wählen kann,
- Alice ihre Wahl nachher nicht ändern kann,
- Bob über die Wahl von Alice nur mit Erlaubnis von Alice erfahren kann.

Bit commitment ist perfekt für das Münzen werfen: Alice wählt ein Bit mit Hilfe des bit commitment-Protokoll und Bob wählt (öffentlich) ein Bit. Zum Schluß deckt Alice ihre Wahl auf.

Bit commitment mit symmetrischen Verfahren

Verlauf des Protokolls:

- Bob erzeugt eine zufällige Bitfolge R und sendet diese an Alice.
- Alice wählt ein Bit b und einen zufälligen Schlüssel K. Sie verschlüsselt R und b mit K und sendet die codierte Nachricht $E_K(R, b)$ an Bob.
- Bob wählt öffentlich sein Bit.
- Alice sendet K an Bob.
- Bob entschlüsselt $E_K(R, b)$ und überprüft ob er R zurückbekommen hat. Dabei erfährt er auch b.

Die Sicherheit des Protokolls basiert auf der Tatsache, daß es praktisch unmöglich ist, einen Schlüssel K' mit $E_K(R, b) = E_{K'}(R, 1-b)$ zu finden.

Bit commitment mit Einweg-Hashfunktionen

Verlauf des Protokolls:

1. Bob erzeugt eine zufällige Folge R_1 und sendet diese an Alice.
2. Alice erzeugt zufällig eine Bitfolge R_2 und wählt ein Bit b.
3. Alice sendet an Bob den Hashwert $H(R_1, R_2, b)$.
4. Bob wählt öffentlich sein Bit.
5. Alice sendet R_2 und b an Bob.
6. Bob berechnet $H(R_1, R_2, b)$ und überprüft, ob der früher erhaltene Hashwert korrekt war.

Die Sicherheit des Verfahrens ist auf der Tatsache basiert, daß es für gegebene R_1 und b praktisch unmöglich ist, Folgen R und R' zu finden, so daß $H(R_1, R, b) = H(R_1, R', 1-b)$.

3.16 Kryptoanalyse

Arten von kryptoanalytischen Angriffen

ciphertext-only Angriff: Einige Kryptogramme sind bekannt (z.B. durch Abhören von chiffrierten Nachrichten); aber es gibt keine Hinweise über die entsprechenden Klartexte.

known plaintext Angriff: Einige Paare (Klartext,Kryptogramm) sind bekannt (z.B. wenn man die Bedeutung von einigen Kryptogrammen erfährt), aber die Paare können nicht ausgewählt werden.

chosen plaintext Angriff: Man kann zu ausgewählten Klartexten die resultierenden Kryptogramme bestimmen (z.B. Analyse von einem kryptographischen Gerät mit internem Schlüssel, der unbekannt ist $\rightsquigarrow$ UNIX-Passwörter).

chosen ciphertext Angriff: Der Angreifer kann zu ausgewählten Kryptogrammen die entschlüsselten Klartexte bestimmen (auch nützlich zur Untersuchung vom kryptographischen Gerät).

Selbst falls ein Angriff sehr viel Zeit und Rechnerkapazität benötigt sind einige schlaue Angriffe vorstellbar:

Viren: Man schreibt einen Virus, der im Betriebsystem steckt und versucht DES Kryptogramme zu knacken. Gelingt dies, dann wird der Rechner mit der Anzeige auf dem Bildschirm: „Hardwarefehlercode xyz; Call our toll-free maintenance service“ blockiert. Der Fehlercode enthält wichtige Informationen über den gefundenen Schlüssel.

„Chinesisches Roulette:“ Jeder Fernseher in China wird mit einem kleinen Chip ausgerüstet, der durch Videotext verschickte Kryptogramme zu knacken versucht. Jeder Fernsehen ist für einige Schlüssel zuständig. Bei Erfolg erscheint der Schlüssel auf dem Bildschirm und man kann seinen Preis (unter Angabe der Fernsehgerätenummer und des Schlüssels) abholen.

In beiden Fällen verfügt man über eine riesige Rechnerkapazität, die andere bezahlen.

3.16.1 Differentielle Kryptoanalyse

- Öffentlich bekannt seit 1990; berücksichtigt bei dem Entwurf von DES.
- Geeignet für iterative Verfahren wie DES.
- Eine Methode, die DES mit kleiner Rundenzahl knackt. Sie braucht ca. 2^{47} Klartext-Kryptogramm Paare beim chosen plaintext-Angriff oder 2^{55} Paare beim known plaintext-Angriff für DES. Bei 8 DES-Runden liegen diese Zahlen bei nur 2^{14} und 2^{38}.
- Viele Verfahren wurden mit differentieller Kryptoanalyse geknackt.

Differentielle Kryptoanalyse einer S-Box

Bei einer DES-Runde werden die Eingabebits permutiert, kopiert (Erweiterungspermutation), mit dem Schlüssel XORt und zu S-Boxen geschickt. Die enscheidenden Phasen sind:

1) XOR mit dem Runden-Schlüssel,
2) Berechnungen in S-Boxen.

Wir werden nur diese Bits der Eingabe betrachten, die nach dem XOR mit dem Runden-Schlüssel zu einer bestimmter S-Box geschickt werden. Sei T der Teil des Netzwerkes mit der gewählten S-Box und XOR-Gatter, deren Ausgaben die Eingaben für die S-Box sind.

Trick mit dem XOR: um mindestens ein Problem zu eliminieren betrachtet man nicht einzelne Eingabefolgen für T sondern Paare von Eingaben. Seien X, Y zwei solche Eingabefolgen.

Beobachtung: Durch das XOR mit den Bits vom Runden-Schlüssel K bekommt man aus X und Y die Folgen $X \oplus K$, $Y \oplus K$. Dann gilt

$$(X \oplus K) \oplus (Y \oplus K) = X \oplus Y.$$

Also wird das XOR von X und Y durch Anwendung von Schlüsselbits nicht geändert – in anderen Worten die alte Differenz bleibt erhalten.

Das XOR von X und Y wird durch die Anwendung der S-Box in den meisten Fällen geändert. Die neue „Differenz“ ist nicht nur von $X \oplus Y$ abhängig sondern auch von den konkreten Eingaben X und Y.

Beispiel: S-Box S_1. Hexadezimalzahlen wird ein Index x hinzugefügt. Für Eingabefolgen X und Y, bei denen $X \oplus Y = 34_x$, gibt es die folgende Möglichkeiten für das XOR der Ausgaben der S-Boxen:

$$\begin{array}{rrrr} 1_x \text{ für } & 8 \text{ Paare } X,Y, & 2_x \text{ für } & 16 \text{ Paare } X,Y, \\ 3_x \text{ für } & 6 \text{ Paare } X,Y, & 4_x \text{ für } & 2 \text{ Paare } X,Y, \\ 7_x \text{ für } & 12 \text{ Paare } X,Y, & 8_x \text{ für } & 6 \text{ Paare } X,Y, \\ D_x \text{ für } & 8 \text{ Paare } X,Y, & F_x \text{ für } & 6 \text{ Paare } X,Y, \end{array}$$

(diese Informationen können durch Simulation der S-Box auf allen möglichen Eingaben gewonnen werden. Die entsprechende Tabellen existieren schon.)
Wenn wir also zum Beispiel feststellen, daß die „Differenz“ nach der Anwendung von S-Box S_1 gleich D_x ist, dann wissen wir, daß die Eingabe zu S_1 eine von 8 möglichen Paaren ist. (In diesem Fall sind das $07_x, 33_x, 11_x, 25_x, 17_x, 23_x, 1D_x, 29_x$ für die erste Folge; die zweite Eingabe der S-Box bekommt man als die erste Eingabe XORt mit 34_x). Jede Möglichkeit ergibt einen möglichen Schlüssel K, z. B. wenn die erste Eingabe für die S-Box 07_x ist, dann gilt $X \oplus K = 07_x$. Also $K = X \oplus 07_x$. Eine ähnliche Analyse kann man für ein anderes Paar X', Y' der Eingabefolgen machen. Jetzt untersucht man die neue „Differenz“ $X' \oplus Y'$. Entsprechend den Ergebnissen bekommen wir eine neue Menge der möglichen Schlüssel. Der tatsächliche Schlüssel ist ein Element der beiden Mengen. Dadurch kann man die Möglichkeiten auf eine kleinere Menge beschränken. Nach noch einigen solche Paaren ist der Schlüssel dann eindeutig.

Die differentialle Kryptoanalyse für DES verallgemeinert den Ansatz für eine einzelne S-Box.

3.16.2 Lineare Kryptoanalyse

Lineare Kryptoanalyse ist bis jetzt die effizienteste Methode gegen DES. Sie braucht durchschnittlich 2^{43} Klartext-Kryptogramm Paaren um den Schlüssel zu finden (known plaintext-Angriff).

Die Idee der S-Boxen: Die Folgen, die während des Chiffrierens entstehen, können nicht durch einfache algebraische Operationen (wie bitweise XOR) aus der Eingabefolge abgeleitet werden.

Idee linearer Kryptoanalyse: Obwohl S-Boxen keine einfache Funktionen berechnen, kann man S-Boxen durch einfache Formeln *approximieren.*

Für eine S-Box S betrachten wir eine Formel

$$i_{j_1} \oplus i_{j_2} \oplus \cdots \oplus i_{j_e} = o_{h_1} \oplus o_{h_2} \oplus \cdots \oplus o_{h_f},$$

wobei i_s bzw. o_s das s-te Eingabe- bzw. Ausgabebit bezeichnet. (Kurz wird diese Formel auch durch $\mathrm{In}[j_1, \ldots, j_e] = \mathrm{Out}[h_1, \ldots, h_f]$ dargestellt.) So eine Formel gilt nicht für alle Eingaben, aber es reicht, daß es **relativ oft** oder **relativ selten** gilt. Dann sagen wir, daß diese Formel die S-Box S *linear approximiert.*
Zum Beispiel, die fünfte S-Box kann durch die Formel

$$\mathrm{In}[4] = \mathrm{Out}[0, 1, 2, 3]$$

charakterisiert werden. Diese Gleichung gilt für 19% der Eingaben. Also kann man sagen, daß

$$i_4 = \neg(o_0 \oplus o_1 \oplus o_2 \oplus o_3)$$

und das trifft in 100% -19% = 81% der Fällen zu (wichtig ist also, daß die Erfolgsrate möglichst weit von 50% abweicht).

Aus den Formeln, die einzelne S-Boxen approximieren, ist es möglich eine Formel zu bauen, die einige Zusammenhänge zwischen der Rundeneingabe, der Rundenausgabe und den Runden-Schlüsseln darstellt. (Diese Zusammenhänge gelten nur statistisch!).
Die Formeln für einzelne Runden kann man verknüpfen.

Beispiel: für 3-Runden-DES kann die folgende Formel abgeleitet werden:

$$\oplus(p_7, p_{18}, p_{24}, p_{29}, p_{47}, c_7, c_{18}, c_{24}, c_{29}, c_{47}) = k^1_{22} \oplus k^3_{22} \tag{3.5}$$

wobei p_i das i-te Bit des Klartextes, c_j das j-te Bit des Kryptogramms und k^u_v das v-te Bit des Runden-Schlüssels in der Runde u bezeichnet. Wir nehmen an, daß die Gleichung (3.5) mit Wahrscheinlichkeit q für zufällige Eingabefolge gilt. (q ist bekannt, aber wir werden q nicht ausrechnen).
Aus Formel (3.5) kann man jetzt nach genügend vielen Tests $k = k^1_{22} \oplus k^3_{22}$ bestimmen: für jedes Paar Klartext-Kryptogramm berechnen wir die linke Seite von (3.5). Sei r der Anteil der berechneten Werte, die gleich 1 sind (und somit $1-r$ der Anteil der berechneten Werte, die gleich 0 sind).

Fall 1: $q > 0.5$ dann $k := 0$ falls $r < 0.5$ und $k := 1$ falls $r > 0.5$,

Fall 2: $q < 0.5$ dann $k := 0$ falls $r > 0.5$ und $k := 1$ falls $r < 0.5$.

Die Wahrscheinlichkeit, daß man dadurch k richtig gewählt hat hängt von $|q - 0.5|$ und der Anzahl der Klartext-Kryptogramm Paare ab. Mehr Paare machen den Test mehr zuverlässig. Es läßt sich z. B. berechnen, daß $2/|q - 0.5|^2$ solche Paare das richtige k mit Wahrscheinlichkeit über 99% ergeben.

Anhang A

Zeichenerklärungen

$\oplus_R$	Addition im Ring R; $R \times R \to R$
$\oplus_n$	Addition im Ring $\mathbb{Z}_n$; $\mathbb{Z}_n \times \mathbb{Z}_n \to \mathbb{Z}_n$
$\oplus_R^{\flat}$	komponentenweise Addition im Ring R; $R^m \times R^m \to R^m$
$\odot_R$	Multiplikation im Ring R; $R \times R \to R$
$\odot_n$	Multiplikation im Ring $\mathbb{Z}_n$; $\mathbb{Z}_n \times \mathbb{Z}_n \to \mathbb{Z}_n$
$\odot_R^{\flat}$	komponentenweise Multiplikation im Ring R; $R^m \times R^m \to R^m$
$GF(p)$	Galois Feld mit p Elementen, d.h. endlicher Körper mit p Elementen
$\mathfrak{mod}$	Kongruenzen im Ring $\mathbb{Z}_n$; $a = b \,\mathfrak{mod}\, n \quad \Leftrightarrow \quad \exists i \in \mathbb{Z} : a = b + i \cdot n$
mod	modulo-Operation; $a = b \bmod n \quad \Leftrightarrow \quad a = b \,\mathfrak{mod}\, n \ \wedge \ 0 \leq a < n$
$\Re(r, m)$	Reed-Muller Code mit den Parametern r und m

Anhang B

DES

In diesem Anhang sollen nochmals im Detail die eingesetzeten Permutationen und S-Boxen aufgeführt werden.

B.1 Initiale Permutation

Die initatiale Permuation $\pi_i : \{1, \ldots, 64\} \rightarrow \{1, \ldots, 64\}$ des Klartextes bildet Bit 1 auf Bit 58, Bit 2 auf Bit 50, usw. ab.

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

Die inverse Permutation (Bit 58 nach Bit 1, Bit 50 nach Bit 2, usw.) wird zum Abschluß auf den chiffrierten Text angewandt, um das abschließende Kryptogramm zu erhalten.

B.2 Schlüsseltransformation

Der Schlüssel wird zuerst *einmal* permutiert und in zwei Hälften unterteilt. Für jede Runde werden die zwei Hälften jeweils um ein oder zwei Positionen rotiert und eine *fest* vorgegebene Reihenfolge von Bits als Rundenschlüssel verwendet.

B.2.1 Schlüsselpermutation

Die Schlüsselpermuation $\pi_k : \{1, \ldots, 64\} \rightarrow: \{1, \ldots, 58\}$ (Parity-Bits werden ignoriert) bildet Bit 1 auf Bit 57, Bit 2 auf Bit 49, usw. ab.

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

B.2.2 Shifts

In jeder der 16 Runden werden die 28 Bits einer Hälfte um jeweils eine oder zwei Positionen rotiert, so daß am Ende insgesamt um 28 Positionen rotiert wurde.

Runde	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
# Shifts	1	1	2	2	2	2	2	2	1	2	2	2	2	2	2	1

B.2.3 Selektion

In jeder Runde werden 48 Bits selektiert $\pi_s : \{1, \ldots, 48\} \to \{1, \ldots, 56\}$ und zu dem Rundenschlüssel zusammengesetzt. Dabei ist Rundenschlüsselbit 1 das Bit 14, Rundenschlüsselbit 2 das Bit 17, usw..

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

B.3 Die Funktion f

In die in jeder Runde angewandte Funktion f fließt zum einen der Rundenschlüssel K_i (48 Bit) und zum anderen die Texthälfte R_{i-1} (32 Bit) ein. Die Texthälfte wird mit Hilfe der Erweiterungspermutation auf 48 Bit erweitert, die Bits des Rundenschlüssels und der „Texthälfte“ mit einem XOR bitweise verknüpft und danach aufgeteilt durch S-Boxen chiffriert.

B.3.1 Erweiterungspermutation

Die Erweiterungspermutation $\pi_e : \{1, \ldots, 48\} \to \{1, \ldots, 32\}$ erweitert und permutiert die „Texthälfte“ R_{i-1}. Dabei ist das neue Bit 1 das alte Bit 32, das neue Bit 2 das alte Bit 1, usw..

32	1	2	3	4	5	4	5	6	7	8	9
8	9	10	11	12	13	12	13	14	15	16	17
16	17	18	19	20	21	20	21	22	23	24	25
24	25	26	27	28	29	28	29	30	31	32	33

B.3.2 S-Boxen

Nachdem der Rundenschlüssel und der erweiterte Text mit einem bitweise XOR verknüpft worden sind, werden die 48 Bits in 8 Gruppen á 6 Bit unterteilt (1 ... 6, 7 ... 12, 13 ... 18, ..., 43 ... 48) und separat durch eine S-Box chiffriert. Dabei bestimmen die Bits 1 und 6 die Zeile und 2 bis 5 die Spalte. Ist also die Eingabe für die 3. S-Box 101010, so wird dieses nach dem Eintrag in der dritten Zeile (Index 10) und sechsten Spalte (Index 0101) chiffriert, also 15 bzw. 1111. Eine S-Box ist also eine Abbildung $s : \{0,1\}^6 \to \{0,1\}^4$.

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

B.3.3 Abschlußpermutation

Die Abschlußpermutation $\pi_f : \{1, \ldots, 32\} \rightarrow \{1, \ldots, 32\}$ permutiert noch einmal die $4 \cdot 8 = 32$ Bit S-Box-Ausgaben. Dabei wird Bit 1 nach Bit 16, Bit 2 nach Bit 7, usw. verschoben.

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

Index