# Kryptographische Aspekte selbstmodifizierender Verbindungsnetzwerke

Diplomarbeit
von

**Felix Holderied**

20. April 1993

Institut für Algorithmen und kognitive Systeme
Universität Karlsruhe

Betreuer:
Priv. Doz. Dr. Patrick Horster
Dipl. Inform. Peer Wichmann

# Inhaltsverzeichnis

**1 Über diese Arbeit** **4**

**2 Einführung** **5**

2.1 Verbindungsnetzwerke . . . . . . . . . . . . . . . . . . . . . . . . . . . 5

2.2 Verbindungsnetzwerke als Substitutionschiffre . . . . . . . . . . . . . . . . . 6

2.3 Die Selbstmodifikation . . . . . . . . . . . . . . . . . . . . . . . . . . . 8

2.4 Sequentielle Maschinen . . . . . . . . . . . . . . . . . . . . . . . . . . . 9

**3 Kryptographie** **10**

3.1 Eigenschaften des Kryptosystems . . . . . . . . . . . . . . . . . . . . . . . 10

3.2 Digitale Unterschriften . . . . . . . . . . . . . . . . . . . . . . . . . . . 11

3.3 $\mathrm{SINC}_n$ als Hashfunktion . . . . . . . . . . . . . . . . . . . . . . . . . 13

**4 Die Vernetzung** **14**

4.1 Entwicklung einer sinnvollen Vernetzung . . . . . . . . . . . . . . . . . . . 14

4.2 Beneš-Netzwerke . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 16

4.3 Mathematische Beschreibung des Beneš-Netzwerks . . . . . . . . . . . . . . . 21

4.4 Zustände des Beneš-Netzwerks . . . . . . . . . . . . . . . . . . . . . . . . 22

4.5 Beneš-äquivalente Netzwerke . . . . . . . . . . . . . . . . . . . . . . . . . 27

**5 Selbstmodifikationsfunktionen** **30**

5.1 Pfadabhängige Invertierung . . . . . . . . . . . . . . . . . . . . . . . . . 31

5.2 Einfache Pfadmodifikation . . . . . . . . . . . . . . . . . . . . . . . . . . 31

5.2.1 Eigenschaften der einfachen Pfadmodifikation . . . . . . . . . . . . . . 32

5.2.2 Brechen des Schlüssels . . . . . . . . . . . . . . . . . . . . . . . . 37

5.2.3 Die Hashfunktion . . . . . . . . . . . . . . . . . . . . . . . . . . . 42

5.3 Komplementäre Pfadmodifikation . . . . . . . . . . . . . . . . . . . . . . . 43

5.4 Modifikation der linken und rechten Pfadnachbarn . . . . . . . . . . . . . . . 44

5.5 Modifikation aller Pfadnachbarn . . . . . . . . . . . . . . . . . . . . . . . 44

5.6 Horizontal/Diagonalmodifikation . . . . . . . . . . . . . . . . . . . . . . . 44

5.7 Andere Modifikation mit Invertierungsmatrizen . . . . . . . . . . . . . . . . 46

5.8 Spaltenweise Shiftoperation . . . . . . . . . . . . . . . . . . . . . . . . . 46

5.9 Pseudozufallsgeneratoren . . . . . . . . . . . . . . . . . . . . . . . . . . 48

5.10 Selbstmodifikation “life” . . . . . . . . . . 49

**6 Attacken** **50**

6.1 Sequentielle, eingeschränkte Suche . . . . . . . . . . 50

6.2 Attacke mit mehrfachem Known-Plaintext . . . . . . . . . . 51

6.3 Chosen Plaintext/Chosen Ciphertext-Attacke . . . . . . . . . . 52

**7 Betriebsarten und Erweiterungen** **52**

7.1 Betriebsarten der Verschlüsselung . . . . . . . . . . 52

7.2 Rückkopplungen . . . . . . . . . . 53

7.3 Verkettung mehrerer SINC . . . . . . . . . . 55

7.4 Sicherheit verschiedener Betriebsarten . . . . . . . . . . 56

7.5 Betriebsarten der Hashfunktion . . . . . . . . . . 58

**8 Weitere Möglichkeiten** **59**

8.1 Expansion des Schlüssels . . . . . . . . . . 59

8.2 Kontraktion der Hashmatrix . . . . . . . . . . 59

**9 Abschließende Bemerkungen** **61**

**A Statistische Betrachtungen** **63**

**B Ein Simulationsprogramm** **71**

**C Nomenklatur** **88**

# 1 Über diese Arbeit

Bevor ich den Begriff des selbstmodifizierenden Verbindungsnetzwerks und dessen mögliche Verwendung in der Kryptographie erläutere, möchte ich ein paar Anmerkungen zur Entstehung dieses Kryptosystems machen.

Die Idee, rekonfigurierbare Verbindungsnetzwerke in der Kryptologie einzusetzen, wird, in der hier vorgestellten Form, erstmals in einer Arbeit von Michael Portz [Port91] erwähnt. Er schlägt vor, durch Kombination eines Beneš-Netzwerks mit einem Funktionsgenerator einen Permutationsgenerator zu erzeugen. Die durch den Funktionsgenerator erzeugte Kontrollfunktion legt dabei den Zustand der Tauscherelemente des Netzwerks fest und spezifiziert so die Permutation. Erfüllt die Kontrollfunktion Kriterien der Pseudozufälligkeit, so ist diese Eigenschaft übertragbar auf die Permutation. Dadurch kann eine gewisse Sicherheit des Verschlüsselungssystems garantiert werden.

Patrick Horster machte den Vorschlag, die Rekonfiguration eines Verbindungsnetzwerks durch Selbstmodifikation zu realisieren. Diese sollte ebenfalls die Eigenschaft der Pseudozufälligkeit besitzen. Die Selbstmodifikation kann dabei unter anderem auch vom Klartext abhängen. Am *Europäischen Institut für Systemsicherheit* in Karlsruhe (E.I.S.S.) entstand das Projekt SINC (*Selfmodifying Interconnection Network based Cryptosystem*), das sich mit der kryptologischen Relevanz von Verbindungsnetzwerken befaßt. In diesem Projekt entstand auch der Prototyp eines $SINC_8$ mit einfacher Pfadmodifikation [HoNP91]. Weitere Publikationen zum Einsatz rekonfigurierbarer Verbindungsnetzwerke in der Kryptographie sind [Vick92, Port92, Hors93].

Ich lernte das SINC in der Vorlesung “Signale, Codes und Chiffren II (Kryptographie)” kennen, die P. Horster im WS 91/92 an der Informatik-Fakultät in Karlsruhe las. Er schlug vor, ein Beneš-Netzwerk mit einfacher Selbstmodifikation als Permutationsbox für eine Substitutionschiffre zu verwenden. Diese kann als Basis für ein Verschlüsselungssystem dienen oder für eine kryptographische Hashfunktion genutzt werden. Ich implementierte eine kleine Version dieses Systems mit Hilfe eines Simulationsprogramms für digitale Schaltungen (DigSim). Das Experimentieren mit dieser Schaltung ergab, daß die Chiffrierung von Zeichen, mit dieser kleinen Version des SINC, sehr starke Regelmäßigkeiten aufweist. Dies ist eine, für Kryptosysteme, sehr schlechte Eigenschaft.

Neben dieser destruktiven Erkenntnis will ich in dieser Arbeit jedoch auch Vorschläge machen, um das System so zu verändern, daß es kryptographischen Kriterien genügt. Dies sind zum einen Kriterien der technischen Realisierbarkeit (Geschwindigkeit, Effizienz, Herstellungsaufwand), zum anderen sind es die Aspekte der Sicherheit. Bei einem Verschlüsselungssystem muß die Sicherheit gegenüber unautorisierter Entschlüsselung gewährleistet sein, eine kryptographische Hashfunktion ist sicher, wenn sie resistent gegen Kollisionen ist.

Ich gehe in dieser Arbeit von einer Verallgemeinerung selbstmodifizierender Netzwerke aus, und versuche anhand der notwendigen Bedingungen eines guten (sicheren) Kryptosystems bzw. einer Hashfunktion, Bedingungen für den Entwurf eines sicheren SINC aufzustellen.

Mein besonderer Dank gilt den Betreuern Patrick Horster und Peer Wichmann für Diskussion und Vorschläge zu dieser Diplomarbeit, und meinem Kommilitonen Armin Biere für wertvolle Tips zur Implementierung des Simulationsprogramms.

# 2 Einführung

## 2.1 Verbindungsnetzwerke

Verbindungsnetzwerke haben in der Informatik verschiedene Anwendungen; prinzipiell dienen sie der Realisierung von variablen Permutationen.

Basiselement ist ein *Tauscher* oder *Kreuzschalter* mit zwei Eingängen und zwei Ausgängen. Der Tauscher kann zwei Zustände annehmen: durchschalten (0) und vertauschen (1).

Abbildung 1: Tauscher eines Verbindungsnetzwerks

Durch rekonfigurierbare Tauscher, deren Stellung durch eine Kontrollfunktion bestimmt wird, können somit verschiedene Permutationen bzw. Verbindungen hergestellt werden. Es können damit Prozessoren oder Speichereinheiten variabel miteinander vernetzt werden. Die Vernetzung kann, je nach Anwendung, rekonfiguriert werden. In der Parallelverarbeitung kann es auch notwendig sein, die Konfiguration während eines laufenden Prozesses zu ändern. Eine weitere Anwendung der Parallelverarbeitung sind sogenannte Sortiernetzwerke. Die Tauscher arbeiten dabei als autonome Vergleicher. Handelt es sich bei den Eingangssignalen beispielsweise um codierte Zahlen, so kann der Tauscher diese vergleichen und die Signale "sortieren". Mit entsprechender Vernetzung der Tauscher können damit Sortiernetzwerke realisiert werden, die wesentlich schneller arbeiten als nichtparallele Sortieralgorithmen (siehe z.B. [Akl85]).

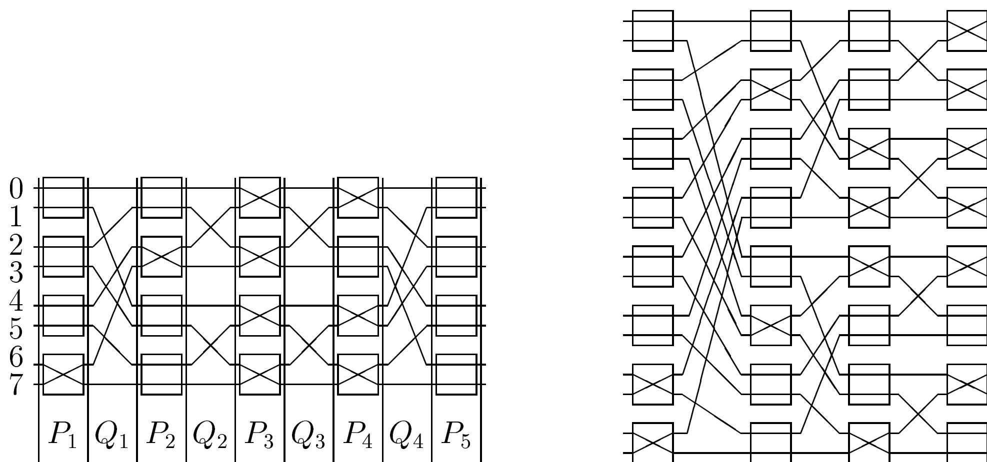



Abbildung 2: Beispiele für Verbindungsnetzwerke

Für die Anwendung in der Kryptographie beschränke ich mich auf Verbindungsnetzwerke mit "rechteckiger" Anordnung der Tauscher (siehe Abb. 2). Dabei sind $Z \cdot S$ Tauscher in $Z$ Zeilen und $S$ Spalten angeordnet. Die Vernetzung der Tauscher erfolgt üblicherweise zwischen zwei benachbarten Spalten. Jeder Ausgang eines Tauschers der Spalte $s$ ($1 \leq s \leq S-1$) ist mit einem Eingang eines Tauscher der Spalte $s+1$ verbunden und umgekehrt. Die Signalrichtung wollen wir dabei von links nach rechts festlegen.

## 2.2 Verbindungsnetzwerke als Substitutionschiffre

Unter einer allgemeinen Substitutionschiffre (Blockchiffre) versteht man eine bijektive Funktion $S$ über einem beschränkten Alphabet $A$, eine sogenannte Permutation. Die Funktion wird festgelegt durch einen Parameter $K$. Dieser wird als Schlüssel bezeichnet. Beispiele für Blockchiffren sind der DES (*Data Encryption Standard*) der FEAL (*Fast Data Encyphering Algorithm*) und PES (*Proposed Encryption Standard*). Auch das RSA-Verfahren (*Rivest, Shamir, Adleman*) kann als Blockchiffre aufgefaßt werden.

Realisiert werden die Verschlüsselungsfunktionen auf verschiedene Arten. Der Algorithmus des DES besteht aus einer Folge von Permutationen und logischen Produktbildungen zwischen dem Klartext und einem individuellen Schlüssel. Die Blockbreite von Klartext und Schlüsseltext ist 64 Bit, der Schlüssel besteht aus 56 signifikanten Bit. Der DES eignet sich gut für eine Hardwareimplementierung.

Der FEAL ist, wie der DES, eine Produktchiffre, jedoch stärker softwareorientiert. Block- und Schlüssellänge ist 64 Bit. Beim RSA-Verfahren werden die Klartextblöcke als Binärzahlen interpretiert. Die Verschlüsselung erfolgt durch Potenzierung des Klartextes mit einem Schlüssel modulo einer Zahl $n$. Die Zahl $n$ ist das Produkt zweier verschiedener, großer Primzahlen. Die Schlüssellänge (Größenordnung des Exponents), die Blocklänge und die Länge der Zahl $n$ sind etwa gleich, und liegen, je nach Sicherheitsanforderung, zwischen 200 und 1024 Bit.

Gemeinsam an diesen Chiffren ist die große Blockbreite von mindestens 64 Bit und eine Schlüssellänge von mindestens 56 Bit. Eine große Schlüssellänge ist notwendig, um eine *known-plaintext*-Attacke zu verhindern. Darunter versteht man eine Attacke, bei der ein Angreifer den Schlüsseltext und Teile des Klartextes kennt. Ist die Schlüssellänge zu klein, so kann der Angreifer anhand dieser Kenntnis den zugehörigen Schlüssel $K$ durch vollständige Schlüsselsuche ermitteln. Ist die Blockbreite zu klein, so besteht die Gefahr, daß in einem Text mehrfach derselbe Block auftritt. Das erlaubt einem potentiellen Angreifer eine Attacke, in diesem Fall die unautorisierte Entschlüsselung des Chiffrats, durch Anlegen eines Codebuchs.

Bei den genannten Chiffren handelt es sich um Permutationen auf den Klartextblöcken. Die Anzahl möglicher Permutationen über $n$-Bit-Blöcken ist $2^n!$. Als Beispiel betrachte ich den DES mit einer Blockbreite von 64 Bit und einer signifikanten Schlüssellänge von 56 Bit. Um alle $2^{64}!$ Permutationen zu codieren sind $\mathrm{ld}\,(2^{64}!) \approx 2^{72}$ Bit nötig [1]. Eine Schlüssellänge von 56 Bit kann daher nur einen ganz kleinen Teil der Permutationen realisieren. Trotzdem können solche Chiffren als sicher betrachtet werden. Die Sicherheit dieser Chiffren beruht darauf, daß die Verschlüsselungsfunktion so "kompliziert" ist, daß aus der Kenntnis verschiedener

[1] Die Funktion ld steht für den Logarithmus zur Basis 2.

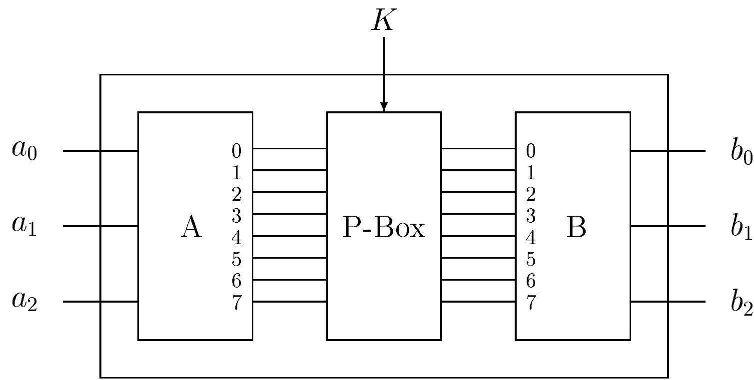



Abbildung 3: Substitutionsbox S

Klartext-, Schlüsseltextblöcke keine Rekonstruktion des Schlüssels oder anderer Klartext-, Schlüsseltextblöcke erfolgen kann. Die Wahrscheinlichkeit, daß gleiche Blöcke mehrfach auftreten, kann ebenfalls gering gehalten werden. Sie ist nicht gering, wenn ein natürlichsprachlicher Text übermittelt wird, bei dem Wörter oder Wortkombinationen mehrfach auftreten, deren Codierung mehr als 64 Bit benötigt. Dies erhöht die Wahrscheinlichkeit mehrerer gleicher Blöcke. Abhilfe schafft hier die Datenkompression, oder die Verwendung bestimmter Betriebsarten der Blockchiffre (siehe [Hors85, SePi89]).

Betrachtet man eine $n$-Bit Blockchiffre mit Schlüssel $K$, so ist die Verschlüsselung durch eine Funktion $e$ festgelegt:

$$e_K : \{0,1\}^n \rightarrow \{0,1\}^n$$

Ist $n$ relativ klein (z.B: $n = 3$), so läßt sich eine Substitutionschiffre durch zwei Wandler und eine Permutationsbox, kurz P-Box, realisieren. Wandler A ist ein $n : 2^n$-Decoder, der eine $n$-Bit Zahl in ein Signal auf der entsprechenden von $2^n$ Leitungen umwandelt. Wandler B ist ein Decoder der ein Eingangssignal auf einer von $2^n$ Leitungen in den entsprechenden $n$-Bit Index dieser Leitung umwandelt (siehe Abb. 3).

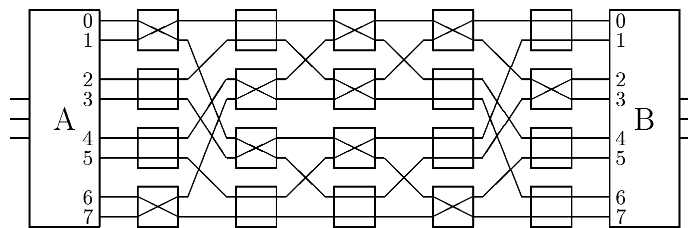



Abbildung 4: Ein Verbindungsnetzwerk als Substitutionschiffre

Der Unterschied zwischen Permutation und Substitution liegt nur in der Codierung. Die Substitution operiert auf einer $n$-Bit Binärzahl und die Permutation auf einem Vektor

$(00\dots010\dots00)$ der Länge $2^n$. Mathematisch betrachtet besteht kein Unterschied zwischen diesen beiden Mengen.

Zur Erzeugung einer veränderlichen, rekonfigurierbaren Permutation in der P-Box eignet sich ein Verbindungsnetzwerk. Abbildung 4 zeigt ein Beispiel einer solchen P-Box. Das Netzwerk realisiert die Permutation bzw. Substitution:

$$\pi = \begin{pmatrix} 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\ 2 & 4 & 6 & 7 & 0 & 1 & 5 & 3 \end{pmatrix}$$

Wie erwähnt, sind diese Chiffren unsicher! Die Länge des Alphabets ist beschränkt durch die wachsende Komplexität der beiden Wandler und der P-Box. Ein Zeichen des Alphabets wird immer durch dasselbe Chiffratzeichen substituiert. Bei längeren Texten ist dadurch eine statistische und strukturelle Kryptoanalyse möglich. Es muß daher noch eine weitere Funktion in die Chiffre eingehen. Diese kann beispielsweise den Schlüssel $K$ während der Verschlüsselung ändern.

## 2.3 Die Selbstmodifikation

Eine Möglichkeit, eine Substitutionschiffre der vorgestellten Art sicherer zu machen, besteht darin, den Schlüssel $K$ häufig zu wechseln. Im Verschlüsselungsbetrieb bedeutet das jedoch, daß eine große Anzahl von geheimen Schlüsseln $K_i$ zwischen den Kommunikationspartnern ausgetauscht werden müßte. Ein sicherer Austausch dieser Schlüssel erfordert aufwendige Sicherheitsvorkehrungen, deren Durchführung ebenfalls Probleme hervorruft. Um dies zu umgehen, gibt es Verfahren zur Schlüsselgenerierung. Meist werden dafür sogenannte *Pseudozufallsgeneratoren* verwendet. Ein PRG (*Pseudo-Random-Generator*) erzeugt, in Abhängigkeit verschiedener Parameter (z.B. Initialwert), eine Folge von Zahlen, die ähnliche Eigenschaften wie Zufallszahlen besitzen. Aus dieser Folge kann eine Schlüsselfolge generiert werden. Eine andere Möglichkeit ist die Selbstmodifikation des Schlüssels.

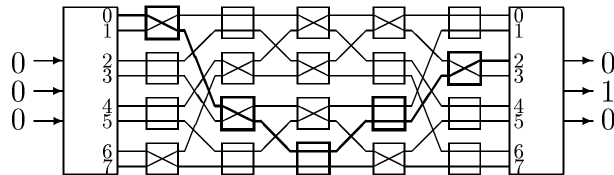
Das Verbindungsnetzwerk wird zu Beginn der Verschlüsselung mit dem Schlüssel $K = K_0$ initialisiert. Nach der Verschlüsselung jedes Zeichens modifiziert sich der Schlüsselzustand selbständig, d.h nur in Abhängigkeit vom derzeitigen Zustand, vom verschlüsselten Zeichen. Man spricht bei diesem Verfahren von einem *Auto-Key-System*. In die Modifikation gehen keine zusätzlichen Parameter ein. Ist $m_0, \dots, m_{l-1}$ die Folge der zu verschlüsselnden Zeichen, dann ergibt sich eine Zustandsfolge $K_t$ des Netzwerks, die durch eine Funktion $\delta$ festgelegt ist.

$$K_0 := K, \quad K_{t+1} := \delta(K_t, m_t) \quad \text{für } 0 \leq t < l$$

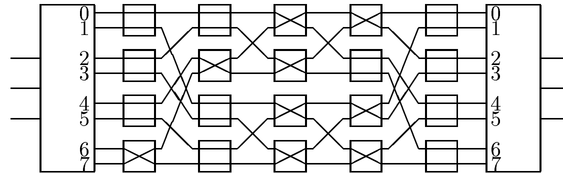
Schränkt man die Funktion $\delta$ dahingehend ein, daß sie nur von $K_t$ abhängig ist und nicht von $m_t$, so kann $\delta$ als Pseudozufallsgenerator betrachtet werden. Parameter der Schlüsselfolge ist dann nur der Initialschlüssel $K = K_0$. Dies ist für manche Verschlüsselungsprotokolle von Vorteil, bezüglich Synchronisation und Fehlerkorrektur.

Ein Beispiel für eine Selbstmodifikation $\delta$ in Abhängigkeit von $K_t$ und $m_t$ ist das pfadabhängige Invertieren (Kippen) der Tauscher. Es werden alle Kreuzschalter invertiert, die vom Signal des verschlüsselten Zeichens durchlaufen werden.

**Beispiel 2.1:**




Verschlüsselt man z.B. "0", erhält man als Chiffrat "2": Der Folgezustand des Netzwerks ist:




Diese Art der Selbstmodifikation erfüllt (insbesondere bei kleinen Netzwerken) nicht die notwendigen Bedingungen der kryptographischen Sicherheit. Bessere Modifikationsfunktionen werden in späteren Kapiteln vorgestellt und diskutiert.

## 2.4 Sequentielle Maschinen

Selbstmodifizierende Netzwerke sind Blockchiffren mit selbständig generierendem Schlüssel, sogenannte *Auto-Key*-Kryptosysteme. Diese lassen sich als sequentielle Maschinen beschreiben.

Eine sequentielle Maschine $\mathcal{M}$ ist festgelegt durch ein 6-Tupel $\mathcal{M} = (\Sigma, \Delta, Z, \delta, \lambda, z_0)$. Dabei bedeuten:

1. $\Sigma$ ist eine endliche Menge, das Eingabealphabet.
2. $\Delta$ ist eine endliche Menge, das Ausgabealphabet.
3. $Z$ ist die endliche Zustandsmenge.
4. $\delta$ ist eine Abbildung von $\Sigma \times Z \longrightarrow Z$ die Zustandsübergangsfunktion.
5. $\lambda$ ist eine Abbildung von $\Sigma \times Z \longrightarrow \Delta$ die Ausgabefunktion.
6. $z_0 \in Z$ ist der Startzustand.

Für die Anwendung auf das $\text{SINC}_n$ definiere ich die Nomenklatur:

- Ein- und Ausgabeblöcke der Chiffre: $\Sigma = \Delta = A = \{0, 1\}^n$.
- Der Klartext $M$ besteht aus einer Zeichenfolge der Länge $l$: $M = m_0, m_1, \ldots, m_{l-1} \in A^l$, der Schlüsseltext $C$ ist ebenfalls eine Zeichenkette der Länge $l$: $C = c_0, c_1, \ldots, c_{l-1} \in A^l$.

- Zustandsmenge sind die Matrizen der Tauscherstellung: $Z = \mathcal{K} = \{\{0,1\}^Z\}^S$. Die Anzahl der Zeilen ist dabei festgelegt durch: $Z = 2^{n-1}$. Die Anzahl der Spalten $S$ ist variabel, es wird jedoch gezeigt, daß $S = 2n-1$ eine geeignete Breite des Netzwerks ist.
- Der Folgezustand $K_{t+1}$ des Zustands $K_t$ wird durch die Funktion $K_{t+1} = \delta(m_t, K_t)$ bestimmt.
- Die Ausgabefunktion wird in der Kryptographie als Verschlüsselungsfunktion bezeichnet: $\lambda = e_{K_t}(m_t)$.
- Der Startzustand $z_0$ wird als $K_0$ (Initialschlüssel) bezeichnet

Für die eindeutige Entschlüsselung eines Textes ist es notwendig, daß $\lambda$ injektiv ist. Da $\Sigma = \Delta = A$ endlich sind, bedeutet dies gleichzeitig, daß $\lambda$ eine Bijektion (Permutation) ist. Damit kann $\mathcal{M} = (\Sigma, \Delta, Z, \delta, \lambda, z_0)$ als Verschlüsselungsmaschine betrachtet werden. Das Chiffrat $C = c_0, c_1, c_2, \ldots, c_{l-1}$ wird definiert durch:

$$c_t := \lambda(m_t, z_t), \ \ z_{t+1} = \delta(m_t, z_t) \ \text{ für } 0 \leq t \leq l-1$$

Da die Funktion $\lambda$ umkehrbar ist, lassen sich $\lambda'$ und $\delta'$ bestimmen, so daß gilt:

$$m_t := \lambda'(c_t, z_t), \ \ z_{t+1} = \delta'(m_t, z_t) \ \text{ für } 0 \leq t \leq l-1$$

Die dadurch festgelegte Maschine ist $\mathcal{M}' = (\Sigma, \Delta, Z, \delta', \lambda', z_0)$, und kann als Entschlüsselungsmaschine betrachtet werden.

# 3 Kryptographie

## 3.1 Eigenschaften des Kryptosystems

Ein klassisches (symmetrisches) Kryptosystem besteht aus einer Verschlüsselungsfunktion $E_K$, die, abhängig vom Parameter $K$, einen Klartext $M$ injektiv auf einen Schlüsseltext $C$ abbildet, sowie einer Funktion $D_K$, die diese Abbildung rückgängig macht.

Diese Darstellung soll hier genügen. Eine exakte Formulierung und eine Klassifikation der verschiedenen Kryptosysteme ist in Kryptographiebüchern (z.B. [Hors85]) nachzulesen.

In der Theorie von Kryptosystemen unterscheidet man verschiedene Klassen der Sicherheit. Für die Analyse des SINC ist die folgende Definition relevant: Ein Kryptosystem ist *praktisch sicher*, wenn kein Verfahren bekannt ist, welches das Kryptosystem mit den verfügbaren Ressourcen unter vertretbarem Kosten- bzw. Zeitaufwand brechen kann.

Die Schwierigkeit, die sich beim Entwurf von Kryptosystemen ergibt, besteht in der fehlenden Beweisbarkeit der Sicherheit. Es ist im allgemeinen nur möglich, die Unsicherheit eines Verschlüsselungssystems definitiv zu beweisen. Daß ein Kryptosystem nicht zu brechen ist, kann meist nur begründet werden. Häufig besteht die Begründung darin, daß das Brechen des Systems auf ein bekanntes mathematisches Problem reduzierbar ist. Als Beispiele seien

genannt: diskreter Logarithmus, Wurzelproblem, Faktorisierungsproblem. In der Informatik eignen sich dafür auch häufig NP-vollständige Probleme. Beispiele für NP-vollständige Probleme sind das Knapsackproblem, Erfüllbarkeit und viele andere. Es ist zwar unbewiesen, daß diese Probleme nicht effizient lösbar sind, die Tatsache jedoch, daß sich große Mathematiker und Informatiker an diesen Problemen versucht haben und nicht zu einer Lösung gefunden haben, wird häufig einem Beweis gleichgesetzt.

Leider ist eine solche Rückführung auf bekannte mathematische Probleme im Falle des SINC schwierig. Die Struktur dieses Netzwerks kann nur schwer mit den bekannten Problemen in Verbindung gebracht werden. Die Aussagen über die Sicherheit des SINC werden in dieser Arbeit daher nur durch Plausibilitätsbetrachtungen gestützt.

Das Brechen eines SINC ist o.B.d.A dem Problem gleichzusetzen, aus der Kenntnis mehrerer Zeichenpaare des Klar- und Schlüsseltextes (*Known* bzw. *Chosen Plaintext*) den Schlüssel $K$ oder einen dazu äquivalenten Schlüssel [2] $K'$ ganz oder teilweise herauszufinden. Damit ist es möglich, die anderen Schlüsseltextzeichen ganz oder teilweise zu entschlüsseln.

Notwendige Bedingungen für Kryptosysteme sind somit:

1. Das Kryptosystem SINC sollte *praktisch sicher* sein.

2. Die Funktionen $E_K$ und $D_K$ sollten einfach berechenbar oder technisch realisierbar sein, damit sie in kurzer Zeit ausführbar sind.

3. Bei einem zufällig gewählten Schlüssel $K$ sollte zwischen Klartext und Schlüsseltextzeichen keine Korrelation bestehen. Für die Wahrscheinlichkeit des Auftretens der einzelnen Zeichen muß gelten:
$$\forall x, y \in A : p(e_K(x) = y) = \frac{1}{|A|}$$
   Das ist die einfachste statistische Aussage, die über das Chiffrat zu machen ist. Weitere statistische Regelmäßigkeiten, sind Zyklen oder Autokorrelationen, die bei einer Folge auftreten können.

## 3.2 Digitale Unterschriften

Ein Anwendungsgebiet der Kryptographie, das derzeit stark an Bedeutung gewinnt, sind digitale Signaturverfahren. Durch sie ist es möglich, Dokumente zu unterschreiben und dadurch die Authentizität des Absenders zu beweisen. Analog erfolgt die Versiegelung von Software. Durch eine Signatur kann sichergestellt werden, daß eine Manipulation eines Programmes bei dessen (zeitlicher oder geographischer) Übertragung, erkannt werden kann.

Ein Prinzip digitaler Unterschriften basiert auf *Public-Key*-Kryptosystemen (PKK). Eine Möglichkeit besteht darin, ein PKK zu verwenden, bei dem Verschlüsselung und Entschlüsselung vertauschbar sind, d.h.
$$D(E(m)) = m = E(D(m))$$

[2] Zwei Schlüssel $K$ und $K'$ sind äquivalent, wenn $E_K = E_{K'}$.

Dies ist bei manchen Public-Key-Kryptosystemen gewährleistet. Als Beispiel sei das RSA-Verfahren genannt [Hors85]. Ist $n = p \cdot q$, mit $p, q$ prim, $p \neq q$ und $e \cdot d \equiv 1 \bmod (p-1)(q-1)$, so gilt

$$D(E(m)) = D(m^e \bmod n) = m^{e \cdot d} \bmod n = E(m^d \bmod n) = E(D(m))$$

Um ein Dokument zu signieren, "verschlüsselt" der Absender $A$ das Dokument mit der Entschlüsselungsfunktion $D_A$ seines PKK's. Diese Operation ist nur dem Sender $A$ möglich. Empfänger $B$ verifiziert die Unterschrift dadurch, daß er die Signatur mit der öffentlich bekannten Verschlüsselungsfunktion des Senders $A$ "entschlüsselt", und das Ergebnis mit dem Klartext vergleicht. Erfüllen die Funktionen $E$ und $D$ bestimmte Sicherheitskriterien, so ist die digitale Unterschrift *praktisch* unfälschbar. Hier verweise ich den interessierten Leser auf entsprechende Kryptographiebücher (z.B. [Hors85, SePi89]), in denen verschiedene PKK's und Signaturprotokolle beschrieben sind.

| Sender $A$ | Kanal | Empfänger $B$ |
|---|---|---|
| $S_A(M) := D_A(M)$ | $\rightarrow (M, S_A(M)) \rightarrow$ | $M = E_A(S_A)$ ? |

Signaturprotokoll

Möchte man ein solches Signaturverfahren auf längere Dokumente anwenden, so ergibt sich ein hoher Aufwand für die Berechnung und Übermittlung von Unterschriften, sowie die Verifikation. Aus diesem Grund verwendet man sogenannte *kryptographische Hash-* oder *Kontraktionsfunktionen*, die einen Text beliebiger Länge auf einen Hashwert $H$ der festen Länge $h$ abbilden.

$$H : \{0,1\}^* \rightarrow \{0,1\}^h$$

Es genügt dann die Unterschrift des Hashwertes als Signatur für den gesamten Text. Diese Verfahren sind auch unter dem Namen *finger-print* (Fingerabdruck) bekannt.

| Sender $A$ | Kanal | Empfänger $B$ |
|---|---|---|
| $S_A(M) := D_A(H(M))$ | $\rightarrow (M, S_A(M)) \rightarrow$ | $H(M) = E_A(S_A)$ ? |

Signaturprotokoll mit krypt. Hashfunktion

Eine kryptographische Hashfunktion muß, im Gegensatz zu einer Hashfunktion zur Speicherverwaltung, sehr strenge Forderungen erfüllen, um Sicherheit zu gewährleisten. Hat ein potentieller Angreifer einen Hashwert $H(M)$ und die zugehörige Signatur $S_A(H(M))$ zur Verfügung, so kann er die Signatur dazu nutzen, jeden anderen Text $M'$, der denselben Hashwert $H(M') = H(M)$ besitzt, mit der Unterschrift des Benutzers $A$ zu versehen. Daher darf es nicht möglich sein, unter vertretbarem Zeitaufwand zwei sinnvolle Dokumente mit demselben Hashwert zu finden. Eine Hashfunktion muß eine *Einwegfunktion* sein. Um hohe Sicherheit zu gewährleisten, wird für die Länge des Hashwertes $64 < h < 512$ gefordert. Derzeit zeichnet sich ein Trend zu einer Länge von 128 Bit ab.

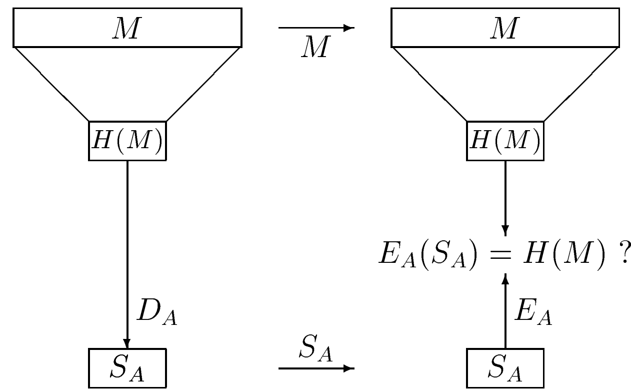



Abbildung 5: Prinzip der Signatur mit einer Hashfunktion

## 3.3 SINC$_n$ als Hashfunktion

Ein selbstmodifizierendes Netzwerk eignet sich als Basis einer Hashfunktion. Die einfachste Möglichkeit ist dabei, ein Dokument $M$ der Länge $l$ mittels SINC mit öffentlichem Initialschlüssel $K_0$ zu "verschlüsseln" und die Zustandsmatrix $K_l$ nach der Verschlüsselung als Hashwert zu verwenden.

$$H_{K_0}(M) := K_l$$

Die Länge des Hashwertes ist damit die Anzahl der Matrixelemente $h = Z \cdot S$. Soll $h$ kleiner sein, so kann der Wert durch eine geeignete Funktion kontrahiert werden. Auf dieses Verfahrens wird noch eingegangen. Auch die Möglichkeit anderer Betriebsarten, z.B. Rückkopplungen des Chiffrats, werden noch angesprochen.

Für die Anwendung eines SINC ist es wichtig, daß die Zustandsübergangsfunktion $\delta$ bestimmte Kriterien erfüllt. Die Funktion muß eine kollisionsresistente Einwegfunktion sein. Kollisionsresistent bedeutet hierbei, daß es *praktisch* unmöglich ist, zwei verschiedene Text $M$ und $M'$ zu finden, die denselben Hashwert besitzen. In Bezug auf das beschriebene Hashverfahren lassen sich folgende Kriterien für das selbstmodifizierende Netzwerk formulieren:

1. Die Funktion $\delta$ sollte einfach realisierbar sein, damit sie in kurzer Zeit ausführbar ist.

2. Es darf nicht in effizienter Zeit möglich sein, zu einem bekannten Hashwert ein anderes Dokument zu finden, das denselben Hashwert besitzt. Beim beschriebenen Verfahren bedeutet das, daß es nicht möglich sein darf, zu gegebenen Initial- und Finalmatrizen ein Dokument zu finden, das bei Verschlüsselung diese ineinander überführt.

3. Jede kleine Änderung des Dokuments muß sich "lawinenartig" auf den Hashwert, d.h. auf den Schlüsselzustand auswirken. Besonderes Augenmerk gilt hierbei der Vertauschung zweier benachbarter Blöcke des Dokuments.

4. Gibt es kurze Zyklen der Zustände ? D.h. ist es möglich, eine Zeichenkette in das Dokument einzufügen, die einen Zustand in sich selbst überführt ?

# 4 Die Vernetzung

## 4.1 Entwicklung einer sinnvollen Vernetzung

Nach den aufgeführten notwendigen Bedingungen für ein Kryptosystem, bzw. eine Hashfunktion in Kapitel 3, soll nun versucht werden, eine Vernetzung $\mathcal{V}$ zu konstruieren die diesen Bedingungen genügt. Aussagen über die kryptologische Sicherheit können dabei erst in Verbindung mit einer entsprechenden Selbstmodifikation gemacht werden. Die mathematische Theorie von Verbindungsnetzwerken beschreibt V.E. Beneš [Bene65]. An dieser Stelle genügt eine vereinfachte Darstellung der Netzwerke, die für diese Anwendung relevant sind. Zuerst ist es erforderlich, die Nomenklatur zur Beschreibung von Verbindungsnetzwerken festzulegen.

Der Zustand eines Verbindungsnetzwerks läßt sich durch eine Matrix beschreiben. Diese bezeichne ich mit $K$ (Key). Sie ist im folgenden der Schlüssel der Chiffre. Die Menge aller dieser Matrizen sei $\mathcal{K}$, sie wird auch als Schlüsselraum bezeichnet. Die Elemente einer Matrix $K$ seien $k(z,s) \in \{0,1\}$ für $z \in [0 : Z-1], \quad s \in [1 : S]$.

$$K := \begin{pmatrix} k(0,1) & k(0,2) & \dots & k(0,S) \\ k(1,1) & k(1,2) & \dots & k(1,S) \\ \vdots & \vdots & & \vdots \\ k(Z-1,1) & k(Z-1,2) & \dots & k(Z-1,S) \end{pmatrix} \tag{1}$$

In einer Spalte von Tauschern wird eine Permutation $P$ erzeugt. Ein Element aus $[0 : 2Z-1]$ wird durch diese Permutation entweder auf sich selbst abgebildet oder mit seinem Nachbarn vertauscht. Für die Permutation $P_s$ der Spalte $s$ $(1 \leq s \leq S)$ gilt:

$$\forall z \in [0 : 2Z-1] : \quad P_s^K(z) := \begin{cases} z + k(z \text{ div } 2, s) & \text{für } z \text{ gerade} \\ z - k(z \text{ div } 2, s) & \text{für } z \text{ ungerade} \end{cases}$$

oder:

$$\forall z \in [0 : 2Z-1] : \quad P_s^K(z) := 2(z \text{ div } 2) + (k(z \text{ div } 2, s) \oplus z) \tag{2}$$

Die Permutation besteht somit nur aus Zyklen der Länge 1 und 2. Es gilt daher: [3]

$$(P_s)^2 = id \quad \text{bzw.} \quad (P_s)^{-1} = P_s \tag{3}$$

Diese Permutationen sind also selbstinvers (involutorisch). Die Permutationen $Q_s$, $s \in [0 : S-1]$ geben die Vernetzung an und sind vom Schlüssel unabhängig. Die Vernetzung $\mathcal{V}$ wird definiert durch das Tupel von Permutationen $Q_i$. Um diese Vernetzung allgemein zu halten, sei eine Eingangspermutation $Q_0$ sowie eine Ausgangspermutation $Q_S$ zugelassen. Die Vernetzung ist damit das $S+1$-Tupel

$$\mathcal{V} := (Q_0, Q_1, \dots Q_S) \quad \text{mit} \quad Q_i \in S_{2Z}$$

Die Permutation $\pi$, die durch ein solches Verbindungsnetzwerk erzeugt wird läßt sich nun als Verkettung von Spaltenpermutationen darstellen. Es gilt:

$$\pi := Q_S \circ P_S \circ Q_{S-1} \circ P_{S-1} \circ \dots \circ Q_2 \circ P_2 \circ Q_1 \circ P_1 \circ Q_0$$

[3] Sind $l_1, l_2, \dots, l_k$ die Zyklenlängen einer Permutation $\pi$, dann gilt: $\pi^{\text{kgV}(l_1,l_2,\dots,l_k)} = id$

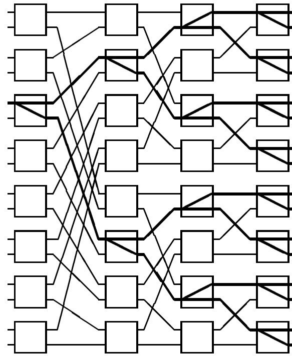


Abbildung 6: Netzwerk mit Binärbaumstruktur

Bei geeigneter Wahl der Vernetzung $\mathcal{V}$ ist es möglich, Netzwerke zu entwerfen, mit denen alle Permutationen aus $S_{2Z}$ erzeugbar sind.

### Größe des Netzwerks

Nach Konstruktion der Substitutionschiffre mittels der beiden Wandler $A$ und $B$ (siehe 2.2) folgt zwangsläufig, daß die Anzahl der Zeilen $Z = 2^{n-1}$ sein muß.

**Lemma 4.1:** Ein Verbindungsnetzwerk eines $\text{SINC}_n$ besteht aus $2^{n-1}$ Zeilen.

Für die Mindestanzahl der Spalten $S$ sind mehrere Überlegungen notwendig. Notwendige Bedingung ist, daß von jedem Eingang des Netzwerks zu jedem Ausgang des Netzwerks mindestens eine mögliche Verbindung, ein sogenannter *Pfad*, besteht. Betrachtet man die Pfade von einem bestimmten Eingang, so ergibt sich eine Verzweigung in jeder Tauscherspalte. Die Anzahl der verschiedenen Pfade eines Eingangs beträgt $2^S$. Strukturiert man das Netzwerk baumartig, so läßt sich erreichen, daß ein Netzwerk mit $S = n$ Spalten einen Pfad von jedem Eingang zu jedem Ausgang besitzt. Daraus folgt eine Mindesttiefe (Breite) des Netzwerks von $S = n$ Spalten (siehe Abb. 6).

Eine weitere Bedingung ist die technische Realisierbarkeit. Um die Entschlüsselungsfunktion mit demselben Baustein zu erzeugen, bietet sich an, die Vernetzung symmetrisch (zur senkrechten Achse) zu gestalten. Dies ermöglicht, daß die Umkehrfunktion, also die Entschlüsselungsfunktion, dadurch realisiert wird, daß das Netzwerk mit der gespiegelten Matrix initialisiert wird. [4] Diese Eigenschaft läßt sich formulieren durch

$$Q_{S-s} = (Q_s)^{-1}.$$

Betrachtet man ein symmetrisches Netzwerk mit $S$ Spalten ($S \geq n$), so zeigt sich, daß von einem Eingang $i$ zum selben Ausgang $i$ mindestens $2^{\lfloor \frac{S}{2} \rfloor}$ verschiedenen Pfade existieren. Dies

[4] Die Selbstmodifikation muß dann natürlich auch symmetrisch sein.

liegt daran, daß aufgrund der Symmetrie zu jedem Pfad der linken Hälfte mindestens ein Pfad der rechten Hälfte existiert (nämlich der symmetrische), so daß der Eingang $i$ auf sich selbst abgebildet wird. Die Anzahl der Pfade eines Eingangs $i$ in der linken Hälfte ist $2^{\lfloor \frac{S}{2} \rfloor}$. Soll nun die Gleichverteilung der Verbindungen gewährleistet sein, so müssen zwischen zwei beliebigen Ein- und Ausgängen jeweils dieselbe Anzahl an Pfaden bestehen. Von einem Eingang zu einem beliebigen Ausgang müssen daher mindestens $2^{\lfloor \frac{S}{2} \rfloor}$ Pfade bestehen. Die Gesamtanzahl der Pfade eines Eingangs muß daher mindestens $2^n \cdot 2^{\lfloor \frac{S}{2} \rfloor}$ betragen. Da die Anzahl der Pfade eines Eingangs $2^S$ ist, erhält man die Ungleichung:

$$2^S \geq 2^n \cdot 2^{\lfloor \frac{S}{2} \rfloor}$$

Löst man diese Ungleichung nach $S$, so erhält man den folgenden

**Satz 4.1:** Ein symmetrisches Verbindungsnetzwerk eines SINC$_n$, das eine feste Anzahl von Pfaden zwischen zwei beliebigen Ein- und Ausgängen besitzt, besteht aus mindestens

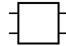
$$S \geq 2n - 1 \text{ Spalten.}$$

Es wird nun gezeigt, daß die Größe $S = 2n - 1$ auch ausreicht, um ein symmetrisches Netzwerk mit den genannten Eigenschaften zu konstruieren. Eine Möglichkeit ist das nach dem Mathematiker V.E. Beneš benannte Netzwerk [Bene65].

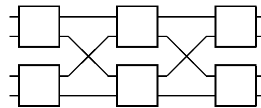
## 4.2 Beneš-Netzwerke

Unter einem Beneš-Netzwerk $B_n$ versteht man ein "rechteckiges" Tauschernetzwerk der Größe $(Z, S) = (2^{n-1}, 2n - 1)$, das folgendermaßen aufgebaut ist:

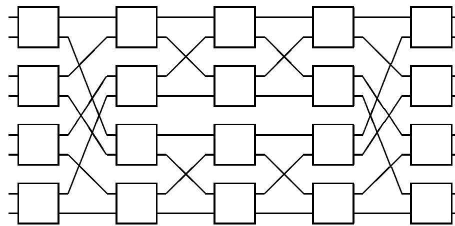
$B_1$:



$B_2$:



$B_3$:



$B_{n+1}$:

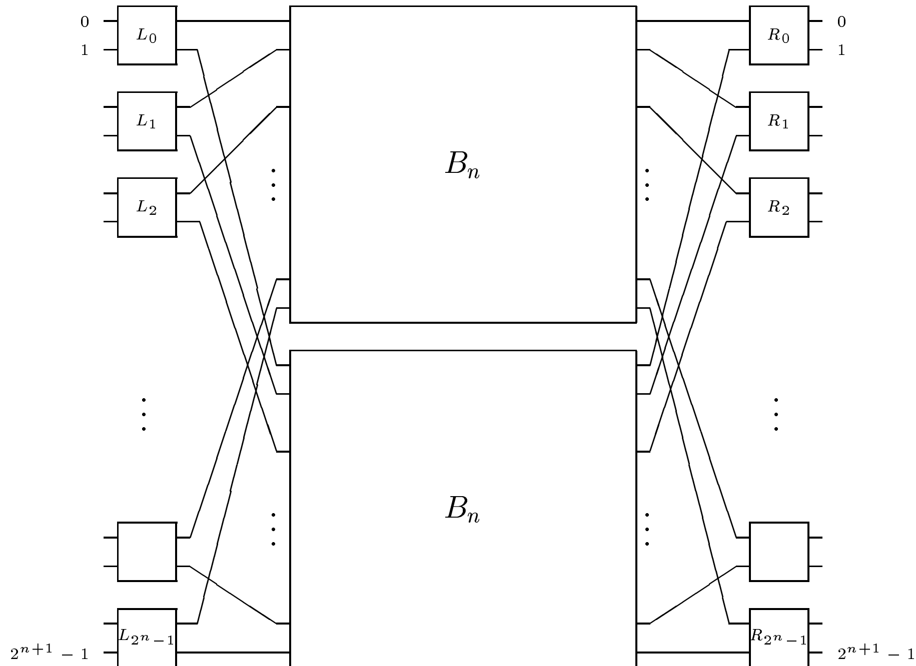



Abbildung 7: Rekursiver Aufbau von Beneš-Netzwerken

Die Entwicklung der Größe und die Struktur des Netzwerks ist in den Abbildungen 9 und 10 zu sehen.

**Satz 4.2:** Mit einem Beneš-Netzwerk $B_n$ läßt sich jede Permutation auf $[0 : 2^n - 1]$ realisieren.

Für diesen Satz möchte ich zwei verschiedene Beweise anführen.

**Beweis 4.1:**

Ich führe einen Induktionsbeweis:

Induktionsanfang: Für $n = 0$ stimmt die Behauptung (trivial).

Induktionsschritt: Die Behauptung stimme für $B_n$. Ich betrachte das Netzwerk rekursiv (siehe Abb. 7). Ziel ist es, die gewünschte Permutation so zu zerlegen, daß die Pfade zweier benachbarter Eingänge $(2i, 2i + 1)$ bzw. Ausgänge, durch verschiedene Unternetzwerke $B_n$ gehen.

Ich betrachte das Problem graphentheoretisch. Das Beneš-Netzwerk ist eine P-Box. Der Graph entsteht dadurch, daß je zwei benachbarte Eingänge und Ausgänge zu einem Knoten zusammengefaßt werden. Die Knoten des Graphen entsprechen damit den Tauschern auf der linken und rechten Seite $(L_0, \ldots, L_{2^n-1}, R_0, \ldots, R_{2^n-1})$. Die Kanten sind dann die entsprechenden Verbindungslinien. Formal: Zwischen $L_i$ und $R_j$ gibt es genau $|\{\pi(2i), \pi(2i+1)\} \cap \{2j, 2j+1\}|$ Kanten.

Beispiel: $\Pi := \begin{pmatrix} 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\ 3 & 0 & 1 & 2 & 4 & 6 & 7 & 5 \end{pmatrix}$

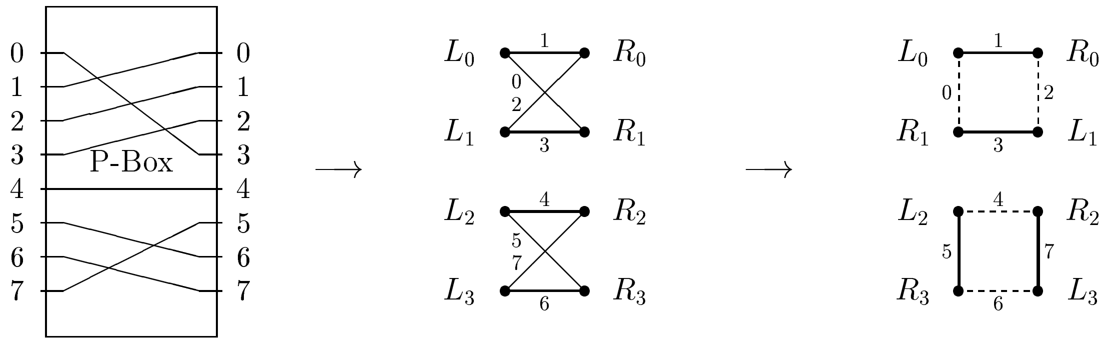



Abbildung 8: Konstruktion einer Tauscherstellung zu gegebener Permutation

Jeder Knoten hat den Grad 2. Jeder Zyklus des Graphen hat gerade Länge, da sich L-Knoten und R-Knoten abwechseln. Dadurch ist es immer möglich, die Kanten mit zwei Farben so zu färben, daß von jedem Knoten eine schwarze und eine weiße (im Beispiel gestrichelte) Kante ausgeht. Der Graph ist somit *zweifärbbar* oder *bipartit*. Jeder schwarzen Kante entspricht nun ein Pfad durch das obere Beneš-Netzwerk, jeder weißen ein Pfad durch das untere. Die Tauscher müssen noch entsprechend gestellt werden. In den beiden Unternetzwerken $B_n$ können alle Permutationen erzeugt werden. Damit ist eine Möglichkeit gefunden, die gewünschte Permutation mit dem Netzwerk $B_{n+1}$ zu erzeugen.

$$n \longrightarrow n+1 \quad \square$$

Im Beispiel sieht der Rekursionschritt so aus:

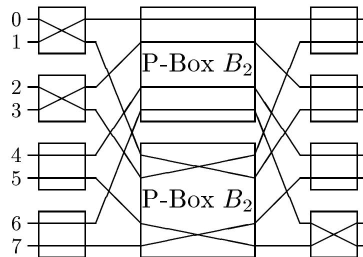



Abbildung 11: Rekursionsschritt

**Anmerkung:** Die Anzahl der Freiheiten bei einem Rekursionsaufruf entspricht der Anzahl der Zyklen des zugehörigen Graphen. Für jeden Zyklus gibt es zwei Möglichkeiten der “Färbung”. Man kann also jeweils einen Tauscher des Zyklus frei setzen, die anderen sind dadurch festgelegt.

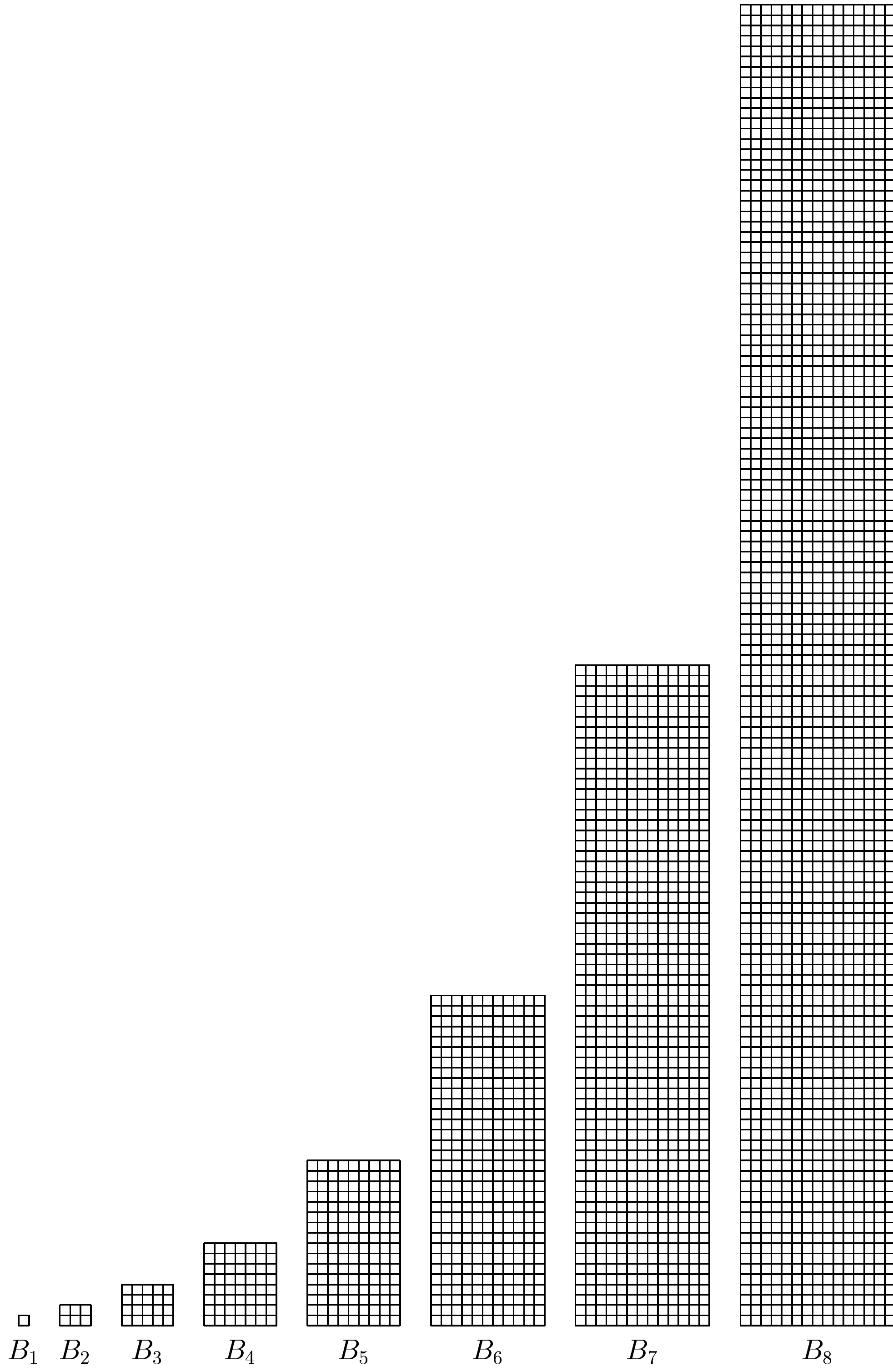



Abbildung 9: Größe der Beneš-Netzwerke

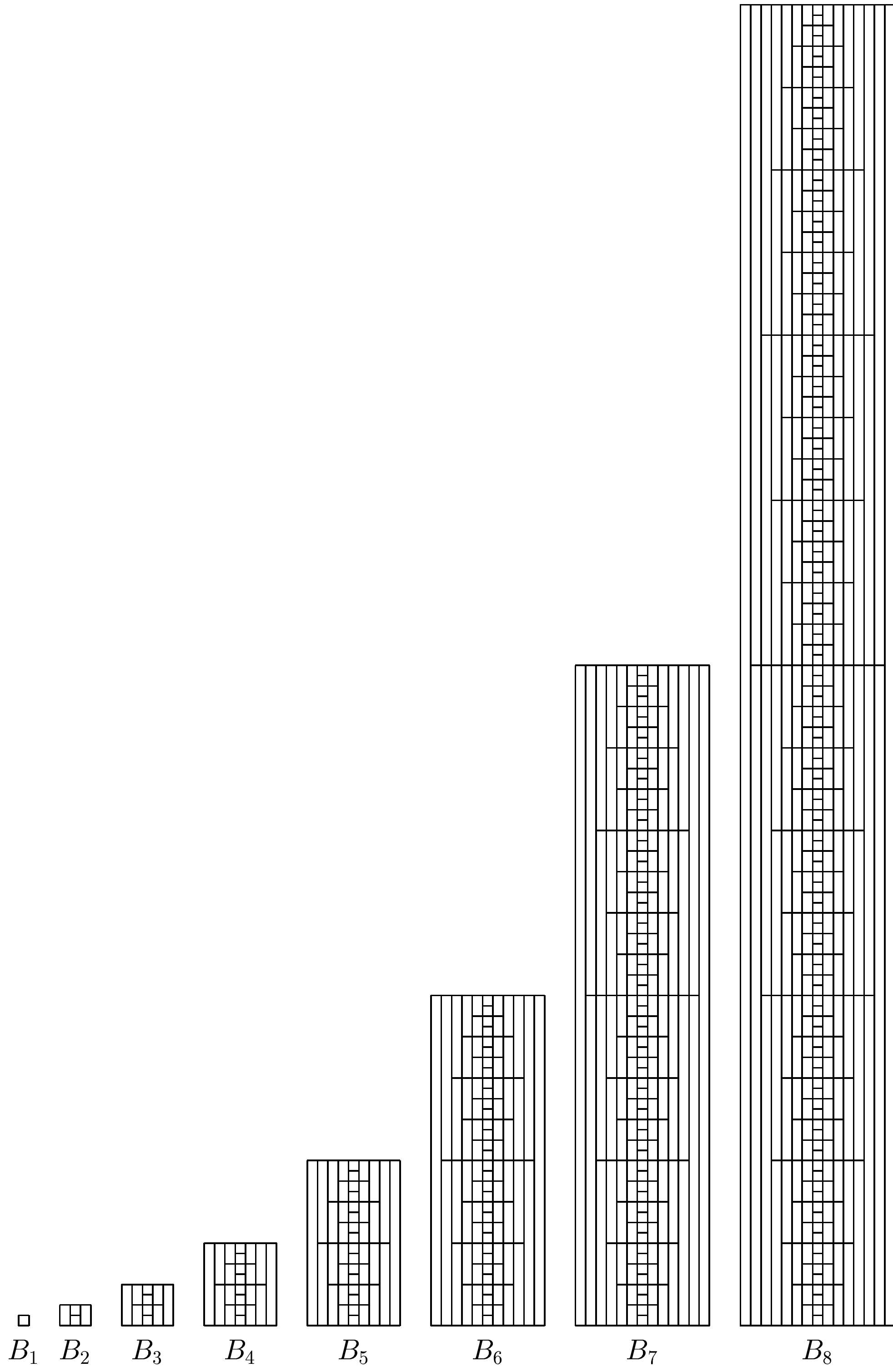



Abbildung 10: Rekursiver Aufbau der Beneš-Netzwerke

**Beweis 4.2:** (zu Satz 4.2)

Eine weitere Möglichkeit, die Vollständigkeit des Beneš-Netzwerks zu beweisen erfolgt durch eine andere Betrachtungsweise. Kann man mit einem Netzwerk alle Permutationen $\pi$ erzeugen, so lassen sich auch alle Umkehrpermutationen $\pi^{-1}$ erzeugen. Dasselbe gilt für die Umkehrung. Es genügt daher der Beweis, daß alle Permutationen $\pi^{-1}$ erzeugbar sind.

Somit soll eine Tauscherstellung konstruiert werden, mit der eine beliebige Permutation $\pi \in S_{2^{n+1}}$, die an der linken Seite des Netzwerks anliegt "sortiert" wird [5]. Der Beweis erfolgt ebenfalls rekursiv, durch vollständige Induktion. Ich zeige nur den Induktionsschritt $B_n \to B_{n+1}$.

Konstruktion: Die Binärdarstellung der Zahl $\pi(i) \in [0 : 2^{n+1}-1]$ sei $a_i|x_i$, mit $a_i \in \{0,1\}^n, x_i \in \{0,1\}$. O.B.d.A betrachtet man $\pi(0)$. Der linke obere Tauscher stehe auf '0' (parallel), d.h das Signal geht in das obere Teilnetzwerk $B_n^o$. Nun sucht man $\pi(i) = a_i|\overline{x_i}$. Dieses Element existiert genau einmal, da es sich um eine Permutation handelt. Ist $a_i|\overline{x_i} = \pi(1)$, so ist dieser Zyklus abgeschlossen und man setzt das Verfahren fort, indem man mit mit einem anderen Eingang beginnt. Ist $a_i|\overline{x_i} \neq \pi(1)$, so muß der entsprechende linke Tauscher $(1, i \text{ div } 2)$ so gestellt werden, daß das Signal das untere Netzwerk $B_n^u$ durchläuft. Das Verfahren wird dann mit $\pi(1)$ oder dem anderen Eingang von Tauscher $(1, i \text{ div } 2)$ fortgesetzt. Sind alle Tauscher der linken Seite festgelegt, so ist die Permutation so zerlegt, daß für alle $a_i \in \{0,1\}^n$ das Signal $a_i|\overline{x_i}$ und $a_i|x_i$ verschiedene Netzwerke $B_n^o$ und $B_n^u$ durchlaufen. Die Tauscherstellungen der rechten Spalte können nun entsprechend gesetzt werden.

An den Unternetzwerken liegt somit eine Permutation aus $S_{2^n}$ an, die nach Induktionsvoraussetzung "sortiert" werden kann. □

**Anmerkung:** Das Beneš-Netzwerk ist abgeschlossen, d.h.

$$B_n \circ B_n = B_n$$

Mit einem Beneš-Netzwerk können alle Permutationen aus $S_{2^n}$ erzeugte werden. Die Verkettung zweier Beneš-Netzwerke erzeugt eine dieser Permutationen. □

## 4.3 Mathematische Beschreibung des Beneš-Netzwerks

Im Hinblick auf eine Implementierung des Netzwerks ist es notwendig, eine allgemeine, mathematische Beschreibung der Vernetzungspermutationen $Q_i$ zu finden. Ich beschränke mich dabei auf die Beschreibung der Permutationen in der linken Hälfte des Netzwerks $(Q_1, \ldots, Q_{n-1})$. Die rechtsseitigen $(Q_n, \ldots, Q_{2n-2})$ ergeben sich aufgrund der Symmetrie durch Umkehrung der linken. Um eine geschlossene Formel für die Permutation $Q_s (1 \leq s \leq n-1)$ aufzustellen, betrachte ich das jeweilige Unternetzwerk $B_{n+1-s}$. Die erste Permutation dieses Netzwerks ergibt sich, indem man den Zeilenindex halbiert und, falls der Index ungerade war, einen Offset der halben Blockbreite addiert. Somit ergibt sich für die erste Permutation $Q_1$:

$$\forall z \in [0 : 2^n - 1] : \quad Q_1(z) := \begin{cases} z \text{ div } 2 & \text{für } z \text{ gerade} \\ z \text{ div } 2 + 2^{n-1} & \text{für } z \text{ ungerade} \end{cases}$$

[5]Vorsicht: Das Beneš-Netzwerk hat nichts mit Sortiernetzwerken zu tun.

Oder als geschlossene Formel:

$$\forall z \in [0 : 2^n - 1] : \quad Q_1(z) := z \text{ div } 2 + (z \bmod 2) \cdot 2^{n-1}$$

Die anderen Permutationen lassen sich auf diese zurückführen. Anstelle von $z$ tritt der entsprechende Index im betreffenden Unternetzwerk $z \bmod 2^{n+1-s}$. Den entsprechenden Offset des Unternetzwerks erhält man durch $(z \text{ div } 2^{n+1-s}) \cdot 2^{n+1-s}$. Dadurch erhält man eine geschlossene Formel für die Permutationen $Q_1, \ldots, Q_{n-1}$:

$$Q_s(z) := \underbrace{((z \text{ div } 2^{n+1-s}) \cdot 2^{n+1-s})}_{\substack{\text{Offset des} \\ \text{Unternetzwerks}}} + \underbrace{(z \bmod 2^{n+1-s}) \text{ div } 2 + ((z \bmod 2^{n+1-s}) \bmod 2) \cdot 2^{n-s}}_{\text{Permutation wie bei } Q_1}$$

Vereinfachen läßt sich dabei der Term $(z \bmod 2^{n+1-s}) \bmod 2 = z \bmod 2$. Damit erhält man die Formel:

$$Q_s(z) := ((z \text{ div } 2^{n+1-s}) \cdot 2^{n+1-s}) + (z \bmod 2^{n+1-s}) \text{ div } 2 + (z \bmod 2) \cdot 2^{n-s} \tag{4}$$

In dieser Form wurden die Permutationen im C-Programm von Anhang B implementiert. Die Permutationen $Q_n, \ldots, Q_{2n-2}$ ergeben sich aus der Umkehrung:

$$Q_s := Q^{-1}_{2n-1-s} \tag{5}$$

## 4.4 Zustände des Beneš-Netzwerks

Der Zustand eines Beneš-Netzwerk kann beschrieben werden durch eine $2^{n-1} \times 2n - 1$ Matrix (Schlüsselmatrix) mit Elementen aus $\{0, 1\}$.

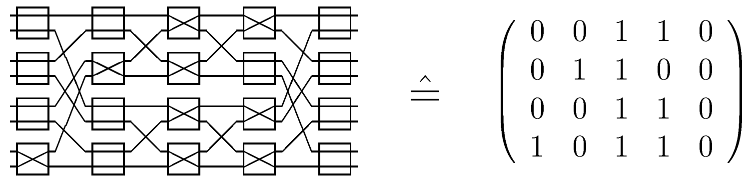


Abbildung 12: Zustandsmatrix eines Beneš-Netzwerks

Die Anzahl der Elemente der Matrix $n2^n - 2^{n-1}$ Bit legt die Anzahl der Zustände des Netzwerks fest. Diese Anzahl entspricht der Größe des Schlüsselraums $\mathcal{K}$:

$$|\mathcal{K}| = 2^{n2^n - 2^{n-1}}$$

Die Anzahl der möglichen Permutationen über $2^n$ ist

$$|S_n| = 2^n!$$

Man kann abschätzen:

$$2^{n2^n - 2^{n-1}} > 2^n! \quad \text{für } n \geq 2$$

Das Netzwerk kann mehr Zustände annehmen, als Permutationen erzeugen. Es muß somit verschiedene Zustände geben, die dieselbe Permutation erzeugen. Es stellt sich die Frage, wieviele Möglichkeiten es gibt, eine vorgegebene Permutation zu erzeugen. Eine untere Schranke ergibt sich, wenn man die Konstruktion (Beweis 4.2) eines Zustands betrachtet. In jedem rekursiven Aufruf, also solange $n > 1$ ist, kann man mindestens einen Tauscher frei belegen. O.B.d.A. sei dies der linke obere Tauscher.

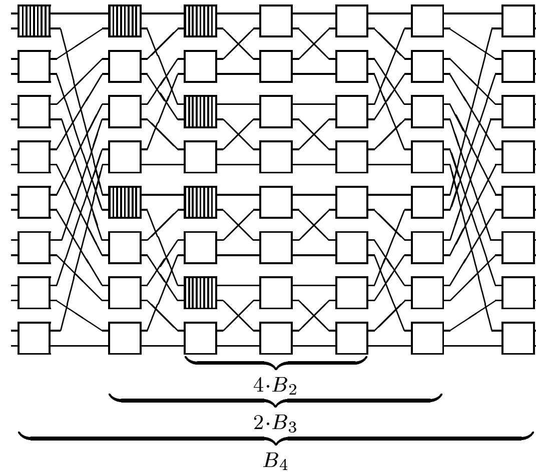



Abbildung 13: Mindestanzahl freier Tauscher

Die Anzahl $R$ der frei setzbaren (redundanten) Tauscher ist damit:

$$\min R = 1 + 2 + 4 + \ldots + 2^{n-2} = 2^{n-1} - 1$$

Damit hat man mindestens $2^{2^{n-1}-1}$ Möglichkeiten, diese Schalter zu setzen, um eine vorgegebene Permutation zu erzeugen. Eine obere Schranke ergibt sich durch eine analoge Abschätzung. Haben alle Zyklen des zugehörigen Graphen (siehe Beweis 4.2) kürzeste Länge, also die Länge 2, so lassen sich bei einem rekursiven Aufruf jeweils alle linken Tauscher frei belegen. Dies ist beispielsweise bei der identischen Abbildung der Fall.

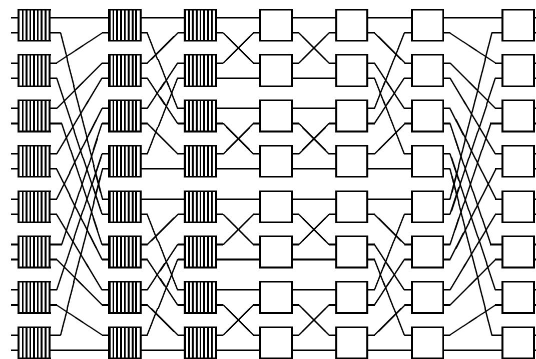


Abbildung 14: Maximale Zahl freier Tauscher

In diesem Fall kann man

$$\max R = 2^{n-1}(n-1)$$

Tauscher frei wählen. Die Anzahl der Möglichkeiten, eine Permutation zu erzeugen, ist also dann $2^{2^{n-1}(n-1)}$. Daraus ergibt sich der folgende

**Satz 4.3:** Für die Anzahl $R$ der frei setzbaren Tauscher eines Beneš-Netzwerks zu einer vorgegebenen Permutation gilt:

$$2^{n-1} - 1 \leq R \leq 2^{n-1}(n-1)$$

Die durchschnittliche Zahl der unabhängigen Tauscher läßt sich durch folgende Überlegung angeben. Es gibt $2^n!$ Permutationen. Die *signifikante* Schlüssellänge ist die durchschnittliche Anzahl der benötigten Bit zur Beschreibung einer Permutation. Es handelt sich informationstheoretisch um die *Entropie*. Sollen alle Permutationen mit derselben Wahrscheinlichkeit auftreten, so ist die Entropie:

$$H = \text{ ld } (2^n!)$$

Die Differenz von absoluter Schlüssellänge und Entropie ist die absolute Redundanz des Netzwerks und entspricht der durchschnittlichen Anzahl der freien Tauscher.

**Satz 4.4:** Die Anzahl der frei setzbaren Tauscher eines Beneš-Netzwerks ist durchschnittlich

$$T = 2^{n-1}(2n-1) - \text{ ld } ((2^n)!)$$

Mit Hilfe der Stirling'schen Formel

$$n! \approx \sqrt{2\pi n}(\frac{n}{e})^n$$

läßt sich der Wert abschätzen:

$$\begin{aligned} T &= 2^{n-1}(2n-1) - \text{ ld } ((2^n)!) \\ &\approx n2^n - 2^{n-1} - \text{ ld } (\sqrt{2\pi 2^n}) \left(\frac{2^n}{e}\right)^{2^n} \\ &= (\text{ ld } e - \frac{1}{2})2^n - \frac{n-1-\text{ ld } \pi}{2} \\ &\approx 0,943 \cdot 2^n - 0,5n - 1,326 \end{aligned}$$

Unter der relativen Redundanz $r$ des Netzwerks versteht man das durchschnittliche Verhältnis der freien Tauscher (Schlüsselbit) zur Gesamtzahl der Tauscher (Schlüssellänge). Obgleich für die Praxis nicht relevant, so ist doch bemerkenswert, daß $r$ für große $n$ gegen 0 strebt.

**Satz 4.5:** Für die relative Redundanz eines Beneš-Netzwerks gilt

$$r \approx \frac{0,943 \cdot 2^n - 0,5n - 1,326}{n2^n - 2^{n-1}}$$

und

$$\lim_{n \to \infty} r = 0$$

Eine weitere Größe ist die Anzahl der Freiheiten bei einem einzelnen Rekursionsaufruf. Nach Konstruktion der Tauscherstellungen hat man hier mindestens einen freien Tauscher. Dies ist der Fall, wenn der zugehörige Graph genau aus einem Zyklus besteht. Im Maximalfall (z.B bei der Identität) kann man alle $2^{n-1}$ Eingangstauscher frei belegen. Die durchschnittliche Anzahl läßt sich auf folgende Art berechnen. Die Anzahl der möglichen Belegungen der äußeren Spalten eines $B_n$ ist $2^{2^n}$. Dazu gibt es $2^{n-1}!^2$ signifikante Belegungen der Unternetzwerke. Daher ergibt sich für die Anzahl der Belegungen zur Erzeugung aller Permutationen

$$2^{2^n} \cdot (2^{n-1}!^2)$$

Da es nur $2^n!$ Permutationen gibt, ist die durchschnittliche Zahl der möglichen Belegungen für eine Permutation

$$\frac{2^{2^n} \cdot (2^{n-1}!^2)}{2^n!}$$

Der Zweierlogarithmus dieses Wertes ergibt die durchschnittliche Anzahl der frei belegbaren Tauscher in den äußeren Spalten.

**Satz 4.6:** Die durchschnittliche Anzahl frei belegbarer Tauscher beim ersten Rekursionsschritt ist

$$\text{ld } \frac{2^{2^n} \cdot (2^{n-1}!^2)}{2^n!} = 2^n + 2 \text{ ld } (2^{n-1}!) - \text{ ld } (2^n!)$$

Für den speziellen Fall des $\text{SINC}_8$ ergeben sich durchschnittlich 20 mögliche Belegungen und somit ld $20 = 4,33$ frei setzbare Tauscher in der ersten Spalte.

Wählt man die Schlüsselmatrix zufällig, so ist die Wahrscheinlichkeit, daß damit die Permutation $\pi$ erzeugt wird, nicht für alle $\pi$ gleich. Dies folgt daraus, daß zwar alle Zustände gleich wahrscheinlich sind, aber die Anzahl der Zustände, mit der eine Permutation erzeugt werden kann, verschieden ist. Die folgende Tabelle aus [Vick92] illustriert am Beispiel $B_3$, wie ungleich die Häufigkeiten der verschiedenen Permutationen verteilt sind. Die Anzahl der Permutationen ist 8!, die Anzahl der möglichen Zustände des $B_3$ ist $2^{20}$.

| # Permutationen | Häufigkeit |
|---|---|
| 8192 | 8 |
| 14336 | 16 |
| 12288 | 32 |
| 2048 | 40 |
| 2816 | 64 |
| 512 | 128 |
| 128 | 256 |
| $\Sigma\#P = 40320 = 8!$ | |
| $\Sigma\#P \cdot H = 1048576 = 2^{20}$ | |
| $\emptyset H = \frac{\Sigma\#P\cdot H}{\Sigma\#P} \approx 26$ | |

Tabelle 1: Auftrittshäufigkeiten von Permutationen in $B_3$

Diese Verteilung widerspricht jedoch nicht der geforderten Gleichverteilung des Chiffrats. Bei Verwendung eines selbstmodifizierenden Beneš-Netzwerks als Permutation einer S-Box besteht in allgemeinen kein äußerer Zugriff auf die gesamte Permutation, sondern nur auf ein einzelnes Element dieser Permutation [6]. Nach der Verschlüsselung ändert sich die Konfiguration des Netzwerks. Ist die Modifikationsfunktion gut gewählt, so lassen sich dadurch nur geringe Rückschlüsse auf die Gesamtpermutation ziehen.

Die geforderte Gleichverteilung über den einzelnen Elementen einer Permutation ist gewährleistet, wenn die Belegung der Tauscher (pseudo-) zufällig gewählt ist. Die Wahrscheinlichkeit dafür, daß $i = \pi(j)$ ist, ist für alle $i, j \in [0 : 2^n - 1]$ konstant.

**Satz 4.7:** Die Wahrscheinlichkeit, daß bei zufälliger Initialisierung des Netzwerks ein Eingang $i$ auf einen Ausgang $j$ abgebildet wird, ist gleichverteilt.

$$\forall i, j : \; p(i = \pi(j)) = \frac{1}{2^n}$$

Begründung: Diese Gleichverteilung wird nach den ersten $n$ Spalten des Beneš-Netzwerks erreicht. Von einem Eingang $i$ des Netzwerks gibt es genau einen Pfad zu jedem Ausgang der mittleren Spalte des Netzwerks.

[6] Vorsicht! Ein Zugriff auf mehrere Elemente der Initialpermutation kann bestehen, wenn verschiedene, dem Angreifer bekannte Dokumente mit demselben Initialschlüssel $K_0$ verschlüsselt werden. Dies muß durch ein entsprechendes Verschlüsselungsprotokoll ausgeschlossen werden.

**Beispiel 4.2:** $n = 4$

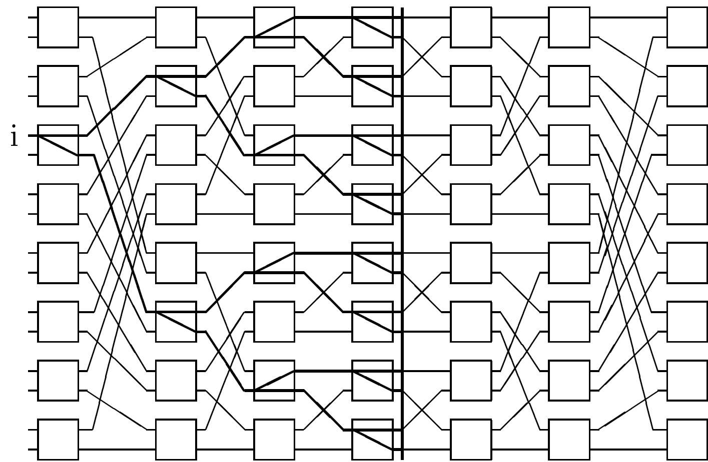



Abbildung 15: Binärbaumstruktur des Netzwerks

Da die Pfade von einem Eingang bis zum Ausgang der mittleren Spalte binärbaumartig strukturiert sind und die Tauscher, nach Voraussetzung, zufällig gestellt sind, ist das Signal an den "Blättern" des Baumes gleichverteilt. Die rechten Spalten $(n, \ldots, 2n-1)$ erzeugen jeweils eine Permutation, die ebenfalls zufällig und von den Spalten $(1, \ldots, n-1)$ unabhängig ist. Durch diese Permutationen bleibt die Gleichverteilung erhalten.

## 4.5 Beneš-äquivalente Netzwerke

Die rekursive Struktur des Beneš-Netzwerks kann sich, in Verbindung mit bestimmten Selbstmodifikationen, negativ auf die kryptographischen Eigenschaften auswirken. Besteht eine starke Korrelation zwischen Modifikationsfunktion und Vernetzung so werden dadurch Regelmäßigkeiten erzeugt, die einen Angriff auf das System erleichtern. Aus diesem Grund ist es sinnvoll, die Kreuzschalter des Beneš-Netzwerks anders zu positionieren und die Vernetzung etwas zu modifizieren, ohne die Eigenschaften des Netzwerks und die Topologie zu verändern. Ich definiere drei mögliche Operationen auf der Vernetzung, die die Eigenschaften der Beneš-Vernetzung invariant lassen.

### 1. Eingangspermutation

Das Vorschalten einer beliebigen Eingangspermutation $IP \in S_{2^n}$ (Initialpermutation) ändert nichts an den kryptographischen Eigenschaften des Netzwerks. Sie kann jedoch Regelmäßigkeiten des Klartextes schon vor der eigentlichen Verschlüsselung verringern. Auch andere Chiffren (z.B. DES) verwenden diese Operation.

**2. Vertauschung von Eingängen, bzw Ausgängen der Kreuzschalter**

Eine Vertauschung der beiden Eingänge (bzw. Ausgänge) eines Kreuzschalters ändert die topologische Struktur des Netzwerks nicht, da diese Operation durch Invertierung des entsprechenden Tauschers aufgehoben wird. Bei der technischen Realisierung kann diese Operation dazu dienen, Kreuzungen der Leitungen zu eliminieren. Ich bezeichne die Gruppe dieser Permutationen $P$ als $\mathcal{P} \subset S_{2^n}$. Die Permutationen $P$ lassen sich wie die Permutation einer Tauscherspalte (siehe 2, S.14) definieren:

$$\forall i \in [0 : 2^n - 1] : \quad P(i) := 2(i \text{ div } 2) + (t(i \text{ div } 2) \oplus i) \text{ mit } t(i \text{ div } 2) \in |0, 1|$$

**3. Permutation von Tauschern innerhalb einer Spalte**

Permutiert man die Tauscher innerhalb einer Spalte unter Beibehaltung der Verbindungen ("Gummibandprinzip"), so bleibt die Topologie des Netzwerks invariant. Darstellbar ist diese Operation durch eine Tauscherpermutation $T$, die vor einer Tauscherspalte $P^i$ ausgeführt wird, und der nachfolgenden Umkehrpermutation $T^{-1}$. Die Gruppe dieser Permutationen $T$ bezeichne ich mit $\mathcal{T}$. Ist $T' \in S_{2^{n-1}}$ die Permutation der Tauscher einer Spalte, so ist die entsprechende Vernetzungspermutation $T$ darstellbar durch

$$\forall i \in [0 : 2^n - 1] : \quad T(i) := 2T'(i \text{ div } 2) + (i \bmod 2) \text{ mit } T' \in S_{2^{n-1}}$$

Die Klasse der Beneš-äquivalenten Vernetzungen werden durch diese Operationen definiert. Eine symmetrische Vernetzung $\mathcal{V}$ mit der Eingangspermutation $V_0$ eines $(2^{n-1}) \times (2n - 1)$ Netzwerks ist darstellbar durch ein $2n$-Tupel:

$$\mathcal{V} = (V_0, V_1, V_2, \ldots, V_{n-1}, V_{n-1}^{-1}, \ldots, V_2^{-1}, V_1^{-1}, V_0^{-1})$$

Die Beneš-Vernetzung entspricht dem Tupel

$$\mathcal{V}_{\text{Beneš}} = (\text{id}, Q_1, Q_2, \ldots, Q_{n-1}, Q_{n-1}^{-1}, \ldots, Q_2^{-1}, Q_1^{-1}, \text{id})$$

Ich definiere die Klasse der Beneš-äquivalenten Vernetzungen als Vernetzungen, die durch genannte Operationen aus dem Beneš-Netzwerk entstehen. Ein Verfahren zur Erzeugung Beneš-äquivalenter Vernetzungen ergibt sich ebenfalls aus der Definition. Wählt man beliebige Operationen der genannten Art und wendet sie auf die Netzpermutationen an, so erhält man ein Beneš-äquivalentes Netzwerk.

**Satz 4.8:** Eine Vernetzung

$$\mathcal{V} = (V_0, V_1, V_2, \ldots, V_{n-1}, V_{n-1}^{-1}, \ldots, V_2^{-1}, V_1^{-1}, V_0^{-1})$$

ist *Beneš-äquivalent*, genau dann wenn

$$V_0 = IP, \quad V_i = PE_{i+1} \circ T_{i+1} \circ Q_i \circ T_i^{-1} \circ PA_i \text{ für } 1 \leq i \leq n-1$$

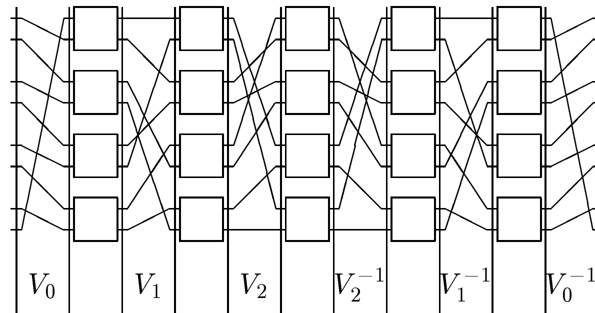
mit

$$IP \in S_{2^n}, PE_2, PE_3, \ldots, PE_n \in \mathcal{P}, PA_1, PA_2, \ldots, PA_{n-1} \in \mathcal{P}, T_1, T_2, \ldots, T_{n-1} \in \mathcal{T}$$

Anmerkung: Die Tauscherpermutation $T_0$ der Eingangsspalte sowie eine Vertauschung der Eingänge dieser Tauscher $PE_1$ wird linksseitig durch die Eingangspermutation absorbiert. Dasselbe gilt für die Ausgangsseite.

**Beispiel 4.3:**

Die folgende Vernetzung ist Beneš-äquivalent.




Die Netzpermutationen lassen sich darstellen durch

$$V_0 = IP, \quad V_1 = PE_2 \circ T_2 \circ Q_1 \circ T_1^{-1} \circ PA_1, \quad V_2 = PE_3 \circ T_3 \circ Q_2 \circ T_2^{-1} \circ PA_2$$

mit $IP \in S_8$, $PE_2, PE_3, PA_1, PA_2 \in \mathcal{P}$, $T_1, T_2, T_3 \in \mathcal{T}$ wie in Abb.16 dargestellt.

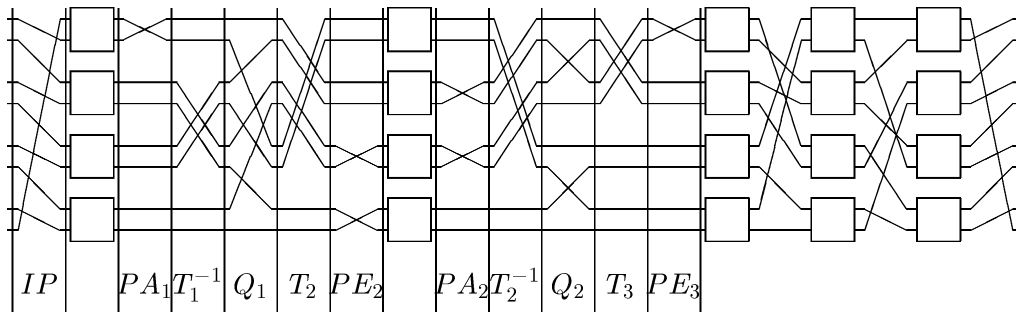



Abbildung 16: Beneš-äquivalente Vernetzung

**Anmerkung zur Realisierung**

Beim Prototyp des $SINC_8$ erfolgte die Vernetzung der Tauscher zwischen der ersten und zweiten Spalte, sowie zwischen den letzten beiden Spalten zweischichtig. In der einen Schicht liegen (waagrecht) 1920 Bahnen der Ausgänge einer Spalte und, dazu versetzt, 1920 Bahnen der Eingänge. Die zweite Schicht besteht aus senkrechten Verbindungsbahnen zwischen diesen Ein- und Ausgängen (siehe [HoNP91]).

Auf diese Art läßt sich jede Vernetzungspermutation technisch realisieren. Der Aufwand der Vernetzung, hier die Breite der zweiten Schicht, hängt davon ab, wie viele Bahnen notwendig sind, um jeden Eingang mit dem entsprechenden Ausgang zu verbinden. Verbindungen, die sich nicht überlappen, können dabei in einer Bahn liegen. Der schlechteste Fall (*worst case*) ist erreicht, wenn jede Verbindung mit jeder anderen überlappt.

Beim Beneš-Netzwerk sind die Permutationen $Q_1$ und $Q_{2^n-2}$ am aufwendigsten zu realisieren. Bei diesen Permutationen überlappen sich die Hälfte aller Verbindungen (zwischen den mittleren beiden Zeilen).

Der Aufwand der Realisierung einer anderen Vernetzung wird daher, gegenüber des schon realisierten Prototyps, nur geringfügig ansteigen.

# 5 Selbstmodifikationsfunktionen

Unter der Maßgabe, daß für die Vernetzung $\mathcal{V}$ die Beneš-Vernetzung, oder eine dazu äquivalente Vernetzung gewählt wird, soll nun versucht werden, eine geeignete Funktion $\delta$ für die Selbstmodifikation zu finden. Diese wird neben der technischen Realisierbarkeit vor allem unter dem Gesichtspunkt der kryptographischen Sicherheit zu betrachten sein.

Die einzige notwendige Bedingung für die Funktion $\delta$ ergibt sich aus der Symmetrie des Netzwerks. Die Umkehrung der Verschlüsselungsoperation soll durch Spiegelung der Initialmatrix $K_0$ erzeugt werden. Es sei $\overline{K}$ die zu $K$ gespiegelte Matrix. Damit die Folgematrizen der Entschlüsselung symmetrisch zu Verschlüsselungsmatrizen bleiben, muß $\delta$ eine Symmetrie im folgenden Sinne besitzen:

$$K_{t+1} = \delta(m_t, K_t) \wedge \; c_t = e_{K_t}(m_t) \quad \Longrightarrow \quad \overline{K_{t+1}} = \delta(c_t, \overline{K_t})$$

Trotz dieser Einschränkung existieren vielfältige Möglichkeiten der Selbstmodifikation. Die Folgezustandsfunktion $\delta$ kann abhängig sein von $K_t, m_t, t$ und vom Weg des Signals innerhalb des Netzwerks. Ich werde versuchen, elementare Selbstmodifikationsfunktionen mathematisch zu beschreiben und zu klassifizieren. Eine Klasse solcher Funktionen ist die pfadabhängige Invertierung

## 5.1 Pfadabhängige Invertierung

Unter einem Pfad $\mathcal{P}_t$ versteht man die Menge der durchlaufenen Tauscher eines Signals $m_t$. Der Pfad läßt sich iterativ definieren durch die Eingangsindizes des Signals.

$$m_t^1 := V_0(m_t), \quad m_t^{s+1} := V_s(P_s(m_t^s)) \text{ für } 1 \leq s \leq 2n-1$$

Der Pfad ist dann die Menge:

$$\mathcal{P}_t := \bigcup_{i=1}^{2n-1} \{(i, m_t^i/2)\}$$

Die pfadabhängige Invertierung zeichnet sich dadurch aus, daß aus der Menge $\mathcal{P}_t$ die Menge der Tauscher festgelegt wird, die invertiert werden. Dies ermöglicht eine einfache Realisierung, indem man die einzelnen Tauscher mit einem "Toggle"-Eingang versieht. Die Folgezustandsfunktion $\delta$ läßt sich durch eine Invertierungsmatrix beschreiben. Die Invertierungsmatrix $I$ läßt sich aus $\mathcal{P}_t$ berechnen. Die Einsen der Matrix entsprechen den Tauschern, die invertiert werden, die Nullen stehen für die Tauscher, die unverändert bleiben. Die Folgezustandsfunktion $\delta$ ist darstellbar durch bitweise Addition von $K_t$ und $I(\mathcal{P}_t)$:

$$\delta(K_t, m_t) := K_t \oplus I(\mathcal{P}_t)$$

## 5.2 Einfache Pfadmodifikation

Eine naheliegende, technisch einfach zu realisierende Selbstmodifikation ist die Invertierung der Tauscher des Verschlüsselungspfades. Jeder Tauscher, der vom Signal des Zeichens $m_t$ durchlaufen wurde, wechselt seinen Zustand. Die Invertierungsmatrix dieser pfadabhängigen Invertierung ist:

$$\forall z \in [0 : Z-1], s \in [0 : S] : i(z,s) := \chi_{\mathcal{P}}((z,s))^7$$

Ich werde diese Selbstmodifikation im weiteren mit EPM bezeichnen. Der am E.I.S.S. entwickelte Prototyp des $\text{SINC}_8$ arbeitet nach diesem Prinzip. Ein Beispiel für eine solche Verschlüsselung ist auf Seite 9 zu sehen.

Die einfache Pfadmodifikation ist kryptographisch unsicher. Ich will an diesem Beispiel zeigen, welche Regelmäßigkeiten und Invarianten bei dieser Art von Auto-Key-Systemen auftreten können und wie diese dazu genutzt werden können, das System zu brechen.

Zu beachten ist, daß sich diese Modifikation, abgesehen von einer möglichen Einganspermutation, an die rekursive Struktur des Netzwerks hält. Es ist daher belanglos, ob statt des Beneš-Netzwerks eine äquivalente Vernetzung gewählt wird. Es genügt deshalb in diesem Kapitel die Analyse der Beneš-Vernetzung.

[7] $\chi_M$ ist die charakteristische Funktion von M: $\chi_M(x) = 1$ für $x \in M$, $\chi_M(x) = 0$ sonst.

### 5.2.1 Eigenschaften der einfachen Pfadmodifikation

#### Abgeschlossenheit

**Satz 5.9:** Das $\mathrm{SINC}_n$ mit einfacher Pfadmodifikation ist nicht abgeschlossen:

$$\mathrm{SINC}_n \circ \mathrm{SINC}_n \neq \mathrm{SINC}_n$$

Beweis: Durch die Verkettung kann die Identität hergestellt werden.

$$\mathrm{SINC}_n \circ \mathrm{SINC}_n^{-1} = id$$

Ein einzelnes SINC kann die identische Abbildung jedoch nicht erzeugen. Dies kann an einem $\mathrm{SINC}_2$ durch Testen aller Fälle nachgewiesen werden. Für $n > 2$ kann dies durch den rekursiven Aufbau auf den $\mathrm{SINC}_2$ zurückgeführt werden.

#### Parität der Zustände

Die Parität zweier Spalten eines selbstmodifizierenden Verbindungsnetzwerks bleibt durch die Verschlüsselung unverändert. In jeder Spalte des Netzwerks wird genau ein Tauscher gekippt. In zwei Spalten werden genau zwei Tauscher invertiert. Dadurch bleibt die Parität der Matrixelemente der beiden Spalten unverändert. Dies gilt natürlich auch in jedem Unternetzwerk.

Mathematisch: Gegeben ist ein $\mathrm{SINC}_n$ mit Schlüsselmatrix $K$. Die Parität der Spalte $s$ $(0 \leq s \leq 2n-1)$ des Netzwerks ist definiert durch:

$$\mathrm{Par}_s(K) := \bigoplus_{z=0}^{2^{n-1}-1} k(z,s)$$

Für den Zugriff auf die Parität der linken Spalten der Unternetzwerke definiere ich die Funktion $\mathrm{Par}_{s,b}(K)$ mit $0 \leq s \leq n-1,\ 0 \leq b \leq 2^{s-1}-1$ Diese Funktion liefert die Parität der linken sowie der mittleren Spalte eines Unternetzwerks.

$$\mathrm{Par}_{s,b}(K) := \bigoplus_{z=b\cdot 2^{n-s}}^{(b+1)\cdot 2^{n-s}-1} (k(z,s) \oplus k(z,n))$$

Die Erweiterung dieser Funktion auf die Parität der rechten Spalten mit der entsprechenden mittleren gilt für $n+1 \leq s \leq 2n-1,\ 0 \leq b \leq 2^{n-s-1}-1$:

$$\mathrm{Par}_{s,b}(K) := \bigoplus_{z=b\cdot 2^{s-n-1}}^{(b+1)\cdot 2^{s-n-1}-1} (k(z,s) \oplus k(z,n))$$

In Abbildung 17 wird die Aufteilung der Paritätsfunktionen am Beispiel $n = 4$ dargestellt.

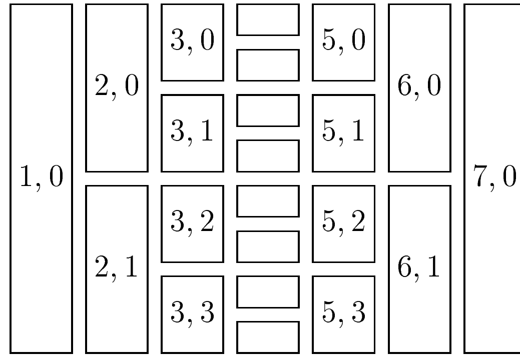



Abbildung 17: Äußere Spalten jedes Unternetzwerks $(s, b)$

Da bei einer Verschlüsselung mit der einfachen Pfadmodifikation in jeder Spalte genau ein Tauscher gekippt wird, bleibt die Parität zweier Spalten erhalten. Es gilt der

**Satz 5.10:** Bei einem $\text{SINC}_n$ mit Beneš-Vernetzung und einfacher Pfadmodifikation bleibt die Parität zweier Spalten eines Unternetzwerks invariant.

$$\text{Par}_{s,b}(K_t) = \text{Par}_{s,b}(K_{t+1})$$

Aus dieser Invariante folgt, daß bei dieser Art der Selbstmodifikation aus einem Startzustand $K_0$ nicht alle Schlüsselzustande $K \in \mathcal{K}$ erreicht werden können, sondern höchstens die mit denselben Paritäten $\text{Par}_{s,b}(K_0)$. Es gibt $2^n - 2$ zulässige Parameter $s, b$. Die Funktionen $\text{Par}_{s,b}$ sind voneinander unabhängig. Das bedeutet, daß sich die Anzahl der erreichbaren Schlüsselzustände $K \in \mathcal{K}$ nach oben abschätzen läßt. Die Menge $\mathcal{K}$ läßt sich in $2^{2^n-2}$ Äquivalenzklassen aufteilen. Stellt man den $\text{SINC}_n$ als endlichen Automaten dar, stellen die Äquivalenzklassen die Zusammenhangskomponenten des zugehörigen Graphen dar. Als Beispiel dient der endliche Automat des $\text{SINC}_2$ mit einfacher Pfadmodifikation, dessen Graph in Abbildung 18 dargestellt ist. Er besitzt $2^{2^2-2} = 4$ Zusammenhangskomponenten.

**Satz 5.11:** Bei einem $\text{SINC}_n$ mit Beneš-Vernetzung und einfacher Pfadmodifikation gilt für die möglichen Zustände, die aus einem Startzustand $K_0$ erreichbar sind:

$$\frac{\text{Anzahl erreichbarer Zustände}}{\text{Anzahl möglicher Zustände}} \leq \frac{1}{2^{2^n-2}}$$

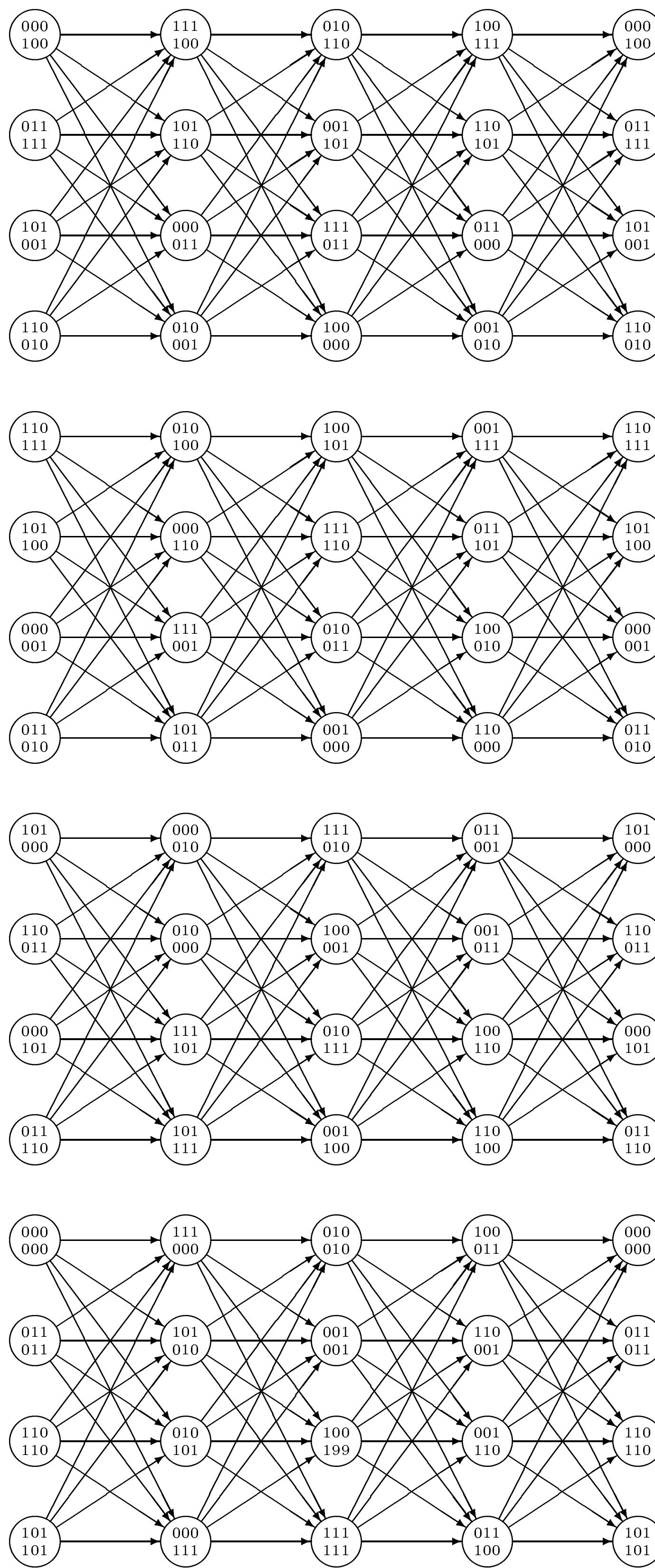



Abbildung 18: Endlicher Automat des $\mathrm{SINC}_2$

### Parität des Nachfolgezustands

Betrachtet man die Gesamtparität einer Schlüsselmatrix, so läßt sich auch hier eine Aussage machen. Es sei:

$$\mathrm{Par}(K) := \bigoplus_{s=1}^{2n-1} \bigoplus_{z=0}^{2^{n-1}-1} k(z,s)$$

Bei einer Verschlüsselung wird eine ungerade Anzahl von Tauschern, nämlich genau $2n-1$, invertiert. Dadurch ändert sich die Gesamtparität und es gilt:

**Satz 5.12:** $\mathrm{Par}(K_t) = 1 - Par(K_{t+1})$

Auf diese Art läßt sich die Zustandsmenge $\mathcal{K}$ in gerade und ungerade Zustände zerlegen. Dadurch, daß sich gerade und ungerade Zustände abwechseln, haben Zyklen einer Zustandfolge immer gerade Länge. Dies wird auch in Abbildung 18 deutlich. Mögliche Zyklenlängen sind nur Vielfache von 4.

### Zyklen

Die Darstellung des endlichen Automaten (Abbildung 18) zeigt, daß beim $\mathrm{SINC}_2$ Zustandszyklen der Länge 4 auftreten können. Das bedeutet, daß es Folgen der Länge 4 gibt, die bei der Verschlüsselung einen Schlüsselzustand in sich selbst überführen. Bei größeren Netzwerken ($n \geq 2$) gibt es Zyklen der Länge $2^n$. Dieser Zyklus ergibt sich bei Verschlüsselung einer konstanten Folge. Dabei werden die Signale abwechselnd durch das obere und das untere Netzwerk verschlüsselt. In den beiden Unternetzwerken wird wieder eine konstante Folge verschlüsselt. Die Zyklenlänge im ganzen Netzwerk ist dadurch doppelt so lang wie in den Unternetzwerken. Es läßt sich induktiv beweisen, daß ein $\mathrm{SINC}_n$ bei Verschlüsselung einer konstanten Folge immer einen Zustandszyklus der Länge $2^n$ durchläuft. Der Beweis erfordert jedoch noch weitere Eigenschaften der erzeugten Chiffratfolge. Für $n \geq 3$ existieren auch Zyklen der Länge $2^{n-1}$. Diese erhält man durch Verschlüsseln der Folge $2k, 2k+1, 2k, 2k+1, \ldots$. Es ist nicht auszuschließen, daß es noch kürzere Zyklen gibt.

**Satz 5.13:** Beim $\mathrm{SINC}_n$ mit einfacher Pfadmodifikation existieren kurze Zyklen.

Die Existenz dieser kurzen Zyklen schließt sowohl die Anwendung des Verfahrens zur Verschlüsselung als auch die Hashfunktion aus. Bei der Verschlüsselung geht der Vorteil des Auto-Key-Systems verloren, dadurch daß der Angreifer mehrfach Chiffrate erhält, die mit demselben Schlüssel chiffriert wurden. Bei der Anwendung als Hashfunktion besteht die Möglichkeit, eine der genannten Folgen in der entsprechenden Zyklenlänge in das Dokument einzufügen, ohne daß sich der Hashwert ändert.

### Äquivalenz von Schlüsseln

Eine weitere wichtige Frage bei der Beurteilung eines Kryptosystems betrifft die *signifikante Schlüssellänge*. Gibt es *äquivalente Schlüssel*? Zwei Schlüssel $K$ und $K'$ sind äquivalent gdw.

$$E_K(M) = E_{K'}(M).$$

Man könnte vermuten, daß ein $\text{SINC}_n$ dieselben äquivalenten Schlüssel wie das entsprechende Beneš-Netzwerks $B_n$ ohne Selbstmodifikation besitzt. Dies ist jedoch nicht der Fall, wie das folgende Beispiel beweist. Die beiden folgenden Netzwerke erzeugen dieselbe Permutation:

Abbildung 19: Äquivalente Belegungen eines $B_2$

Ich betrachte das $\text{SINC}_2$ mit den Schlüsseln:

$$K^1 := \begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix}, \quad K^2 := \begin{pmatrix} 1 & 0 & 1 \\ 0 & 0 & 0 \end{pmatrix}$$

Bei der Verschlüsselung des Textes '0,1' mit dem entsprechenden $\text{SINC}_2$ ergeben sich jedoch verschiedene Chiffrate:

$$E_{K^1}(0,1) = 0,2 \neq 0,3 = E_{K^2}(0,1)$$

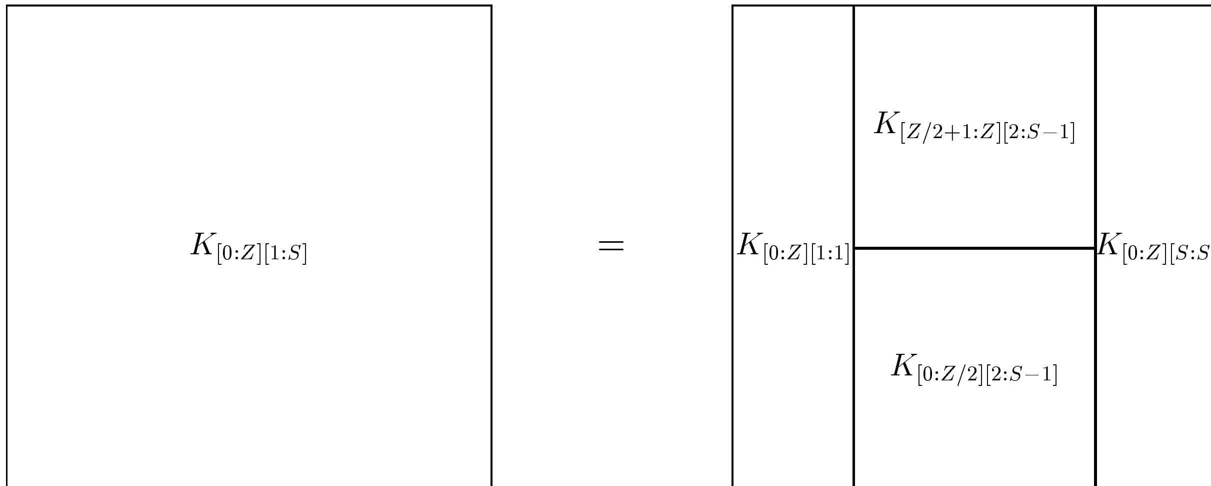



Abbildung 20: Die Zerlegung der Schlüsselmatrix

Es gibt jedoch auch beim $\text{SINC}_n$ äquivalente Schlüssel. Um dies zu zeigen, definiere ich eine Schreibweise für Untermatrizen einer Matrix $K$.

$K$ ist eine Matrix der Form $((\{k\}))_{[0,Z][1,S]}$. Dann sei $K_{[z1,z2][s1,s2]}$ mit $0 \leq z1 \leq z2 \leq Z$, $1 \leq s1 \leq s2 \leq S$ die Matrix, die sich aus dem Schnitt der Zeilen $z1, \ldots z2$ mit den Spalten $s1, \ldots s2$ ergibt. Die Aufteilung des Beneš-Netzwerks in linke Spalte, rechte Spalte und die beiden Untermatrizen läßt sich darstellen mit $Z = 2^{n-1} - 1,\ S = 2n - 1$ (siehe Abb. 20).

Mit Hilfe dieser Schreibweise läßt sich die Klasse der äquivalenten Schlüssel eines $\text{SINC}_n$ mit einfacher Pfadmodifikation beschreiben.

In Worten: Zwei Schlüsselmatrizen $K$ und $K'$ eines $\text{SINC}_n$ sind äquivalent, wenn sie sich durch Invertierung der ersten und letzten Spalte und Vertauschung der beiden Untermatrizen ineinander überführen lassen. Außerdem bleiben sie äquivalent, wenn diese Operation auf ein Unternetzwerk angewendet wird. Mathematisch formuliert lautet dieser Satz:

**Satz 5.14:** Zwei Schlüssel $K$ und $K'$ eines $\text{SINC}_n$ mit einfacher Pfadmodifikation sind äquivalent ($K \sim K'$), wenn gilt:

$$K = K' \quad \text{für } n = 1$$

$$\left.\begin{array}{l} K_{[0:Z][1:1]} = K'_{[0:Z][1:1]} \ \wedge\ K_{[0:Z][S:S]} = K'_{[0:Z][S:S]} \\ \wedge\ K_{[0:Z/2][2:S-1]} \sim K'_{[0:Z/2][2:S-1]} \ \wedge\ K_{[Z/2+1:Z][2:S-1]} \sim K'_{[Z/2+1:Z][2:S-1]} \\ \text{oder} \\ K_{[0:Z][1:1]} = \overline{K'_{[0:Z][1:1]}} \ \wedge\ K_{[0:Z][S:S]} = \overline{K'_{[0:Z][S:S]}} \\ \wedge\ K_{[0:Z/2][2:S-1]} \sim K'_{[Z/2+1:Z][2:S-1]} \ \wedge\ K_{[Z/2+1:Z][2:S-1]} \sim K'_{[0:Z/2][2:S-1]} \end{array}\right\} \text{für } n > 1$$

**Beispiel 5.4:** Folgende Schlüsselmatrizen eines $\text{SINC}_3$ sind äquivalent:

$$\begin{pmatrix} 0 & 0 & 1 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 1 & 0 \end{pmatrix} \sim \begin{pmatrix} 1 & 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 & 1 \\ 0 & 1 & 0 & 0 & 1 \end{pmatrix} \sim \begin{pmatrix} 1 & 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 & 1 \\ 0 & 0 & 1 & 1 & 1 \end{pmatrix} \sim \begin{pmatrix} 1 & 1 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 & 1 \\ 1 & 1 & 0 & 0 & 1 \\ 0 & 0 & 1 & 1 & 1 \end{pmatrix}$$

Ein mathematischer Beweis für diesen Satz erfordert einen hohen Aufwand für Notation und Definition der Operationen.

Plausibel läßt sich dieser Sachverhalt durch das "Gummibandmodell" machen. Vertauscht man die beiden Untermatrizen, so ist dies äquivalent zur Vertauschung der Ausgänge der Tauscher der linken Spalte, sowie der Eingänge der Tauscher auf der rechten Seite. Durch Invertierung dieser Spalten wird dies rückgängig gemacht (siehe Abb.7). Die Selbstmodifikation hält sich an dieselbe rekursive Struktur und erhält dadurch die Äquivalenz.

### 5.2.2 Brechen des Schlüssels

Aufgrund der genannten Regelmäßigkeiten ist es möglich einen Angriff gegen dieses Kryptosystem zu unternehmen. Es ist davon auszugehen, daß ein potentieller Angreifer eine mit

$\mathrm{SINC}_n$ verschlüsselte Kommunikation abhört und einen Teil des Klartextes bereits kennt. Ohne Berücksichtigung der Allgemeinheit sei dies ein zusammenhängendes Stück am Anfang des Dokuments. Es ist jedoch mit den erwähnten Invarianten möglich, das Verfahren auf unzusammenhängende Klartext-, Schlüsseltextfragmente zu übertragen. Ziel ist es, aus dem *known plaintext* einen Schlüssel $N$, der zum Schlüssel $K$ äquivalent ist, in effizienter Zeit zu berechnen.

Gegeben sei eine Folge von $l$ Klartext- Schlüsseltextpaaren. Ich bezeichne sie mit $MC = (m_0, c_0), (m_1, c_1), \ldots, (m_{l-1}, c_{l-1})$. Gesucht ist ein Schlüssel $N$, so daß gilt

$$E_N(m_0, \ldots, m_{l-1}) = c_0, \ldots, c_{l-1}.$$

Eine Möglichkeit einen solchen Schlüssel $N$ zu finden ist die vollständige Suche. Diese Suche ist bei jedem Kryptosystem möglich, benötigt jedoch einen Aufwand von $O(2^{\text{Schlüssellänge}})$. Beim $\mathrm{SINC}_8$, der eine Schlüssellänge von 1920 Bit hat, scheitert diese Möglichkeit am Zeit- bzw. Rechenaufwand, der alle in der Informatik realisierbaren Größenordnungen um ein Vielfaches übersteigt.

Eine andere Möglichkeit besteht durch sogenanntes *Backtracking*. Man betrachtet alle möglichen Pfade des Netzwerks, die den Klartext $m_0$ in $c_0$ überführen. Dann betrachtet man alle Pfade für $(m_1, c_1)$ unter der Maßgabe, der Tauscherstellung des Pfades $(m_0, c_0)$. Die Fortsetzung dieses Verfahrens erweist sich jedoch als ebenso aufwendig wie die vollständige Schlüsselsuche.

Es muß somit eine Methode zur Einschränkung der Suche gefunden werden. Die "Schwachstelle" des SINC ist dabei die in 5.2.1 beschriebene Äquivalenz von Schlüsseln.

Bevor ich die Attacke formal beschreiben werde, versuche ich die zugrundeliegende Idee zu erklären.

### Die Idee

Die Äquivalenzklasse eines Schlüssel $K$ enthält immer einen Schlüssel $N$ (normiert), dessen linker oberer Tauscher $n_0(0, 1)$ die Stellung "0" hat. Ich kann also diesen Tauscher mit "0" belegen. Dieser Tauscher dient als "fester Punkt" beim Brechen des Schlüssels. Nun betrachte ich die bekannte Klartext- Schlüsseltextfolge $(m_0, c_0), (m_1, c_1), \ldots, (m_{l-1}, c_{l-1})$. Das erste $m_i \in \{0, 1\}$ ist das erste Signal, das den Tauscher $n(0, 1)$ durchlaufen hat. Existiert kein solches $m_i$, so ist der linke obere Tauscher für die Verschlüsselung nicht relevant und es kann o.B.d.A. ein anderer Tauscher mit "0" belegt werden. Da der erste Tauscher nach Voraussetzung mit "0" belegt war, weiß man, welches der Unternetzwerke durchlaufen wurde. Damit kann man anhand des Outputs $c_i$ die Stellung des betreffenden Tauschers in der rechten Spalte $n_i(2n - 1, c_i \text{ div } 2)$ feststellen. Anhand der $c_k \in [0 : i - 1]$ kann man abzählen, wie oft der Tauscher $(2n - 1, c_i \text{ div } 2)$ vor dem Zeitpunkt $i$ invertiert wurde. Diese Anzahl ist $|\{i \in [0 : i - 1] : c_i \text{ div } 2 = c_k \text{ div } 2\}|$ Es folgt daher für die Initialbelegung dieses Tauschers:

$$n_0(2n - 1, c_i \text{ div } 2) = n_i(2n - 1, c_i \text{ div } 2) \oplus |\{i \in [0 : i - 1] : c_i \text{ div } 2 = c_k \text{ div } 2\}|$$

Dies ist der zweite "feste Punkt" des gesuchten Schlüssels $N$. Da die SINC-Vernetzung symmetrisch ist, läßt dieses Verfahren sowohl den Schluß von der linken auf die rechte Spalte, als auch

den in umgekehrter Richtung zu. Enthält die Folge $MC = (m_0, c_0), (m_1, c_1), \ldots, (m_{l-1}, c_{l-1})$ "genügend" verschiedene Werte, so ist es möglich, die komplette Belegung der linken und rechten Spalte von $N$ zu berechnen. Ist dies gelungen, kann die Folge $MC$ aufgespalten werden in zwei Teilfolgen $MC_0$ und $MC_1$. $MC_0$ ist die Teilfolge, deren Signal das obere Unternetzwerk durchlaufen hat, und $MC_1$ das untere. Mit diesen Teilfolgen kann der Brechungsalgorithmus rekursiv auf die Untermatrizen $N_{[0:Z/2][2:S-1]}$ und $N_{[Z/2+1:Z][2:S-1]}$ angewendet werden.

Eine Beschreibung dieses Verfahrens in Pascal-ähnlicher Notation ist auf Seite 40 nachzulesen.

### Beispiel für eine Attacke

Zur Illustration der Attacke verwende ich hier ein Beispiel eines $\mathrm{SINC}_3$. Die Initialbelegung ist:

$$K = K_0 = \begin{pmatrix} 1 & 0 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 \end{pmatrix}$$

Als Folge $MC_0$ ergibt sich:

| | $MC_0$ | | | | | | | | | | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $i$: | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 | 25 | 26 |
| $m_i$: | 2 | 3 | 7 | 0 | 0 | 1 | 1 | 4 | 1 | 6 | 3 | 5 | 7 | 1 | 4 | 5 | 5 | 0 | 7 | 1 | 2 | 3 | 3 | 6 | 1 | 2 | 0 |
| $c_i$: | 5 | 0 | 7 | 6 | 2 | 3 | 7 | 4 | 7 | 1 | 2 | 7 | 2 | 3 | 5 | 0 | 3 | 6 | 6 | 3 | 5 | 4 | 6 | 6 | 5 | 2 | 0 |

1. Initialisierung der gesuchten Matrix $N$:

$$N := \begin{pmatrix} 0 & - & - & - & - \\ - & - & - & - & - \\ - & - & - & - & - \\ - & - & - & - & - \end{pmatrix}$$

2. Suche nach einem $(m_i, c_i)$, das einen schon initialisierten Tauscher durchläuft. Für $i = 3$ erhält man das Paar (0,6). Da der Tauscher (0,1) bei der Initialisierung nach Annahme auf "0" stand und bis $i = 3$ nicht invertiert wurde, durchläuft das Signal das obere Unternetzwerk. Es folgt daher

$$n_3(5, 3) = 0.$$

Dieser Tauscher wurde seit Beginn der Verschlüsselung 1 mal invertiert, denn nur für $i = 2$ gilt $c_i \in \{6, 7\}$:

$$|\{i \in [0:2] : c_i \text{ div } 2 = 3\}| = |\{2\}| = 1$$

Die Initialbelegung des Tauschers war also

$$n(5, 3) = n_0(5, 3) = n_3(5, 3) \oplus 1 = 1$$

**input:** $MC := (m_0, c_0), (m_1, c_1), (m_2, c_2) \dots (m_{l-1}, c_{l-1}) \in [0 : 2^{n-1} - 1] \times [0 : 2^{n-1} - 1]$
**output:** $N : E_N(m_0, m_1, m_2, \dots m_{l-1}) = c_0, c_1, c_2, \dots c_{l-1}$

**function** *Attacke (n: integer, MC :* $(m_0, c_0), (m_1, c_1), \dots$ *) : Matrix*
**var**
  $N_l, N_r$ *: Vektor der Dimension* $2^{n-1}$*;*
  $in, out$ *: integer;*
**begin**
  **if** $n = 1$ **then**
    $N := (m_0 + c_0) \mod 2$ *(trivial)*
  **else**
    **begin**
    *Initialisiere* $N_l$ *mit* $N_r$ *mit "unbekannt";*
    *Initialisiere* $N_l[1]$ *mit 0;*
    **while** $N_l, N_r$ *nicht vollständig bekannt* **do**
    **begin**
      *Suche ein* $(m_i, c_i)$ *mit*
        *(a)* $N_l[m_i \ div\ 2] \in \{0, 1\} \quad \wedge \quad N_r[c_i \ div\ 2] = unbekannt$ *, oder*
        *(b)* $N_l[m_i \ div\ 2] = unbekannt \quad \wedge \quad N_r[c_i \ div\ 2] \in \{0, 1\}$
      **if** *(a)* **then**
      **begin**
        $in := |\{t | m_t \ div\ 2 = m_i \ div\ 2, 0 \leq t < i\}|;$
        { *in gibt an, wie oft* $Tauscher_{m_i \ div\ 2,1}$ *seit Beginn der Verschlüsselung gekippt wurde* }
        $out := |\{t | s_t \ div\ 2 = s_i \ div\ 2, 0 \leq t < i\}|;$
        { *out gibt an, wie oft* $Tauscher_{c_i \ div\ 2,2n-1}$ *seit Beginn der Verschlüsselung gekippt wurde* }
        $N_r[c_i \ div\ 2] := (out + c_i + in + m_i + N_l[m_i \ div\ 2]) \mod 2;$
        { $(in + m_i + N[m_i \ div\ 2]) \mod 2$ *gibt an ob das Signal durch das obere, Teilnetzwerk* $(N')$ *oder das untere* $(N'')$ *verschlüsselt wurde.* }
      **end;**
      **if** *(b)* **then** *verfahre analog zu (a);*
    **end;**
    *Generiere die Klartext/Schlüsseltextfolgen* $MC'$ *und* $MC''$ *für die beiden Untermatrizen* $N'$ *und* $N''$
    $N' :=$ *Attacke(n-1,MC');*
    $N'' :=$ *Attacke(n-1,MC");*
    $N := \begin{array}{|c|c|c|} \hline & N'' & \\ N_l & \hline & N_r \\ & N' & \\ \hline \end{array}$
    **end; return** $N$ *;*
**end;**

Algorithmus zur Brechung des SINC mit einfacher Pfadinvertierung

und damit gilt

$$N := \begin{pmatrix} 0 & - & - & - & - \\ - & - & - & - & - \\ - & - & - & - & - \\ - & - & - & - & 1 \end{pmatrix}$$

$2'$. Setzt man dieses Verfahren fort, so erhält man die linke und rechte Spalte der Matrix $N$. In Klammern steht jeweils die Reihenfolge, in der die Initialisierung stattfindet und der Index $i$ des ausgewerteten Paares $(m_i, c_i)$.

$$N := \begin{pmatrix} 0\ (-) & - & - & - & 0\ (4.9) \\ 0\ (5.1) & - & - & - & 1\ (3.4) \\ 0\ (7.7) & - & - & - & 1\ (6.0) \\ 1\ (2.2) & - & - & - & 1\ (1.3) \end{pmatrix}$$

Aus der Kenntnis dieser Belegung läßt sich feststellen, durch welches Unternetzwerk das jeweilige Textpaar verschlüsselt wurde. $s = 0$ ist das obere Netzwerk, $s = 1$ das untere.

| | | | | | | | | | | | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $m_i$ | 2 | 3 | 7 | 0 | 0 | 1 | 1 | 4 | 1 | 6 | 3 | 5 | 7 | 1 | 4 | 5 | 5 | 0 | 7 | 1 | 2 | 3 | 3 | 6 | 1 | 2 | 0 |
| $c_i$ | 5 | 0 | 7 | 6 | 2 | 3 | 7 | 4 | 7 | 1 | 2 | 7 | 2 | 3 | 5 | 0 | 3 | 6 | 6 | 3 | 5 | 4 | 6 | 6 | 5 | 2 | 0 |
| $s$ | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 1 | 1 | 0 | 1 | 1 | 0 | 1 |

Die Teilfolgen $MC_{00}$ und $MC_{01}$ ergeben sich durch die Zerlegung von $MC_0$ nach $s$. Die Paare $(m_i, c_i)$ dieser Folgen ergeben sich durch Ganzzahldivision mit 2.

| | $MC_{00}$: | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $i$: | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
| $m_i$: | 1 | 1 | 3 | 0 | 0 | 2 | 3 | 2 | 3 | 0 | 2 | 2 | 0 | 0 | 1 | 1 |
| $c_i$: | 2 | 0 | 3 | 3 | 3 | 2 | 0 | 3 | 1 | 1 | 2 | 0 | 3 | 1 | 3 | 1 |

| | $MC_{01}$: | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| $i$: | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
| $m_i$: | 0 | 0 | 0 | 1 | 2 | 3 | 1 | 1 | 3 | 0 | 0 |
| $c_i$: | 1 | 1 | 3 | 1 | 1 | 3 | 2 | 2 | 3 | 2 | 0 |

Führt man das gezeigte Verfahren nun mit diesen Folgen aus, so erhält man die linke und rechte Spalte der Untermatrizen:

$$N_{[0,1][2,4]} := \begin{pmatrix} 0 & - & 0 \\ 0 & - & 1 \end{pmatrix} N_{[2,3][2,4]} := \begin{pmatrix} 0 & - & 1 \\ 1 & - & 1 \end{pmatrix}$$

Das Verfahren wird rekursiv fortgesetzt und es ergeben sich für für die Teilfolgen der Teilfolgen die Werte:

| | $MC_{000}$: | | | | | |
|---|---|---|---|---|---|---|
| $i$: | 0 | 1 | 2 | 3 | 4 | 5 |
| $m_i$: | 0 | 0 | 0 | 1 | 0 | 0 |
| $c_i$: | 0 | 1 | 0 | 0 | 0 | 1 |

| | $MC_{001}$: | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| $i$: | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| $m_i$: | 0 | 1 | 0 | 1 | 1 | 1 | 1 | 1 | 0 | 0 |
| $c_i$: | 1 | 1 | 1 | 1 | 0 | 1 | 0 | 1 | 1 | 0 |

| | $MC_{010}$: | | | | | |
|---|---|---|---|---|---|---|
| $i$: | 0 | 1 | 2 | 3 | 4 | 5 |
| $m_i$: | 0 | 0 | 0 | 0 | 1 | 0 |
| $c_i$: | 0 | 1 | 0 | 1 | 1 | 1 |

| | $MC_{011}$: | | | | |
|---|---|---|---|---|---|
| $i$: | 0 | 1 | 2 | 3 | 4 |
| $m_i$: | 0 | 1 | 1 | 0 | 0 |
| $c_i$: | 0 | 0 | 1 | 1 | 0 |

Damit erhält man für die Untermatrizen die Werte:

$$N_{[0,0][3,3]} = (0)$$
$$N_{[1,1][3,3]} = (1)$$
$$N_{[2,2][3,3]} = (0)$$
$$N_{[3,3][3,3]} = (0)$$

Fügt man die berechneten Matrizen zusammen, erhält man die Matrix:

$$N := \begin{pmatrix} 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 1 & 1 & 0 & 1 & 0 \end{pmatrix}$$

Nach Satz 5.2.1 gilt:

$$N = \begin{pmatrix} 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 1 & 1 & 0 & 1 & 0 \end{pmatrix} \sim \begin{pmatrix} 1 & 0 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 \end{pmatrix} = K$$

Damit ist der Schlüssel der Chiffre gebrochen.

**Anmerkung:** Treten in der Klartext- und Schlüsseltextfolge bestimmte Zeichen nicht auf oder ist das Chiffrat kurz, so kann es passieren, daß es Tauscher gibt, die nie von einem Pfad berührt werden. Die Attacke kann dann durchgeführt werden ohne Berücksichtigung dieser Tauscher. Ist ein Großteil der Tauscherstellungen bekannt, so kann die Attacke eventuell mit bekanntem Schlüsseltext weitergeführt werden. Es können dabei einzelne Tauscherstellungen geraten werden. Enthält der Klartext Redundanz, so werden falsche Tauscherstellungen meist zu einem sinnlosen Klartext führen.

### 5.2.3 Die Hashfunktion

Wie beschrieben besteht die Möglichkeit, ein SINC als Hashfunktion zu verwenden. Das Dokument wird mittels eines selbstmodifizierenden Verbindungsnetzwerks mit öffentlichem Initialschlüssel $K$ "verschlüsselt" und der Endzustand $K_l$ wird als Hashwert verwendet.

$$H(M) = H_K(M) = K_l(M)$$

Eine eventuelle Kontraktion des Hashwertes durch eine Funktion $f$ kann erfolgen, soll jedoch vorerst nicht berücksichtigt werden. Ist die Hashfunktion $H(M)$ nicht kollisionsresistent, so ist die Funktion $f(H(M))$ ebenfalls nicht kollisionsresistent.

Eine Funktion ist kollisionsresistent, wenn es nicht gelingt, zwei verschiedene Dokumente $M$ und $M'$ zu finden, die, mit gleichem Initialschlüssel "verschlüsselt", denselben Endzustand $K_l(M)$, bzw. $K_{l'}(M')$ erzeugen.

Die einfache Pfadmodifikation (EPM) ist nicht kollisionsresistent. Es werden genau die durchlaufenen Tauscher invertiert. Dies sind bei großen Netzwerken verhältnismäßig wenige. Es besteht die Möglichkeit, daß im Dokument $M$ zwei aufeinanderfolgende Zeichen (in allen Tauschern) disjunkte Pfade durchlaufen. Diese beiden Zeichen lassen sich vertauschen, ohne daß sich der Folgezustand ändert.

**Beispiel 5.5:** $\mathrm{SINC}_3$

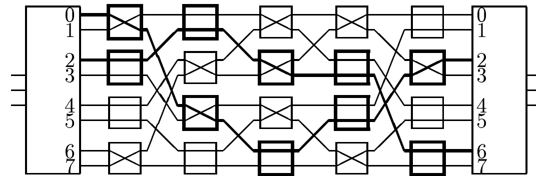



Abbildung 21: Zwei Pfade mit disjunkten Tauschern

Ein Beneš-Netzwerk $B_n$ hat genau $2^n$ Eingänge und Pfade. Damit folgt für die gesuchte Wahrscheinlichkeit:

**Satz 5.15:** Für die Wahrscheinlichkeit, daß die Pfade zweier verschiedener Eingänge eines Beneš-Netzwerks sich in keinem Tauscher berühren gilt:

$$p_n \geq 1 - \frac{2n-1}{2^n-1}$$

Speziell für das diskutierte Netzwerk $B_8$ gilt damit

$$p_8 \geq 1 - \frac{15}{255} \approx 94,1\%$$

Diese Rechnung führt unmittelbar zu dem Schluß, daß die Invertierung von nur $2n-1$ Tauschern als Selbstmodifikation zu wenig ist. Die Selbstmodifikation sollte so beschaffen sein, daß möglichst in jedem Pfad mindestens ein Tauscher invertiert wird.

Ein weitere Eigenschaft, die sich negativ auf die Hashfunktion auswirkt, ist die Existenz kurzer Zyklen dieser Modifikation.

## 5.3 Komplementäre Pfadmodifikation

Um den Nachteil der disjunkten Pfade zu beheben, könnte man statt der Invertierung des Pfades, genau die Tauscher invertieren, die nicht zum Pfad gehören. Die Funktion der Invertierungsmatrix lautet:

$$\forall z \in [0:Z-1], s \in [0:S] : i(z,s) := 1 - \chi_{\mathcal{P}}((z,s))$$

Die Forderung, daß in jedem Pfad mindestens ein Tauscher invertiert wird ist hier fast erfüllt. Der einzige Pfad der nicht betroffen ist, ist der des verschlüsselten Zeichens. Ein gravierender Mangel taucht dabei ebenfalls sofort auf. Verschlüsselt man zweimal hintereinander dasselbe Zeichen, so wird der Ausgangszustand wieder hergestellt. Es ist also möglich, beliebige Doppelzeichen und damit gerade Palindrome in das Dokument einzufügen, ohne daß sich dessen Hashwert ändert. Damit ist diese Selbstmodifikation für eine Hashfunktion ebenfalls ungeeignet.

## 5.4 Modifikation der linken und rechten Pfadnachbarn

Eine andere Art der Selbstmodifikation ist die Invertierung der Nachbarn. Um die beschriebene Attacke gegen die einfache Pfadinvertierung zu verhindern, ist es möglich, statt des Pfades, die linken und rechten Nachbarn des Pfades zu invertieren.

$$\forall z \in [0 : Z-1], s \in [0 : S] : i(z,s) := \chi_{\mathcal{P}}((z,s-1)) \vee \chi_{\mathcal{P}}((z,s+1))$$

Über den Pfad im Inneren des Netzwerks kann nicht so leicht eine Aussage gemacht werden. Durch diese Modifikation wirkt sich der Zustand im Inneren des Netzwerks auf die äußeren Tauscher aus. Dadurch wird die rekursive Struktur zerstört und die beschriebene Attacke verhindert. Im Hinblick auf eine Hashfunktion ist jedoch auch diese Modifikation untauglich, da zum einen der eigene Pfad häufig invariant bleibt, und zum anderen zu wenig Tauscher invertiert werden.

## 5.5 Modifikation aller Pfadnachbarn

Modifiziert man alle 8 Nachbarn der betroffenen Tauscher, so treten ähnliche Probleme auf wie in 5.4. Es werden zu wenig Tauscher invertiert. Es kann eine ähnliche Attacke wie bei der einfachen Pfadmodifikation durchgeführt werden. Die Tauscher, die in der ersten Spalte invertiert werden sind vom Eingangstauscher und vom zweiten Tauscher des Pfades abhängig. Dafür kommen nur zwei in Frage. Es läßt sich daher, ohne Kenntnis des Schlüssels, über fast alle Tauscher der ersten Spalte aussagen, ob sie bei einer Verschlüsselung invertiert werden oder nicht. Dasselbe gilt auch für die letzte Spalte. Daher wird in vielen Fällen eine ähnliche Attacke wie gegen die einfache Pfadmodifikation zum Erfolg führen.

## 5.6 Horizontal/Diagonalmodifikation

Wünschenswert wäre eine Modifikation, bei der etwa die Hälfte aller Tauscher gekippt werden. Dies käme einer Pseudozufallsfolge von Zuständen näher. Zwei zufällig gewählte Matrizen unterscheiden sich durchschnittlich in der Hälfte aller Bits.

Ein Ansatz dazu könnte in der Selbstmodifikation der Horizontalen und Diagonalen liegen. Invertiert werden alle Tauscher, die in einer Zeile oder einer Diagonale liegen, in der mindestens ein Tauscher des Pfades liegt. Die Invertierungsmatrix läßt sich beschreiben durch:

$\forall z \in [0 : Z-1], s \in [0 : S]:$

$$i(z,s) := \begin{cases} 1 & \text{falls } \exists (z',s') \in \mathcal{P} : z = z' \vee z+s \equiv z'+s' \bmod Z \vee z-s \equiv z'-s' \bmod Z \\ 0 & \text{sonst} \end{cases}$$

Veranschaulicht wird diese Invertierung in Abb. 22 (5.6). Wie man in dieser Abbildung sieht, weist die Invertierungsmatrix starke Regelmäßigkeiten dadurch auf, daß durch die rekursive Struktur des Beneš-Netzwerks in den mittleren Spalten eine gewisse Ballung auftritt. Mindestens 13 von 15 Pfadelementen liegen innerhalb einer Hälfte (waagrecht geteilt) des Netzwerks. Mindestens 11 von 15 Tauschern liegen innerhalb eines Viertels, usw. Dadurch werden in der Nähe des mittleren Tauschers $(x,7) \in \mathcal{P}$ mehr Tauscher gekippt als in entfernteren Zeilen. Dieser Makel läßt sich jedoch leicht beheben, indem man statt der Beneš-Vernetzung eine Beneš-äquivalente Vernetzung wählt, bei der die rekursive Struktur aufgelöst wurde. Ein Beispiel für die Invertierungsmatrix bei einer solchen Vernetzung ist Abb. 22 (5.6').

In dieser Modifikation sind die genannten Regelmäßigkeiten der einfachen Pfadmodifikation eliminiert. Durch die Überlappung verschiedener Diagonalen und Horizontalen ist die Anzahl der invertierten Tauscher nicht konstant. Dadurch ist keine Aussage über Paritäten möglich. Ein weiterer Vorteil ist, daß die Invertierungsmatrix nicht nur von der rekursiven Struktur abhängt. Die Tauscher eines Pfades in den mittleren Spalten des Netzwerks, über deren Position eine Aussage nur schwer möglich ist, wirken sich ebenfalls auf die Modifikation der äußeren Tauscher aus. Äquivalente Schlüssel werden dadurch ausgeschlossen.

Über die durchschnittliche Anzahl der invertierten Tauscher läßt sich eine Aussage machen. Betrachtet werden soll der Fall $n = 8$. Ich gehe nicht von einem Pfad eines Beneš-Netzwerks aus, da hier die genannte Ballung in der Mitte des Pfades auftritt. Ich betrachte einen Zufallspfad, der sich ergibt, wenn man eine beliebige Beneš-äquivalente Vernetzung zufällig erzeugt. Es sei der "Schatten" eines Tauschers die Menge aller Tauscher, die auf derselben Horizontale oder einer Diagonale des Tauscher liegen. Dies sind genau 43 Tauscher Die Wahrscheinlichkeit, daß ein zufällig gewählter Tauscher im "Schatten" eines anderen Tauschers liegt ist $\frac{43}{1920}$. Die Wahrscheinlichkeit, daß ein Tauscher in keinem "Schatten" der 15 Tauscher des Pfades liegt, ist somit

$$(1 - \frac{43}{1920})^{15} \approx 0,7119.$$

Damit werden bei dieser Art der Selbstmodifikation ca. 29% der Tauscher invertiert.

Ein kleiner Makel dieser Modifikation ergibt sich, wenn mehrfach aufeinanderfolgend dasselbe Zeichen verschlüsselt wird. Bei den statistischen Betrachtungen (Anhang A) ist dies die '0'. Es ist auffallend, daß auch bei langen Testreihen die Chiffrate '0,28,228' nicht auftreten, die Nachbarn '1,29,229' dafür etwa doppelt so häufig. Dies liegt daran, daß die Ausgangstauscher dieser Chiffrate sowie der Eingangstauscher der '0' bei jeder Verschlüsselung gekippt werden. Daher laufen diese Tauscher synchron. Durch den rekursiven Aufbau des Netzwerks kann nur einer der beiden Ausgänge der entsprechenden Tauscher erreicht werden. Dieser "Fehler" tritt jedoch nicht mehr auf, sobald eine weitere Operation auf die Matrix angewendet wird.

Betrachtet man dieses Verfahren als Hashfunktion, so stellt sich die Frage nach der Vertauschbarkeit zweier benachbarter Zeichen der Klartextfolge. Diese wirkt sich nicht auf den Hashwert aus, wenn durch die Invertierungsfunktion des einen Pfades der andere nicht betroffen wird.

Diese Aussage ist bei der Diagonal/Horizontalmodifikation kommutativ. Die Wahrscheinlichkeit, daß sich eine Vertauschung zweier aufeinanderfolgender Klartextzeichen nicht auf den Hashwert (die Schlüsselmatrix) auswirkt, ist

$$0,7119^{15} \approx \frac{1}{163}$$

Dies ist zu hoch, um kryptographische Sicherheit zu gewährleisten.

## 5.7 Andere Modifikation mit Invertierungsmatrizen

Betrachtet man andere Modifikationen, die nach demselben Schema funktionieren wie die genannten, so kann man verschiedene allgemeine Aussagen treffen.

Ist die Anzahl der invertierten Tauscher zu klein, so besteht die Gefahr der Vertauschbarkeit zweier Zeichen. Ist sie zu groß, so besteht die Gefahr der gegenseitigen Überschattung. Das bedeutet, daß ein Pfad eines Zeichen $m_i$ durch die Invertierung $i+1$ vollständig rückgängig gemacht wird. Dadurch kann eine beliebig lange Sequenz der Form $m_i, m_{i+1}, m_i, m_{i+1}, \ldots, m_i, m_{i+1}$, an der betreffenden Stelle eingefügt werden, ohne daß sich der Schlüsselzustand ändert. Ist die Invertierung kommutativ, wie in Beispiel SM5.6 so ist die Wahrscheinlichkeit selbst im Idealfall 50% zu hoch. Die Wahrscheinlichkeit sowohl für den Fall, daß ein Pfad unverändert bleibt, als auch dafür, daß ein Pfad komplett invertiert wird, liegt in diesem Fall bei

$$0,5^{15} = \frac{1}{32768}$$

Dieser Wert ist für kryptographische Anwendungen zu hoch.

Eine Selbstmodifikation, die sich durch eine pfadabhängige Invertierungsmatrix darstellen läßt, hat den prinzipiellen Nachteil, daß sich die Modifikation einer geraden Anzahl von Durchläufen eines Pfades aufhebt. Ich halte es daher für nötig, weitere Operationen in die Selbstmodifikation einzubringen. Eine elementare Operation, die sich anbietet, ist ein zyklischer Spaltenshift.

## 5.8 Spaltenweise Shiftoperation

In der bereits implementierten VLSI-Version des $\text{SINC}_8$ gibt es die Möglichkeit, die Belegungen der Tauscher innerhalb einer Spalte zu "shiften" (siehe Abb. 23). Diese Operation wird zum Laden des Schlüssels benötigt, kann aber auch zyklisch eingesetzt werden. Es bietet sich daher an, diese Operation auch bei der Selbstmodifikation einzusetzen.

Das Shiften kann entweder unbedingt erfolgen, d.h. nach jeder Verschlüsselung wird modifiziert und geshiftet, oder es kann, abhängig von $K_t$ oder $\mathcal{P}$ ein einfacher oder mehrfacher Shift durchgeführt werden (*Stop & Go*). Durch das Shiften werden die Belegungen der Tauscher eines Pfades auf Tauscher verschoben, die im allgemeinen nicht mehr zu einem gemeinsamen Pfad gehören. Wird ein Pfad mehrfach durchlaufen, so hebt sich die Modifikation nicht auf. Bei der Vernetzung ist dabei darauf zu achten, daß keine "parallelen" Pfade, d.h. Pfade die durch

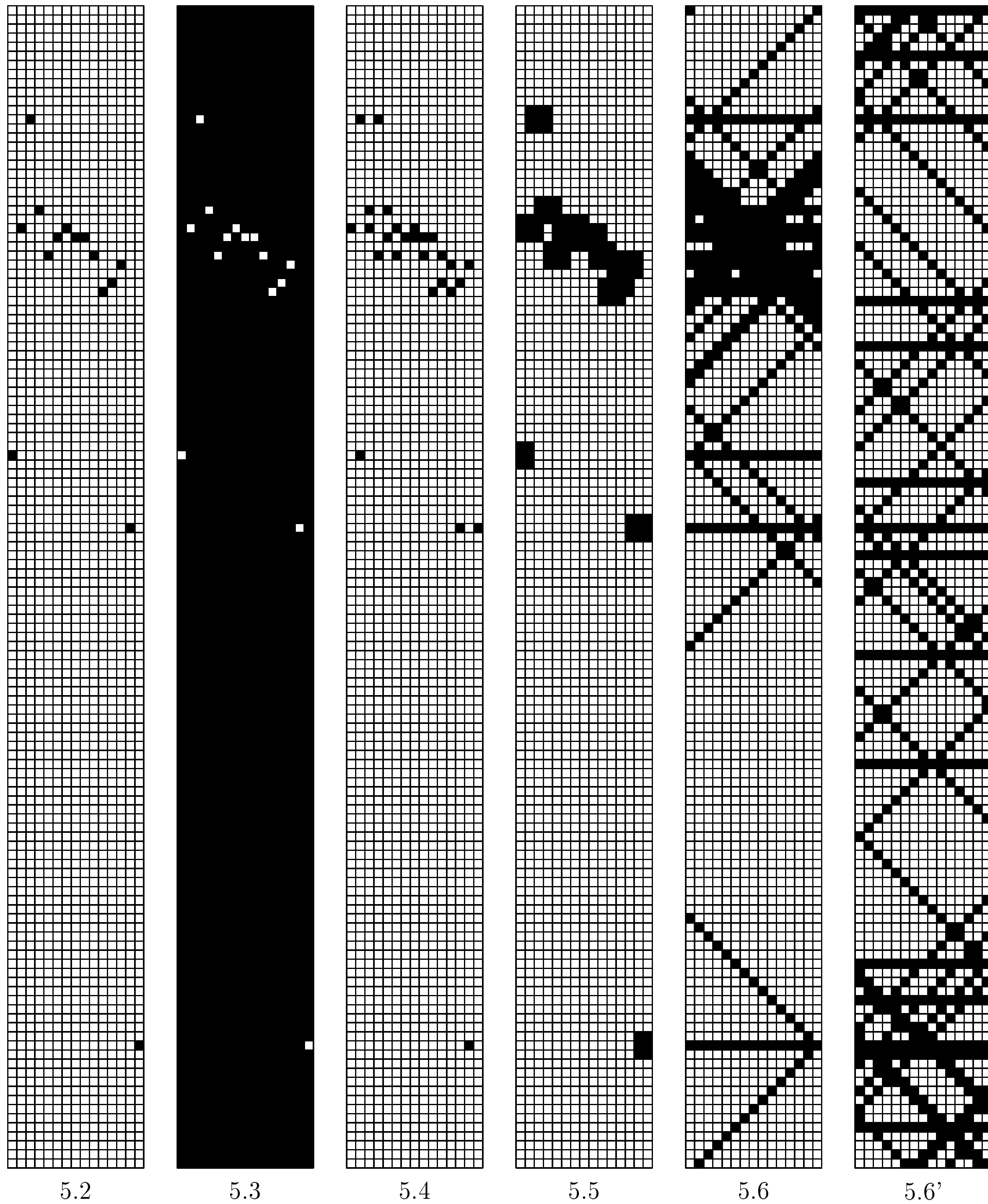



Abbildung 22: Invertierungsmatrizen verschiedener Modifikationen

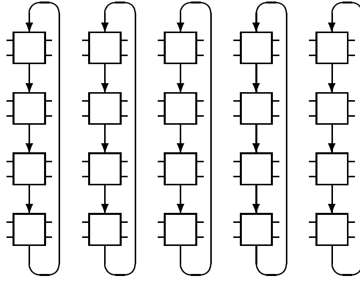


Abbildung 23: Zyklischer Spaltenshift

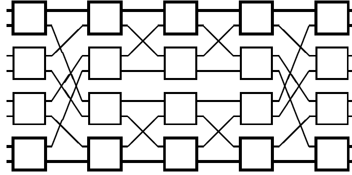


Abbildung 24: Parallele Pfade

einen Spaltenshift ineinander übergehen, auftreten. Bei der Beneš-Vernetzung tritt genau ein Paar paralleler Pfade auf (siehe Abb. 24). Verknüpft man die einfache Pfadmodifikation mit einer unbedingten Shiftoperation, so werden äquivalente Schlüssel eliminiert. Die beschriebene Attacke funktioniert trotzdem noch, wenn man den Algorithmus leicht modifiziert. Um die Tauscherstellung der äußeren Spalten zu ermitteln führt man zwei Attacken parallel durch. Dabei wird der linke obere Tauscher einmal mit "0" und einmal mit "1" initialisiert. Eine dieser Initialisierungen wird im allgemeinen zum Widerspruch führen. Bei den rekursiven Aufrufen muß beachtet werden, daß die Belegungen von einem Netzwerk ins nächste geshiftet werden.

## 5.9 Pseudozufallsgeneratoren

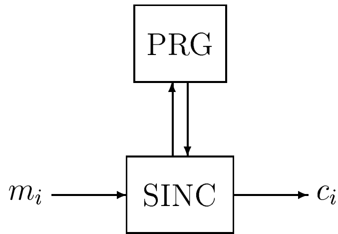
Ein andere Klasse von Zustandsfolgen ist dadurch charakterisiert, daß sie nicht vom Chiffrat $m_t$, bzw. vom Pfad $\mathcal{P}$, abhängig ist, sondern nur vom aktuellen Schlüsselzustand und/oder vom Zeitpunkt $t$. Es handelt es sich dabei um sogenannte Permutationsgeneratoren. Die Folgefunktion ist darstellbar durch

$$K_{t+1} := \delta(K_t, t)$$

Ein Startwert $K_0$ erzeugt eine feste Folge von Schlüsselzuständen $K_t$. Diese kann, abhängig von $\delta$, Eigenschaften einer Zufallsfolge haben. Die entsprechende Chiffre ist eine *sequentielle* Substitutionschiffre. Jeder Schlüssel $K_t$ erzeugt eine Bijektion auf dem Alphabet $A$.

Der Vorteil dieser Funktionen liegt in der Synchronisation und Fehlerkorrektur bei der Über-

mittlung des Chiffrats. Bei einer textabhängigen Funktion $\delta$ führt ein Fehler in der Übermittlung des Chiffrats zur völligen Zerstörung der nachfolgenden Zeichen. Wird die Chiffre durch einen Pseudozufallsgenerator (*Pseudo Random Generator, PRG*) gesteuert, so beschränkt sich der Fehler nur auf das falsch empfangene Zeichen, sofern die Synchronisation von Sender und Empfänger gewährleistet ist.




In den Arbeiten von M. Portz [Port91, Port92] werden Permutationsgeneratoren ausführlicher diskutiert. Er geht dabei von den Eigenschaften eines Pseudozufallsgenerators aus und definiert ähnliche Eigenschaften für pseudozufällige Permutationsgeneratoren. Eine Konstruktion eines pseudozufälligen Permutationsgenerators aus einem Pseudozufallsgenerator erfolgt durch Kombination mit einem Verbindungsnetzwerk. Dabei wird eine binäre Zufallsfolge dazu genutzt, die Tauscherstellungen des Netzwerks festzulegen.

Ich möchte an dieser Stelle noch die Möglichkeit ansprechen, einen Pseudozufallsgenerator implizit in der Tauschermatrix zu realisieren. Dies hat den Vorteil, daß die Operationen der Selbstmodifikation direkt beim Hardware-Entwurf in der Matrix implementiert werden können. Dadurch ist eine einfache und effiziente Realisierung möglich. Ich will versuchen, unter dem Aspekt der Realisierung die möglichen Generatoren einzuschränken.

Die einzige notwendige Forderung an $\delta$ ist die Symmetrie, die die Entschlüsselung mit der gespiegelten Matrix gewährleistet.

## 5.10 Selbstmodifikation "life"

Eine Funktion, die auf beliebigen 0,1-Matrizen operiert, ist den meisten Informatikern unter dem Namen "life" bekannt. Diese Funktion generiert den Folgezustand eines Matrixelementes (graphisch meist als Pixel dargestellt), aus der Belegung der 8 Nachbarn. Die ursprüngliche Version dieser Funktion lautet: [8]

$$k_{t+1}(z,s) = \begin{cases} k_t(z,s) & \text{für sum } = 2 \\ 1 & \text{für sum } = 3 \\ 0 & \text{sonst} \end{cases} \quad \text{mit} \quad \text{sum} := \sum_{max\{|i-z|,|j-s|\}=1} k_t(i,j)$$

Diese Funktion ist als PRG unbrauchbar. Zum einen existieren stabile Zustandsmatrizen, d.h. Zustände, die unverändert bleiben, zum anderen herrscht keine Gleichverteilung von '0' und '1'. Die '0' tritt wesentlich häufiger auf. Statt dieser Funktion kann eine beliebige symmetrische Funktion verwendet werden.

$$k_{t+1}(z,s) = f(K_{[z-1][s-1]})$$

[8] Ich betrachte die Zeilen- und Spaltenindizes immer modulo $Z$ bzw. $S$. Die Nachbarn der Randpunkte liegen dann entsprechend gegenüber.

Auch die spaltenweise Shiftoperation (vgl. 5.8) ist eine solche Funktion. Eine sehr gleichmäßige Verteilung der Matrixelement 0,1 liefert beispielsweise folgende Funktion:

$$k_{t+1}(z,s) = \begin{cases} 0 & \text{für sum } \in \{3,5\} \\ 1 & \text{für sum } \in \{4\} \\ k_t(z,s) & \text{für sum } \in \{1,6,8\} \\ 1-k_t(z,s) & \text{für sum } \in \{0,2,7\} \end{cases} \quad \text{mit} \quad \text{sum} := \sum_{max\{|i-z|,|j-s|\}=1} k_t(i,j)$$

Diese Funktion ist ebenfalls im Simulationsprogramm (siehe Anhang B) enthalten.

Bei dieser Art der Modifikation ist es nicht nötig, daß alle Tauscher durch dieselbe Funktion $f$ modifiziert wird. Um eine stärkere "Unregelmäßigkeit" zu erzeugen, können für die Tauscher verschiedene Funktionen $f_{z,s}$ gewählt werden.

$$k_{t+1}(z,s) = f_{z,s}(K_{[z-1,z+1][s-1,s+1]})$$

Zu beachten ist hier lediglich die Symmetrie der Funktionen. Die Funktionen der Tauscher auf der linken Seite müssen symmetrisch zu denen der rechten Seite sein:

$$f_{z,s}(K_{[z-1,z+1][s-1,s+1]}) := f_{z,s^{-1}}(K_{[z-1,z+1][s-1,s+1]^{-1}})$$

Denkbar ist beispielweise eine Kombination von Funktionen, die von mehreren Nachbarn abhängen, und Shiftoperationen in verschiedene Richtungen (links, rechts, oben, unten, diagonal).

# 6 Attacken

## 6.1 Sequentielle, eingeschränkte Suche

Bei einem praktisch sicheren Kryptosystem sollte das Finden des Schlüssels nur durch vollständige Schlüsselsuche möglich sein. Bei den meisten Blockchiffren führt jede Änderung eines einzelnen Schlüsselbits zu einer völligen Änderung des Chiffrats. Dies wird als *Lawinen* oder *Avalanche-Effekt* bezeichnet. Daher ist es einem (*known-plaintext*) Angreifer meist nicht möglich, auch nur einzelne Schlüsselbits mit höherer Wahrscheinlichkeit als 50% zu erraten. In diesem "alles oder nichts"-Effekt begründet sich auch die Sicherheit einer Blockchiffre. Entweder der Angreifer findet den kompletten Schlüssel oder er hat keine auswertbare Information über die einzelnen Schlüsselbits oder deren Zusammenhänge. Liegen Schlüssellänge und Blockbreite in derselben Größenordnung, genügen meist auch wenige Klartext- Schlüsseltextblöcke, um den kompletten Schlüssel eindeutig festzulegen.

Bei selbstmodifizierenden Netzwerken ist dies nicht der Fall. Beim $\text{SINC}_8$ ist die Verschlüsselung eines einzelnen Zeichens lediglich von 15 Bit abhängig. Die anderen 1905 sind ohne Auswirkung auf diese Verschlüsselung. Durch die große Schlüssellänge wird zwar eine hohe Zahl von *known-plaintext*-Paaren benötigt um den Schlüssel eindeutig festzulegen, dafür besteht die Gefahr, daß Teile des Schlüssels sequentiell bestimmt werden können.

Bei der gezeigten Attacke auf das SINC mit einfacher Pfadmodifikation werden Schlüsselbit einzeln bestimmt. Es stellt sich die Frage, ob sich dies auch auf andere kompliziertere Modifikationen übertragen läßt.

Gegeben sei eine Folge von Klartext-, Schlüsseltextpaaren $(m_0, c_0), (m_1, c_1), \ldots, (m_l, c_l)$.

Die Anzahl der Pfade vom Eingang $m_0$ zum Ausgang $c_0$ ist $2^{15} = 32768$. Nun betrachtet man alle diese Fälle und vergleicht die Pfade des zweiten Paares $m_1, c_1$. Sind alle diese Pfade konsistent bezüglich der Belegungen des ersten Pfades und der Modifikation, so erhöht sich die Anzahl der möglichen Belegungen der beiden Pfade auf $2^{15} \cdot 2^{15} = 1073741824$. Schneiden sich die beiden Pfaden jedoch, so führen einige dieser Belegungen zum Widerspruch und brauchen nicht weiterverfolgt zu werden. Durch dieses "Beschneiden" des entstehenden Baumes wird das exponentielle Wachstum des Aufwandes stark eingeschränkt.

Es ist nun in erster Linie von der Modifikation abhängig, inwieweit Aussagen über die Stellungen und die Modifikationen der anderen Tauscher gemacht werden können. Daher sollte dieses Kriterium bei der Wahl der Modifikation berücksichtigt werden.

## 6.2 Attacke mit mehrfachem Known-Plaintext

Verschlüsselt man verschiedene Texte mit demselben Schlüssel, so besteht die Gefahr, daß ein möglicher Angreifer mehrere Klartext- Schlüsseltextpaare zum Initialschlüssel $K_0$ besitzt.

Gegeben sind mehrere Known-Plaintextfolgen, die mit demselben Schlüssel $K$ verschlüsselt wurden. $(m_0^1, c_0^1), (m_1^1, c_1^1), \ldots, (m_0^2, c_0^2), (m_1^2, c_1^2), \ldots, (m_0^k, c_0^k), (m_1^k, c_1^k), \ldots$.

Ziel des Angreifers ist es, aus dieser Information den Schlüssel $K = K_0$ herzuleiten. Der Angreifer hat Zugriff auf mehrere Elemente der ersten Substitution. Im besten Fall kennt der Angreifer die komplette Permutation, die durch $K_0$ erzeugt wird.

Die Anzahl der Matrixbelegungen, die eine gegebene Permutation erzeugen, liegt zwischen $2^{127}$ und $2^{896}$ und ist durchschnittlich $2^{236}$ (siehe 4.4). Diese Zahl ist natürlich zu hoch, um alle möglichen Schlüsselbelegungen zu testen. Es ist jedoch auch hier möglich, durch geschicktes "Branch and Bound" bei der Konstruktion einer Matrixbelegung bestimmte Belegungen auszuschließen und dadurch den Aufwand auf ein ausführbares Maß zu begrenzen. Die Permutation wird bei Beneš-Netzwerken rekursiv erzeugt. Bei den Rekursionsschritten können bestimmte Möglichkeiten ausgeschlossen werden.

Betrachtet man nicht die gesamte Schlüsselbelegung, sondern nur die erste und letzte Spalte des Schlüssels so ist die Anzahl der möglichen Belegungen verhältnismäßig klein. Sie liegt zwischen 2 und $2^{128}$. Im Durchschnitt gibt es etwa 20 mögliche Belegungen (siehe 4.4). Diese können einzeln getestet werden und eventuell mit den Folgechiffraten $(m_1^1, c_1^1), (m_1^2, c_1^2), \ldots, (m_1^k, c_1^k)$ zum Widerspruch geführt werden. Dies gelingt insbesondere dann, wenn definitive Aussagen über den Folgezustand einzelner Tauscher der äußeren Spalten gemacht werden können. Bei der einfachen Pfadmodifikation ist der Folgezustand jedes Tauschers bekannt. Bei der Modifikation der Diagonalen können beispielsweise Aussagen über die Tauscher gemacht werden, die genau auf der entsprechenden Diagonalen oder Horizontalen liegen. Diese Tauscher werden mit Sicherheit invertiert. Eine spaltenweise Shiftoperation

verhindert das nicht. Durchschnittlich genügen zwei Tauscher um eine falsche Belegung auf die gezeigte Art zum Widerspruch zu bringen. Bei den gezeigten Selbstmodifikationen ist eine Aussage über bestimmte Tauscher durchweg möglich. Um diese Möglichkeit zu verhindern, sollte die Selbstmodifikationsfunktion $\delta$ so beschaffen sein, daß die Stellung jedes Tauschers von möglichst vielen anderen Tauschern abhängt. Da sich das aus technischen Gründen schwer realisieren läßt, muß die Absicherung gegen diese Attacke durch eine entsprechendes Verschlüsselungsprotokoll erfolgen. Es sollten daher nie mehrere Texte mit demselben Schlüssel $K$ verschlüsselt werden.

### 6.3 Chosen Plaintext/Chosen Ciphertext-Attacke

Eine Attacke, die in den meisten Verschlüsselungsprotokollen sicher ausgeschlossen ist, ist eine kombinierte Chosen Plaintext/Chosen Ciphertext-Attacke. Diese wird möglich, wenn der Angreifer sowohl zugriff auf das Verschlüsselungsgerät, als auch auf das Entschlüsselungsgerät hat. Er hat dann die Möglichkeit abwechselnd Klartextzeichen zu verschlüsseln und Schlüsseltextzeichen zu entschlüsseln. Diese Attacke kann beim SINC im einfachen Verschlüsselungsmodus in vielen Fällen zum Erfolg führen.

Der Angreifer “rät” die Stellung eines Tauschers in einer äußeren Spalte und ver- oder entschlüsselt ein Signal, das diesen Tauscher berührt. Dadurch erhält er die Stellung des durchlaufenen Tauschers in der gegenüberliegenden Spalte. Die Selbstmodifikation muß dann so beschaffen sein, daß keine Aussage über weitere Tauscher in den äußeren Spalten möglich ist. Gewinnt der Angreifer Information über mindestens einen weiteren Tauscher, so hat er die Möglichkeit sich von einem “Fixpunkt”, also einem ihm bekannten Tauscher, zu einem anderen zu “hangeln”. Diese Attacke kann jedoch durch die Wahl der Betriebsart verhindert werden.

## 7 Betriebsarten und Erweiterungen

### 7.1 Betriebsarten der Verschlüsselung

Wie gezeigt läßt sich das Netzwerk SINC für verschiedene Zwecke in der Kryptographie einsetzen. Es ist daher sinnvoll, in Abhängigkeit von der Anwendung verschiedene Betriebsarten zu betrachten. Die verschiedenen Betriebsarten unterscheiden sich sich vor allem in der Fehlerfortpflanzung, der Verschlüsselungsrate und der Sicherheit gegenüber bestimmten Angriffen. Die Wahl des Betriebsmodus ist dabei auch noch von der Selbstmodifikation abhängig.

Einfachste Betriebsart ist der direkte Verschlüsselungsmodus. Dieser Betriebsmodus wurde auch bei den bisherigen Betrachtungen über das Netzwerk verwendet. Die Verschlüsselungsoperation ist dabei definiert durch

$$c_t := e_{K_t}(m_t), \quad K_{t+1} := \delta(K_t, m_t)$$

Beim DES ist dieser Modus als *Electronic Codebook Mode (ECB)* normiert. Im Gegensatz

zum DES werden bei einer Auto-Key-Chiffre gleiche Klartextblöcke im allgemeinen nicht durch gleiche Schlüsseltextblöcke ersetzt werden.

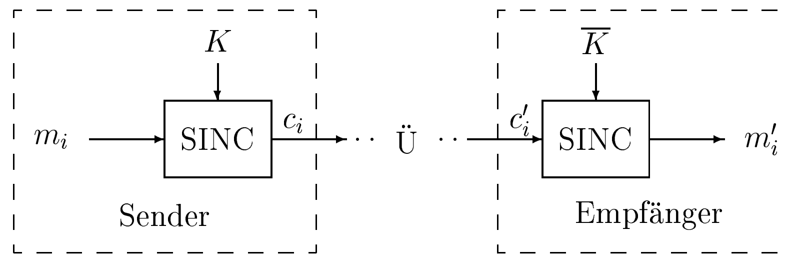



Abbildung 25: Direkter Verschlüsselungsmodus

Der Modus eignet sich für alle Selbstmodifikationen. Ist die Modifikation chiffratabhängig ($K_{t+1} = \delta(K_t, m_t)$), so hat dieser Verschlüsselungsbetrieb den Nachteil, daß ein Fehler in der Übertragung des Chiffrats ($c_i \neq c'_i$) den Schlüssel zerstört und damit die Entschlüsselung aller folgenden Zeichen unmöglich macht. Handelt es sich um ein chiffratunabhängiges $\delta$, so muß lediglich die Synchronisation von Sender und Empfänger gewährleistet sein.

Eine Erweiterung dieses Modus besteht darin, auf Klartext- und Schlüsseltextzeichen einen zusätzlichen Schlüssel $K_{in}$ und $K_{out}$ der jeweiligen Blockbreite zu addieren (siehe [Hors93]). Die Sicherheit des Systems wird dadurch eventuell erhöht. Die Verschlüsselungsfunktion modifiziert sich dann zu

$$c_t := K_{out} \oplus e_{K_t}(m_t \oplus K_{in}), \quad K_{t+1} := \delta(K_t, m_t \oplus K_{in})$$

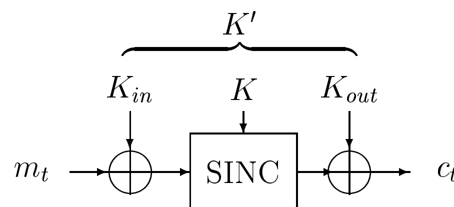



Abbildung 26: Erweiterter Verschlüsselungsmodus

## 7.2 Rückkopplungen

Eine weitere Möglichkeit besteht in einer Rückkopplung des Chiffrats auf den Schlüssel. Besteht ein Schreibzugriff auf bestimmte Tauscher des Netzwerks, kann über ein ROM die Schlüssselmatrix noch zusätzlich beeinflußt werden.

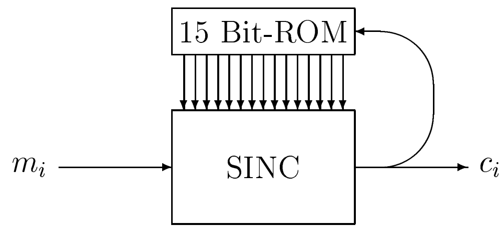



Abbildung 27: Rückkopplung des Chiffrats auf den Schlüssel

Eine andere Möglichkeit ist der Einsatz als Pseudozufallsgenerator *(PRG-Modus)*. Diese Vorschläge finden sich auch in [Hors93]. Bei dieser Art des Betriebs braucht die Bedingung der Symmetrie, der Vernetzung und der Modifikation nicht erfüllt sein. Sender und Empfänger ver-

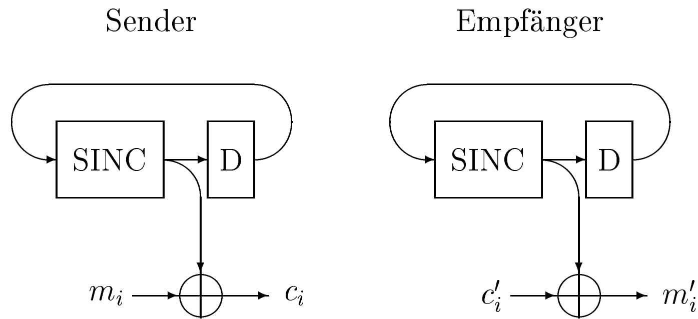



Abbildung 28: SINC als Pseudozufallsgenerator

wenden nur die Verschlüsselungsfunktion des SINC. Diese muß daher nicht notwendigerweise umkehrbar sein. Dadurch eröffnen sich möglicherweise weitere Vernetzungen und Modifikationsfunktionen, die relativ frei gewählt sein können.

Die Verschlüsselungsoperation des Pseudozufallsgenerator (siehe Abbildung 28) ist definiert durch

$$c_t := r_t \oplus m_t, \quad r_{t+1} := e_{K_t}(r_t), \quad K_{t+1} := \delta(K_t)$$

Beginnend mit einem Initalschlüssel $K_0$ und einem Initalinput $(r_0)$ wird das Chiffrat in einem Speicherbaustein D (n parallele Delay-Flip-Flops) einen Takt verzögert, und als Input rückgekoppelt. Die dadurch entstehende Pseudozufallsfolge kann dann als Stromchiffre eingesetzt werden. Bei diesem Modus erzeugt ein falsch übertragenes Chiffratzeichen lediglich ein falsch entschlüsseltes Klartextzeichen. Daher eignet sich dieser Modus gut für fehlerbehaftete Datenübertragung. Ein weiterer Pseudozufallsgenerator ist in Abbildung 29 zu sehen. Es handelt sich dabei um einen *Stop & Go*-Generator. Es werden Vergleichswerte angelegt, die bei Übereinstimmung mit dem Output des Generators einen Zählimpuls geben. Der Zähler bestimmt in Verbindung mit dem Output, welche Adresse des EPROMs ausgelesen wird. Der ausgelesene Wert wird dann mit SINC verschlüsselt. Die minimale Zyklenlänge entspricht der Größe des Zählers. Durch diese Kombination können kurze Zyklen ausgeschlossen werden.

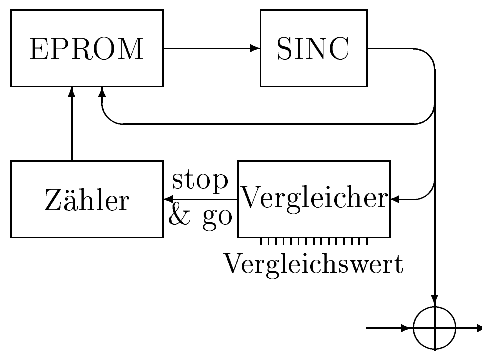



Abbildung 29: Pseudozufallsgenerator

## 7.3 Verkettung mehrerer SINC

Denkbar ist auch eine Kombination mehrerer kleiner Substitutionsboxen (hier: SINC) zu einer größeren. Dadurch läßt sich der Block vergrößern, ohne daß der Aufwand exponentiell ansteigt. Als Basisbaustein könnte sich ein $SINC_4$ eignen. Mit einer Schlüssellänge von 56 Bit entspricht er eher der Größenordnung üblicher Blockchiffren als ein $SINC_8$ mit 1920 Bit.

**Beispiel 7.6:**

Zusammenschaltung von 6 $SINC_4$ zu einer 8 Bit Blockchiffre. Die Schlüssellänge beträgt bei dieser Kombination $6 \cdot 56 = 336$ Bit. Dies ist etwa $1/6$ der Größe des $SINC_8$. Der Signalpfad hat die Länge 21. Das Signal wird zusätzlich dadurch verzögert, daß es jeweils am Eingang und Ausgang des $SINC_4$ codiert und decodiert wird. Dies wird möglicherweise durch die schnellere Modifikation des Netzwerks ausgeglichen. Ein weiterer Vorteil, der Anwendung als Hashfunktion oder Verschlüsselung, besteht in einem stärkeren Lawineneffekt. Von den 336 Tauschern werden 42 durchlaufen. Zum Vergleich: Beim $SINC_8$ sind es 15 von 1920.

Die Gleichverteilung über den Chiffraten bei zufälliger Initialisierung bleibt bei Verkettung erhalten. Die Vollständigkeit des Permutationsnetzwerks bleibt nicht erhalten. Dies ist daran erkennbar, daß die Anzahl möglicher Permutationen größer ist als die Anzahl möglicher Zustände $2^8! > 2^{336}$.

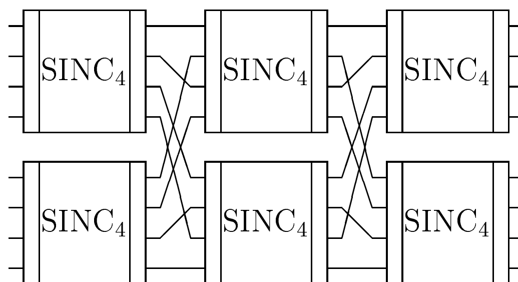



Abbildung 30: Vernetzung mehrerer $SINC_4$

**Beispiel 7.7:**

Zusammenschaltung von 8 $SINC_4$ zu einer 16 Bit Blockchiffre (siehe Abb. 31). Diese Art der Kaskadierung von mehreren SINC ist noch effizienter, da die Vernetzung selbstinvers ist und dadurch zwei Stufen ausreichen. Setzt man dieses Verfahren rekursiv fort, so quadriert sich

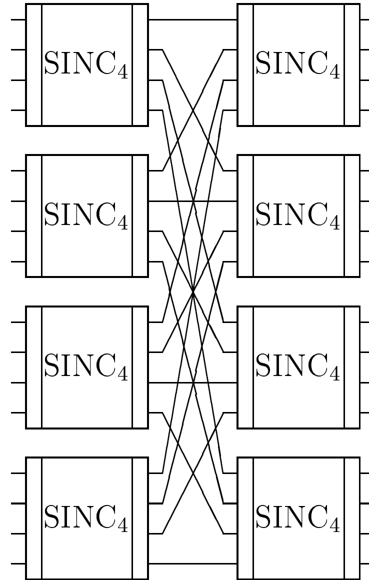



Abbildung 31: Zweistufige Kaskadierung von $SINC_4$

die Anzahl der Eingänge bei jedem Schritt. Die Tiefe des Netzwerks verdoppelt sich dabei nur. Das Wachstum des gesamten Netzwerks liegt in $O(b \log b)$ bezüglich der Blockbreite $b$.

Die Sicherheit bezüglich einer Attacke mit mehrfachem Klartext (siehe 6.2) ist bei dieser Vernetzung nicht gewährleistet. Sind mehrere Elemente der Gesamtpermutation bekannt, so läßt sich die Belegung mit geringem Aufwand berechnen.

**Beispiel 7.8:**

Zusammenschaltung von 64 $SINC_4$ zu einer 64 Bit Blockchiffre (siehe Abb. 7.3). Diese Zusammenschaltung besteht aus 3548 Tauschern. Die Pfade berühren 448 Tauscher, also etwa 13 %. Der Avalancheeffekt eines einzelnen Eingangsbits ist bei dieser Schaltung jedoch nicht erfüllt. Kippt man am Eingang ein einzelnes Bit, so wird dies eine völlige Änderung des Ausgangssignals des betroffenen $SINC_4$ bewirken. Es werden also nach der ersten Spalte etwa 2 Bit verändert, nach der 2. Spalte 4 Bit und nach der 4. Spalte durchschnittlich 16 Bit. Der Avalancheeffekt ändert jedoch durchschnittlich 32 Bit des Chiffrats.

## 7.4 Sicherheit verschiedener Betriebsarten

Prinzipiell ist es möglich, durch die Wahl der Betriebsart die Sicherheit des Kryptosystems zu erhöhen. Es sollte jedoch vermieden werden, eine unsichere Selbstmodifikation durch zusätz-

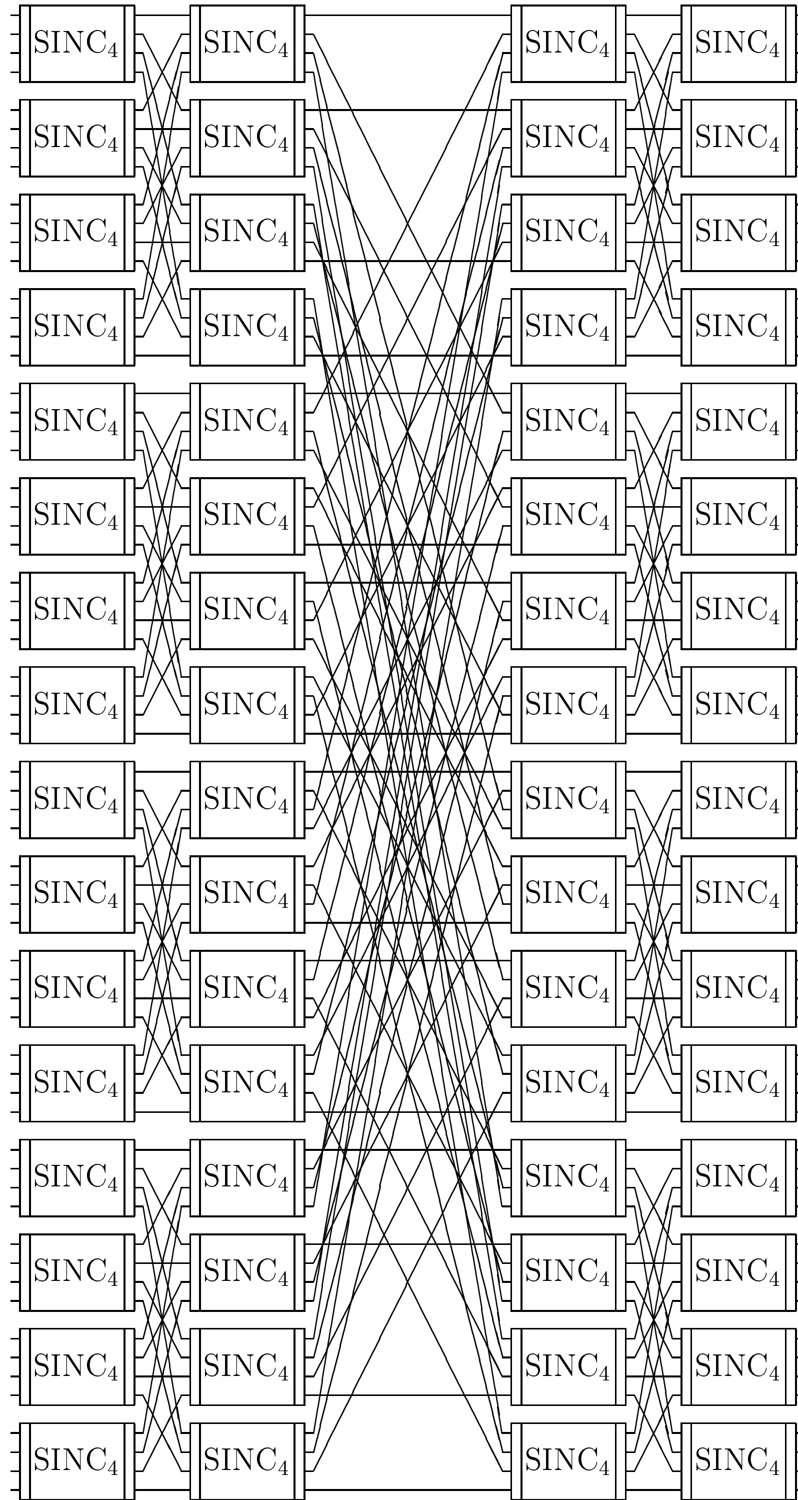



Abbildung 32: 64 Bit Chiffre (3548 Tauscher)

liche Operationen auf den Inputs oder Outputs oder durch die Wahl eines aufwendigen Betriebsmodus "sicherer" zu machen. Eine eventuelle Attacke wird dadurch zwar erschwert, da die Zugriffe auf die Ein- und Ausgangsdaten nicht mehr direkt erfolgen können, es besteht jedoch die Gefahr, daß bestimmte Tricks eine Transformation der Brechungsmethode ermöglichen. Übersieht man diese Möglichkeit, so ist das Kryptosystem nicht sicherer, sondern lediglich undurchsichtiger geworden. Die Betrachtungen über die Sicherheit habe ich aus diesem Grund, stets auf den direkten Verschlüsselungsmodus bezogen.

## 7.5 Betriebsarten der Hashfunktion

Auch hier bietet sich an verschiedene Modi zu betrachten, einen Hashwert $H$ zu berechnen. Abgesehen von den Pseudozufallsgeneratoren eignen sich alle Betriebsarten der Verschlüsselung auch für die Hashwertbildung. Ist die Sicherheit der Modifikation gewährleistet, so kann die Funktion $\delta$ als Einwegfunktion betrachtet werden. Gut geeignet ist auch hier eine zusätzliche Rückkopplung auf den Schlüssel oder eine Vernetzung mehrerer kleinerer SINC.

Das Beispiel aus Abbildung 33 zeigt eine Möglichkeit, eine Hashfunktion durch vernetzte $\text{SINC}_4$ mit Rückkopplung des Chiffrats zu realisieren. Diese Kombination besteht aus 336 Tauschern. Mit Hilfe einer einfachen Kontraktionsfunktion kann der Hashwert auf 128 Bit reduziert werden. Als Kontraktion eignet sich beispielsweise eine Auswahlfunktion oder eine Paritätsfunktion.

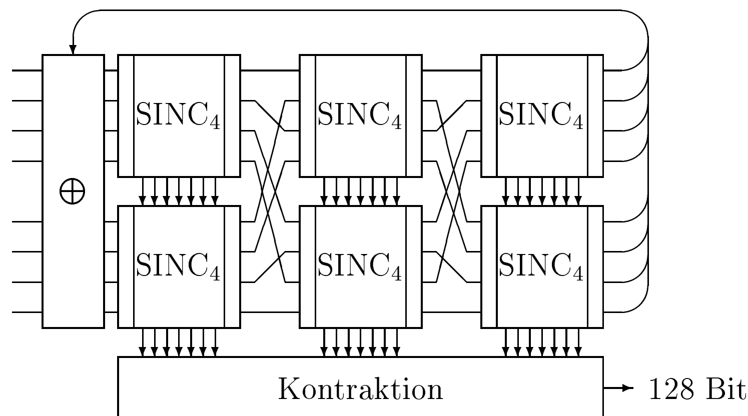



Abbildung 33: Beispiel für eine Hashfunktion

# 8 Weitere Möglichkeiten

## 8.1 Expansion des Schlüssels

Eine Schlüssellänge von 1920, wie beim diskutierten $SINC_8$ ist im Vergleich zu anderen Chiffren sehr lang. Die vollständige Schlüsselsuche ist bereits bei einer Länge von 64 Bit aus Komplexitätsgründen nicht zu realisieren.

Ist die Chiffre sicher, in dem Sinn, daß nur die vollständige Suche des Schlüssels möglich ist, so ist es nicht nötig, einen Schlüssel der Länge 1920 zu verwenden. Es genügt ein kürzerer Schlüssel von z.B. 128 Bit, der entsprechend vervielfältigt wird. Welche Funktionen sich für diese Expansion eignen, hängt sehr stark von der gewählten Selbstmodifikation und der Vernetzung ab.

**Beispiel 8.9:**

Eine einfache Schlüsselexpansion ist die zyklische Wiederholung des Schlüssels. Ein Schlüssel $K'$ beliebiger Länge $k' < 1920$ wird beim seriellen Laden periodisch eingelesen. Das Schlüsselbit $K_i, \quad 1 \leq i \leq 1920$ ergibt sich somit durch

$$K_i = K'_{i \bmod k'}$$

Sinnvoll ist dabei eine ungerade Schlüssellänge $k'$. Ist die Zahl nicht teilerfremd zur Zeilenzahl 128, so können Wiederholungen in gleichen Zeilen auftreten. Dies korreliert eventuell mit der Struktur des Netzwerks.

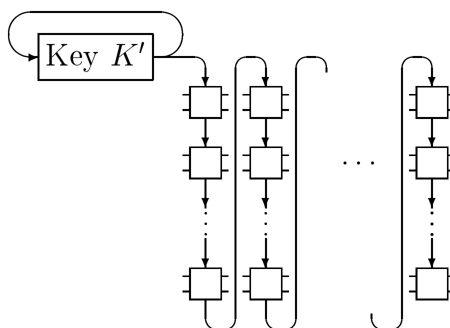



Abbildung 34: Laden des Schlüssels

## 8.2 Kontraktion der Hashmatrix

Ebenfalls aus Komplexitätsgründen empfielt es sich nicht, Hashwerte der Länge 1920 zu verwenden. Auch hier bietet sich an, diesen durch eine zusätzliche Kontraktionsfunktion auf 128, 256 oder 512 Bit zu verkleinern.

**Beispiel 8.10:**

Analog zur Expansionfunktion des Schlüssels läßt sich die Hashwertkontraktion durchführen. Soll der Zustandsmatrix auf einen Hashwert $H'$ der Länge $h'$ kontrahiert werden, so verwendet man dazu ein Ringschieberegister der Länge $h'$. Dieses wird mit '00 ...0' initialisiert, anschließend werden die Bits der Zustandsmatrix seriell ausgelesen und zyklisch aufaddiert. Die Kontraktionsfunktion lautet dann

$$H'_i = \bigoplus_{\substack{\forall z: \\ 0 \leq i + z \cdot h' < 1920}} K_z$$

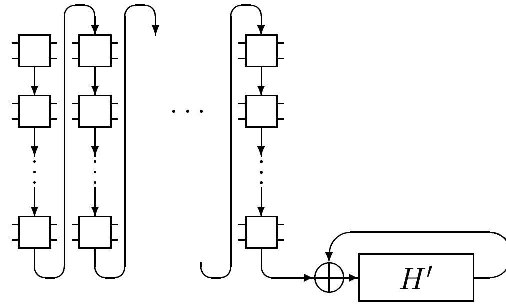



Abbildung 35: Einfache Kontraktion

**Beispiel 8.11:**

Bei der bereits implementierten Version des $\text{SINC}_8$ besteht der Zugriff auf die 15 Bit der untersten Zeile des Netzwerks. Sollen alle 1920 Bit der finalen Schlüsselmatrix in den Hashwert eingehen, so muß 128 mal spaltenweise geshiftet werden. Es bietet sich an, während dieser Operationen einen 128-Bit Hashwert seriell zu berechnen. Diese Lösung ist schneller, als das serielle Auslesen der einzelnen Bits.

Der Hashwert $H$ ergibt sich aus der Finalmatrix durch eine Kontraktionsfunktion $f$ der Form $f(K) : \{0,1\}^{1920} \rightarrow \{0,1\}^{128}$. Es sei $H := H_0, H_1, \ldots, H_{127}$. Berechnet man den Wert zeilenweise, so ist das $i$-te Bit $H_i$ lediglich abhängig von den 15 Bit der $i$-ten Zeile.

$$H_i(K) := f(K_{[i,i][1,15]})$$

Damit sich jedes Bit der Matrix auch tatsächlich auf den Hashwert auswirkt muß$f$ die Paritätsfunktion sein.

Eine bessere serielle Berechnung könnte erfolgen, wenn die Funktion $h$ von einem zusätzlichen Parameter $Z_i$ abhängig ist. Damit handelt es sich auch hier um einen endlichen Automaten. Festgelegt sind Initialzustand $Z_0$ und $h$. Die Bit des Hashwertes ergeben sich durch:

$$H_i(K) := h(Z_i, K_{[i,i][1,15]}), \quad Z_{i+1} := f(Z_i)$$

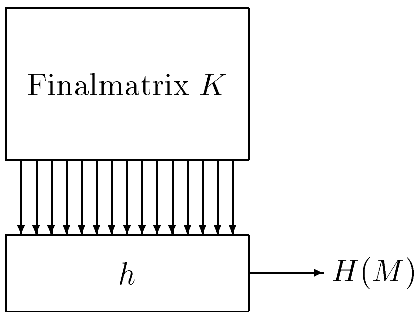



Abbildung 36: Kontraktion des Hashwertes

# 9 Abschließende Bemerkungen

Als Ergebnis dieser Arbeiten möchte ich folgendes zusammmenfassen. Die Verwendung von selbstmodifizierenden Verbindungsnetzwerken in der Kryptographie ist ein Forschungsgebiet, das in einigen Richtungen weiterverfolgt werden kann. Die von mir gemachten Beobachtungen, Vorschläge und Kriterien basieren größtenteils auf intuitiven Aspekten. Ein andere Betrachtungsweise ist die Analyse der Funktionen mit algebraischen Hilfsmitteln. Es ist fraglich, ob hier eine strukturierte Erfassung erfolgen kann, bzw. ob dadurch die Analyse vereinfacht wird. Für einen "Beweis" der Sicherheit, gemeint ist damit eine Rückführung auf andere algebraische bzw. mathematische Probleme, könnte dies hilfreich sein.

Nach meinem Dafürhalten ist die Verwendung der Beneš-, oder einer dazu äquivalenten Vernetzung effizient und praktikabel. Die Sicherheit des Systems als Verschlüsselungsmaschine, Pseudozufallsgenerator oder Hashfunktion hängt jedoch weitgehend von der Modifikation, dem Betriebsmodus und dem Verschlüsselungsprotokoll ab. An dieser Stelle ist es schwer, ein Urteil über die Sicherheit abzugeben. Es steht jedoch fest, daß elementare Operationen auf der Schlüsselmatrix kryptographische Standards nicht erfüllen. Die Verkettung vieler Elementaroperationen führt zu komplizierteren Strukturen, deren Eigenschaften sicherlich auch aufwendigen statistischen Analysen standhalten würde. Dies ist jedoch kein Garant für die Sicherheit des Systems. Eine strukturelle Analyse, durch "scharfes Anschauen" und selbstgewähltem Klartext kann hier in vielen Fällen das System brechen. Die von mir gemachten Vorschläge sind Kompromißlösungen zwischen technischer Realisierbarkeit, Effizienz und Sicherheit. Eine Selbstmodifikation zu finden, die allen Kriterien gerecht wird und deren Sicherheit begründet werden kann, ist keine leichte Aufgabe.

Ich versichere, diese Arbeit selbständig verfaßt und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt zu haben.

Karlsruhe, den 20. April 1993

(Felix Holderied)

# A Statistische Betrachtungen

Die Qualität eines Kryptosystems läßt sich mitunter durch statistische Kryptoanalyse bestimmen. Überprüft man die Chiffratfolgen eines strukturierten Klartextes nach verschiedenen Kriterien auf ihre Pseudozufälligkeit, so lassen sich bei schlechten Kryptosystemen starke Regelmäßigkeiten und Korrelationen entdecken. Ein Umkehrschluß ist hier jedoch nicht zulässig! Auch wenn das Chiffrat nach statistischen Methoden nicht von einer Zufallsfolge zu unterscheiden ist, so ist dies kein Garant für die Sicherheit des Systems. Als Beispiel seien hier linear rückgekoppelte Schieberegister (LFSR) genannt. Diese können Folgen mit sehr großer Periode erzeugen. Betrachtet man einen Teil einer solchen Folge, der wesentlich kleiner als die Periode ist, so genügt diese Folge allen Kriterien der Statistik. Trotzdem gibt es Algorithmen, die in effizienter Zeit die Belegung der Rückkopplungskoeffizienten berechnen.

Für die statistische Analyse des SINC habe ich verschiedene Testläufe erzeugt. Die einfachste Analyse besteht darin, einen strukturierten Klartext mit einer Zufallsinitialisierung der Schlüsselmatrix zu verschlüsseln, und die Häufigkeit der Chiffratzeichen zu analysieren. Als strukturierter Klartext dient die konstante Folge $\underbrace{0, 0, \ldots, 0}_{16384}$.

Die erzeugte Chiffratfolge sollte nun den Kriterien einer Zufallsfolge entsprechen. Das heißt, daß die Häufigkeit der einzelnen Zeichen etwa gleich ist, und die Verteilung der Häufigkeiten binomial. Diese Kriterien können beliebig verfeinert werden.

Mit Hilfe dieser Tests lassen sich nun die verschiedenen Modifikationen analysieren. Bei den elementaren Operationen genügt dabei meist schon ein Blick in die Häufigkeitstabelle, um Regelmäßigkeiten zu erkennen.

Die Sicherheit des $\text{SINC}_8$ mit einfacher Pfadinvertierung (5.2) kann durch diesen Test widerlegt werden. Tabelle 2 zeigt die Häufigkeiten der Chiffratzeichen bei zufälliger Initialisierung. Wie man sieht, treten als Häufigkeiten nur Vielfache von 8 auf. Dies liegt daran, daß das Chiffrat eine periodische Folge der Länge 2048 ist.

Ein weiteres "schlechtes" Beispiel ist die Modifikation der linken und rechten Nachbarn (5.4). Hier wird aus der Statistik (Tabelle 3) deutlich, daß das Chiffrat nach 5 Zeichen in die konstante Folge $58, 58, \ldots$ übergeht.

Modifiziert man alle Nachbarn (5.5) so ergibt sich für die Häufigkeiten eine bessere Gleichverteilung (siehe Tabelle 4). Diese erfüllt jedoch auch keine Binomialverteilung. Man erkennt, daß die Summen der Häufigkeiten zweier benachbarter Ausgänge $(|c_{2i}| + |c_{2i+1}|)$ wesentlich gleichmäßiger verteilt sind als die einzelnen Chiffratzeichen. Dies liegt in der Struktur des Netzwerks begründet.

Auch die Selbstmodifikation 5.6 weist Schwächen auf. Hier ist auffallend, daß drei Chiffrate nie auftreten. Dies ist ebenfalls strukturbedingt (siehe 5.6).

Ein Beispiel für eine Kombination mehrerer Operationen ist in Tabelle 7 zu sehen. Hier wurde Modifikation 5.6 mit einem zyklischen Spaltenshift, sowie der Modifikation 5.10 verkettet. Diese Statistik weist eine recht gute Gleichverteilung über den Chiffratzeichen auf.

| $c_i$ | $\lvert c_i \rvert$ | $c_i$ | $\lvert c_i \rvert$ | $c_i$ | $\lvert c_i \rvert$ | $c_i$ | $\lvert c_i \rvert$ | $c_i$ | $\lvert c_i \rvert$ | $c_i$ | $\lvert c_i \rvert$ | $c_i$ | $\lvert c_i \rvert$ | $c_i$ | $\lvert c_i \rvert$ |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 16 | 32 | 40 | 64 | 8 | 96 | 16 | 128 | 16 | 160 | 104 | 192 | 16 | 224 | 16 |
| 1 | 8 | 33 | 64 | 65 | 16 | 97 | 24 | 129 | 24 | 161 | 80 | 193 | 24 | 225 | 8 |
| 2 | 16 | 34 | 40 | 66 | 8 | 98 | 16 | 130 | 16 | 162 | 104 | 194 | 16 | 226 | 16 |
| 3 | 8 | 35 | 64 | 67 | 16 | 99 | 24 | 131 | 24 | 163 | 80 | 195 | 24 | 227 | 8 |
| 4 | 0 | 36 | 80 | 68 | 0 | 100 | 8 | 132 | 16 | 164 | 56 | 196 | 16 | 228 | 0 |
| 5 | 8 | 37 | 40 | 69 | 8 | 101 | 16 | 133 | 8 | 165 | 48 | 197 | 8 | 229 | 8 |
| 6 | 0 | 38 | 80 | 70 | 0 | 102 | 8 | 134 | 16 | 166 | 56 | 198 | 16 | 230 | 8 |
| 7 | 8 | 39 | 40 | 71 | 8 | 103 | 16 | 135 | 8 | 167 | 48 | 199 | 8 | 231 | 0 |
| 8 | 8 | 40 | 64 | 72 | 8 | 104 | 16 | 136 | 8 | 168 | 72 | 200 | 8 | 232 | 8 |
| 9 | 32 | 41 | 56 | 73 | 32 | 105 | 8 | 137 | 16 | 169 | 64 | 201 | 0 | 233 | 32 |
| 10 | 8 | 42 | 56 | 74 | 8 | 106 | 16 | 138 | 8 | 170 | 72 | 202 | 8 | 234 | 8 |
| 11 | 32 | 43 | 64 | 75 | 32 | 107 | 8 | 139 | 16 | 171 | 64 | 203 | 0 | 235 | 32 |
| 12 | 8 | 44 | 96 | 76 | 16 | 108 | 0 | 140 | 0 | 172 | 32 | 204 | 8 | 236 | 8 |
| 13 | 16 | 45 | 40 | 77 | 8 | 109 | 8 | 141 | 8 | 173 | 24 | 205 | 16 | 237 | 16 |
| 14 | 8 | 46 | 96 | 78 | 8 | 110 | 0 | 142 | 0 | 174 | 32 | 206 | 8 | 238 | 8 |
| 15 | 16 | 47 | 40 | 79 | 16 | 111 | 8 | 143 | 8 | 175 | 24 | 207 | 16 | 239 | 16 |
| 16 | 8 | 48 | 496 | 80 | 8 | 112 | 8 | 144 | 16 | 176 | 656 | 208 | 8 | 240 | 8 |
| 17 | 0 | 49 | 568 | 81 | 0 | 113 | 0 | 145 | 24 | 177 | 616 | 209 | 16 | 241 | 0 |
| 18 | 8 | 50 | 496 | 82 | 0 | 114 | 8 | 146 | 16 | 178 | 656 | 210 | 8 | 242 | 0 |
| 19 | 0 | 51 | 568 | 83 | 8 | 115 | 0 | 147 | 24 | 179 | 616 | 211 | 16 | 243 | 8 |
| 20 | 16 | 52 | 656 | 84 | 16 | 116 | 16 | 148 | 16 | 180 | 504 | 212 | 24 | 244 | 16 |
| 21 | 8 | 53 | 552 | 85 | 8 | 117 | 8 | 149 | 8 | 181 | 464 | 213 | 16 | 245 | 8 |
| 22 | 16 | 54 | 552 | 86 | 16 | 118 | 16 | 150 | 16 | 182 | 464 | 214 | 24 | 246 | 16 |
| 23 | 8 | 55 | 656 | 87 | 8 | 119 | 8 | 151 | 8 | 183 | 504 | 215 | 16 | 247 | 8 |
| 24 | 8 | 56 | 208 | 88 | 8 | 120 | 24 | 152 | 8 | 184 | 248 | 216 | 8 | 248 | 32 |
| 25 | 16 | 57 | 168 | 89 | 16 | 121 | 16 | 153 | 0 | 185 | 272 | 217 | 0 | 249 | 8 |
| 26 | 8 | 58 | 208 | 90 | 16 | 122 | 24 | 154 | 8 | 186 | 248 | 218 | 8 | 250 | 32 |
| 27 | 16 | 59 | 168 | 91 | 8 | 123 | 16 | 155 | 0 | 187 | 272 | 219 | 0 | 251 | 8 |
| 28 | 32 | 60 | 184 | 92 | 32 | 124 | 8 | 156 | 8 | 188 | 152 | 220 | 8 | 252 | 8 |
| 29 | 8 | 61 | 208 | 93 | 8 | 125 | 16 | 157 | 16 | 189 | 128 | 221 | 16 | 253 | 16 |
| 30 | 32 | 62 | 184 | 94 | 32 | 126 | 8 | 158 | 8 | 190 | 128 | 222 | 16 | 254 | 8 |
| 31 | 8 | 63 | 208 | 95 | 8 | 127 | 16 | 159 | 16 | 191 | 152 | 223 | 8 | 255 | 16 |

Tabelle 2: Testfolge zur einfachen Pfadmodifikation (5.2)

| $c_i$ | $\|c_i\|$ | $c_i$ | $\|c_i\|$ | $c_i$ | $\|c_i\|$ | $c_i$ | $\|c_i\|$ | $c_i$ | $\|c_i\|$ | $c_i$ | $\|c_i\|$ | $c_i$ | $\|c_i\|$ | $c_i$ | $\|c_i\|$ |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 32 | 0 | 64 | 0 | 96 | 0 | 128 | 0 | 160 | 1 | 192 | 0 | 224 | 0 |
| 1 | 0 | 33 | 0 | 65 | 0 | 97 | 0 | 129 | 0 | 161 | 0 | 193 | 0 | 225 | 0 |
| 2 | 0 | 34 | 0 | 66 | 0 | 98 | 0 | 130 | 0 | 162 | 1 | 194 | 0 | 226 | 0 |
| 3 | 0 | 35 | 0 | 67 | 0 | 99 | 0 | 131 | 0 | 163 | 1 | 195 | 0 | 227 | 0 |
| 4 | 0 | 36 | 0 | 68 | 0 | 100 | 0 | 132 | 0 | 164 | 0 | 196 | 0 | 228 | 0 |
| 5 | 0 | 37 | 0 | 69 | 0 | 101 | 0 | 133 | 0 | 165 | 0 | 197 | 0 | 229 | 0 |
| 6 | 0 | 38 | 0 | 70 | 0 | 102 | 0 | 134 | 0 | 166 | 0 | 198 | 0 | 230 | 0 |
| 7 | 0 | 39 | 0 | 71 | 0 | 103 | 0 | 135 | 0 | 167 | 0 | 199 | 0 | 231 | 0 |
| 8 | 0 | 40 | 0 | 72 | 0 | 104 | 0 | 136 | 0 | 168 | 0 | 200 | 0 | 232 | 0 |
| 9 | 0 | 41 | 0 | 73 | 0 | 105 | 0 | 137 | 0 | 169 | 0 | 201 | 0 | 233 | 0 |
| 10 | 0 | 42 | 0 | 74 | 0 | 106 | 0 | 138 | 0 | 170 | 1 | 202 | 0 | 234 | 0 |
| 11 | 0 | 43 | 0 | 75 | 0 | 107 | 0 | 139 | 0 | 171 | 0 | 203 | 0 | 235 | 0 |
| 12 | 0 | 44 | 0 | 76 | 0 | 108 | 0 | 140 | 0 | 172 | 0 | 204 | 0 | 236 | 0 |
| 13 | 0 | 45 | 0 | 77 | 0 | 109 | 0 | 141 | 0 | 173 | 0 | 205 | 0 | 237 | 0 |
| 14 | 0 | 46 | 0 | 78 | 0 | 110 | 0 | 142 | 0 | 174 | 0 | 206 | 0 | 238 | 0 |
| 15 | 0 | 47 | 0 | 79 | 0 | 111 | 0 | 143 | 0 | 175 | 0 | 207 | 0 | 239 | 0 |
| 16 | 0 | 48 | 0 | 80 | 0 | 112 | 0 | 144 | 0 | 176 | 0 | 208 | 0 | 240 | 0 |
| 17 | 0 | 49 | 0 | 81 | 0 | 113 | 0 | 145 | 0 | 177 | 0 | 209 | 0 | 241 | 0 |
| 18 | 0 | 50 | 0 | 82 | 0 | 114 | 0 | 146 | 0 | 178 | 0 | 210 | 0 | 242 | 0 |
| 19 | 0 | 51 | 0 | 83 | 0 | 115 | 0 | 147 | 0 | 179 | 0 | 211 | 0 | 243 | 0 |
| 20 | 0 | 52 | 0 | 84 | 0 | 116 | 0 | 148 | 0 | 180 | 0 | 212 | 0 | 244 | 0 |
| 21 | 0 | 53 | 0 | 85 | 0 | 117 | 0 | 149 | 0 | 181 | 0 | 213 | 0 | 245 | 0 |
| 22 | 0 | 54 | 0 | 86 | 0 | 118 | 0 | 150 | 0 | 182 | 0 | 214 | 0 | 246 | 0 |
| 23 | 0 | 55 | 0 | 87 | 0 | 119 | 0 | 151 | 0 | 183 | 0 | 215 | 0 | 247 | 0 |
| 24 | 0 | 56 | 0 | 88 | 0 | 120 | 0 | 152 | 0 | 184 | 0 | 216 | 0 | 248 | 0 |
| 25 | 0 | 57 | 0 | 89 | 0 | 121 | 0 | 153 | 0 | 185 | 0 | 217 | 0 | 249 | 0 |
| 26 | 0 | 58 | 16379 | 90 | 0 | 122 | 0 | 154 | 0 | 186 | 1 | 218 | 0 | 250 | 0 |
| 27 | 0 | 59 | 0 | 91 | 0 | 123 | 0 | 155 | 0 | 187 | 0 | 219 | 0 | 251 | 0 |
| 28 | 0 | 60 | 0 | 92 | 0 | 124 | 0 | 156 | 0 | 188 | 0 | 220 | 0 | 252 | 0 |
| 29 | 0 | 61 | 0 | 93 | 0 | 125 | 0 | 157 | 0 | 189 | 0 | 221 | 0 | 253 | 0 |
| 30 | 0 | 62 | 0 | 94 | 0 | 126 | 0 | 158 | 0 | 190 | 0 | 222 | 0 | 254 | 0 |
| 31 | 0 | 63 | 0 | 95 | 0 | 127 | 0 | 159 | 0 | 191 | 0 | 223 | 0 | 255 | 0 |

Tabelle 3: Testfolge zur Modifikation der rechten und linken Nachbarn (5.4)

| $c_i$ | $\|c_i\|$ | $c_i$ | $\|c_i\|$ | $c_i$ | $\|c_i\|$ | $c_i$ | $\|c_i\|$ | $c_i$ | $\|c_i\|$ | $c_i$ | $\|c_i\|$ | $c_i$ | $\|c_i\|$ | $c_i$ | $\|c_i\|$ |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 32 | 0 | 64 | 0 | 96 | 0 | 128 | 0 | 160 | 1 | 192 | 0 | 224 | 0 |
| 1 | 0 | 33 | 0 | 65 | 0 | 97 | 0 | 129 | 0 | 161 | 0 | 193 | 0 | 225 | 0 |
| 2 | 0 | 34 | 0 | 66 | 0 | 98 | 0 | 130 | 0 | 162 | 1 | 194 | 0 | 226 | 0 |
| 3 | 0 | 35 | 0 | 67 | 0 | 99 | 0 | 131 | 0 | 163 | 1 | 195 | 0 | 227 | 0 |
| 4 | 0 | 36 | 0 | 68 | 0 | 100 | 0 | 132 | 0 | 164 | 0 | 196 | 0 | 228 | 0 |
| 5 | 0 | 37 | 0 | 69 | 0 | 101 | 0 | 133 | 0 | 165 | 0 | 197 | 0 | 229 | 0 |
| 6 | 0 | 38 | 0 | 70 | 0 | 102 | 0 | 134 | 0 | 166 | 0 | 198 | 0 | 230 | 0 |
| 7 | 0 | 39 | 0 | 71 | 0 | 103 | 0 | 135 | 0 | 167 | 0 | 199 | 0 | 231 | 0 |
| 8 | 0 | 40 | 0 | 72 | 0 | 104 | 0 | 136 | 0 | 168 | 0 | 200 | 0 | 232 | 0 |
| 9 | 0 | 41 | 0 | 73 | 0 | 105 | 0 | 137 | 0 | 169 | 0 | 201 | 0 | 233 | 0 |
| 10 | 0 | 42 | 0 | 74 | 0 | 106 | 0 | 138 | 0 | 170 | 1 | 202 | 0 | 234 | 0 |
| 11 | 0 | 43 | 0 | 75 | 0 | 107 | 0 | 139 | 0 | 171 | 0 | 203 | 0 | 235 | 0 |
| 12 | 0 | 44 | 0 | 76 | 0 | 108 | 0 | 140 | 0 | 172 | 0 | 204 | 0 | 236 | 0 |
| 13 | 0 | 45 | 0 | 77 | 0 | 109 | 0 | 141 | 0 | 173 | 0 | 205 | 0 | 237 | 0 |
| 14 | 0 | 46 | 0 | 78 | 0 | 110 | 0 | 142 | 0 | 174 | 0 | 206 | 0 | 238 | 0 |
| 15 | 0 | 47 | 0 | 79 | 0 | 111 | 0 | 143 | 0 | 175 | 0 | 207 | 0 | 239 | 0 |
| 16 | 0 | 48 | 0 | 80 | 0 | 112 | 0 | 144 | 0 | 176 | 0 | 208 | 0 | 240 | 0 |
| 17 | 0 | 49 | 0 | 81 | 0 | 113 | 0 | 145 | 0 | 177 | 0 | 209 | 0 | 241 | 0 |
| 18 | 0 | 50 | 0 | 82 | 0 | 114 | 0 | 146 | 0 | 178 | 0 | 210 | 0 | 242 | 0 |
| 19 | 0 | 51 | 0 | 83 | 0 | 115 | 0 | 147 | 0 | 179 | 0 | 211 | 0 | 243 | 0 |
| 20 | 0 | 52 | 0 | 84 | 0 | 116 | 0 | 148 | 0 | 180 | 0 | 212 | 0 | 244 | 0 |
| 21 | 0 | 53 | 0 | 85 | 0 | 117 | 0 | 149 | 0 | 181 | 0 | 213 | 0 | 245 | 0 |
| 22 | 0 | 54 | 0 | 86 | 0 | 118 | 0 | 150 | 0 | 182 | 0 | 214 | 0 | 246 | 0 |
| 23 | 0 | 55 | 0 | 87 | 0 | 119 | 0 | 151 | 0 | 183 | 0 | 215 | 0 | 247 | 0 |
| 24 | 0 | 56 | 0 | 88 | 0 | 120 | 0 | 152 | 0 | 184 | 0 | 216 | 0 | 248 | 0 |
| 25 | 0 | 57 | 0 | 89 | 0 | 121 | 0 | 153 | 0 | 185 | 0 | 217 | 0 | 249 | 0 |
| 26 | 0 | 58 | 16379 | 90 | 0 | 122 | 0 | 154 | 0 | 186 | 1 | 218 | 0 | 250 | 0 |
| 27 | 0 | 59 | 0 | 91 | 0 | 123 | 0 | 155 | 0 | 187 | 0 | 219 | 0 | 251 | 0 |
| 28 | 0 | 60 | 0 | 92 | 0 | 124 | 0 | 156 | 0 | 188 | 0 | 220 | 0 | 252 | 0 |
| 29 | 0 | 61 | 0 | 93 | 0 | 125 | 0 | 157 | 0 | 189 | 0 | 221 | 0 | 253 | 0 |
| 30 | 0 | 62 | 0 | 94 | 0 | 126 | 0 | 158 | 0 | 190 | 0 | 222 | 0 | 254 | 0 |
| 31 | 0 | 63 | 0 | 95 | 0 | 127 | 0 | 159 | 0 | 191 | 0 | 223 | 0 | 255 | 0 |

Tabelle 4: Testfolge zu Modifikation aller Nachbarn (5.5)

| $c_i$ | $\lvert c_i \rvert$ | $c_i$ | $\lvert c_i \rvert$ | $c_i$ | $\lvert c_i \rvert$ | $c_i$ | $\lvert c_i \rvert$ | $c_i$ | $\lvert c_i \rvert$ | $c_i$ | $\lvert c_i \rvert$ | $c_i$ | $\lvert c_i \rvert$ | $c_i$ | $\lvert c_i \rvert$ |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 319 | 32 | 51 | 64 | 65 | 96 | 85 | 128 | 50 | 160 | 72 | 192 | 55 | 224 | 71 |
| 1 | 0 | 33 | 70 | 65 | 77 | 97 | 92 | 129 | 63 | 161 | 64 | 193 | 70 | 225 | 68 |
| 2 | 28 | 34 | 52 | 66 | 71 | 98 | 89 | 130 | 65 | 162 | 63 | 194 | 72 | 226 | 65 |
| 3 | 35 | 35 | 66 | 67 | 54 | 99 | 95 | 131 | 66 | 163 | 64 | 195 | 67 | 227 | 59 |
| 4 | 40 | 36 | 68 | 68 | 56 | 100 | 28 | 132 | 69 | 164 | 69 | 196 | 58 | 228 | 0 |
| 5 | 38 | 37 | 67 | 69 | 64 | 101 | 21 | 133 | 72 | 165 | 72 | 197 | 67 | 229 | 136 |
| 6 | 34 | 38 | 55 | 70 | 67 | 102 | 18 | 134 | 54 | 166 | 57 | 198 | 72 | 230 | 64 |
| 7 | 35 | 39 | 53 | 71 | 53 | 103 | 27 | 135 | 72 | 167 | 70 | 199 | 68 | 231 | 62 |
| 8 | 68 | 40 | 61 | 72 | 56 | 104 | 62 | 136 | 71 | 168 | 72 | 200 | 64 | 232 | 72 |
| 9 | 52 | 41 | 64 | 73 | 72 | 105 | 65 | 137 | 64 | 169 | 67 | 201 | 78 | 233 | 59 |
| 10 | 88 | 42 | 62 | 74 | 51 | 106 | 56 | 138 | 66 | 170 | 64 | 202 | 64 | 234 | 55 |
| 11 | 74 | 43 | 63 | 75 | 55 | 107 | 58 | 139 | 61 | 171 | 73 | 203 | 61 | 235 | 74 |
| 12 | 61 | 44 | 57 | 76 | 56 | 108 | 58 | 140 | 71 | 172 | 74 | 204 | 86 | 236 | 71 |
| 13 | 62 | 45 | 57 | 77 | 48 | 109 | 74 | 141 | 63 | 173 | 64 | 205 | 76 | 237 | 71 |
| 14 | 67 | 46 | 55 | 78 | 61 | 110 | 57 | 142 | 63 | 174 | 47 | 206 | 78 | 238 | 65 |
| 15 | 85 | 47 | 48 | 79 | 59 | 111 | 70 | 143 | 82 | 175 | 79 | 207 | 58 | 239 | 77 |
| 16 | 63 | 48 | 71 | 80 | 63 | 112 | 57 | 144 | 63 | 176 | 61 | 208 | 57 | 240 | 74 |
| 17 | 61 | 49 | 57 | 81 | 59 | 113 | 60 | 145 | 62 | 177 | 52 | 209 | 58 | 241 | 68 |
| 18 | 53 | 50 | 59 | 82 | 59 | 114 | 67 | 146 | 52 | 178 | 68 | 210 | 60 | 242 | 69 |
| 19 | 70 | 51 | 56 | 83 | 73 | 115 | 69 | 147 | 75 | 179 | 72 | 211 | 79 | 243 | 60 |
| 20 | 56 | 52 | 31 | 84 | 49 | 116 | 46 | 148 | 66 | 180 | 53 | 212 | 61 | 244 | 62 |
| 21 | 85 | 53 | 26 | 85 | 65 | 117 | 67 | 149 | 61 | 181 | 61 | 213 | 56 | 245 | 55 |
| 22 | 61 | 54 | 91 | 86 | 49 | 118 | 57 | 150 | 70 | 182 | 74 | 214 | 62 | 246 | 61 |
| 23 | 73 | 55 | 98 | 87 | 57 | 119 | 76 | 151 | 67 | 183 | 74 | 215 | 64 | 247 | 61 |
| 24 | 66 | 56 | 63 | 88 | 59 | 120 | 70 | 152 | 70 | 184 | 80 | 216 | 59 | 248 | 64 |
| 25 | 63 | 57 | 68 | 89 | 61 | 121 | 53 | 153 | 67 | 185 | 65 | 217 | 76 | 249 | 72 |
| 26 | 55 | 58 | 50 | 90 | 62 | 122 | 70 | 154 | 60 | 186 | 63 | 218 | 66 | 250 | 62 |
| 27 | 66 | 59 | 66 | 91 | 57 | 123 | 71 | 155 | 73 | 187 | 54 | 219 | 68 | 251 | 74 |
| 28 | 0 | 60 | 74 | 92 | 66 | 124 | 55 | 156 | 63 | 188 | 66 | 220 | 65 | 252 | 60 |
| 29 | 137 | 61 | 67 | 93 | 68 | 125 | 62 | 157 | 65 | 189 | 74 | 221 | 56 | 253 | 63 |
| 30 | 68 | 62 | 59 | 94 | 61 | 126 | 61 | 158 | 59 | 190 | 67 | 222 | 56 | 254 | 65 |
| 31 | 55 | 63 | 66 | 95 | 67 | 127 | 59 | 159 | 70 | 191 | 63 | 223 | 68 | 255 | 63 |

Tabelle 5: Testfolge zur Diagonal/Horizontalmodifikation (5.6)

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

Tabelle 6: Testfolge zu (5.6) mit Beneš-äquivalenter Vernetzung

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

Tabelle 7: Testfolge zu Kombination mehrerer Selbstmodifikationen

Abbildung 37: Hier gehört eigentlich Oberfläche des Simulationsprogramms hin.

# B Ein Simulationsprogramm

Zur Illustration und zum Test verschiedener Selbstmodifikationen habe ich ein C-Programm unter X-Windows geschrieben. Abbildung 37 zeigt die Benutzeroberfläche des Programms. Im linken Fenster wird die aktuelle Schlüsselmatrix (`key`) dargestellt ($0 \hat{=}$ weiß, $1 \hat{=}$ schwarz). Die Initialisierung und Änderung dieser Matrix erfolgt mit Hilfe der Maus. Linke Maustaste: Anklicken eines Tauscher invertiert den Tauscher; Mittlere Maustaste: Klicken in eine Zeile der Schlüsselmatrix invertiert die Zeile; Rechte Maustaste: Invertieren einer Spalte. Die Matrix `inversion` zeigt die Differenz der letzten beiden Schlüsselmatrizen.

Im Menufenster stehen verschiedene Optionen und Anweisungen zur Auswahl. `Random Input` erzeugt einen zufälligen Klartext. Mit `Manual Input` besteht die Möglichkeit den Klartext entweder Zeichen für Zeichen einzugeben und verschlüsseln zu lassen, oder über eine pipe beim Starten des Programms einzulesen. Die Befehle `Save Key` und `Save Invers.` speichern die Matrizen in LaTeX-Format unter dem Namen `Matrix`. Mit `Step`, `Step 8`, `Step 64`, `Step 8192` wird eine Verschlüsselung von einem, 8, 64 oder 8192 Zeichen durchgeführt. Während der Verschlüsselung werden die Häufigkeiten der Chiffratzeichen protokolliert. `Clear Stat.` löscht diese Statistik und `Save Stat.` schreibt die Häufigkeiten auf eine Datei `Statistik`. Durch `Net` kann zwischen einem Beneš-Netzwerk (`0`) und einem Beneš-äquivalenten Netzwerk (`1`) ausgewählt werden. Durch `Invers.` wird eine von 6 Selbstmodifikationen ausgewählt. Stellt man den `Show Matrix`-Modus ab, so läuft das Programm um ein vielfaches schneller. Dies ist notwendig bei längeren Testfolgen. `Clear` löscht den Schlüssel, `Quit` beendet das Programm.

Der Quelltext des Programms:

```
/*********************************************************************/
/*                                                                   */
/*            This is an experimental Implementation of a            */
/*            Selfmodifying Interconnection-Network (SINC)           */
/*            with X-display                                         */
/*                                                                   */
/*            This program is part of a Diploma-Work                 */
/*            Further informations see:                              */
/*                                                                   */
/*         Diplomarbeit:                                             */
/*            Kryptographische Aspekte Selbstmodifizierender         */
/*            Verbindungsnetzwerke                                   */
/*                                                                   */
/*         Author: Felix Holderied                                   */
/*                                                                   */
/*         Betreuer: Priv. Doz. Dr. P Horster                        */
/*                                                                   */
/*            Institut fuer Algorithmen und kognitive Systeme        */
/*            Universitaet Karlsruhe                                 */
/*                                                                   */
/*         Date:   February 1993                                     */
/*                                                                   */
```

```
/*********************************************************************/

#include <stdio.h>
#include <X11/X.h>
#include <X11/Xlib.h>
#include <X11/Xatom.h>
#include <X11/Xutil.h>

extern Display* XOpendisplay();
Window ik_window, buttons_window, root;
Display* display;
GC gc_draw, gc_erase, gc_buttons_draw, gc_buttons_erase;
XSetWindowAttributes ik_atts, buttons_atts;
XEvent xevent;

/* These are the coos of the left two corners of the matrices */
#define xkm 35
#define ykm 20
#define xim 140
#define yim 20

void inversion_0(),inversion_1(),inversion_2(),inversion_3(),
    inversion_4(),inversion_5(),inversion_6();
void (*inversion_functions[7])()=
  { inversion_0, inversion_1, inversion_2, inversion_3,
    inversion_4, inversion_5, inversion_6 };

static keys[15][128];
static inversion[15][128];
static path[15];
static statistic[256];
static stat2[1000];
int net=1;
int inv=0;
int m=1;
static perms[15][256];           /* Last permutation must be identity */

static int randommode=1;

void clear_keys()                /* Sets the key-matrix to (0) */
{
  int i,j;

  for(i=0;i<15;i++)
    for(j=0;j<128;j++)
      keys[i][j]=0;
}
```

```
void clear_stat()                /* Sets both statistics to (0) */
{
  int i;
  for(i=0;i<256;i++)
    statistic[i]=0;
  for(i=0;i<1000;i++)
    stat2[i]=0;
}

void clear_inversion()           /* Sets the inversion-matrix to (0) */
{
  int i,j;

  for(i=0;i<15;i++)
    for(j=0;j<128;j++)
      inversion[i][j]=0;
}

void benes_perms()        /* sets the net-permutations Q to a benes net */
{
  int i,j;

  for(j=0;j<256;j++){
    perms[0][j]=(j/256)*256+(j%2)*128+(j%256)/2;
    perms[1][j]=(j/128)*128+(j%2)* 64+(j%128)/2;
    perms[2][j]=(j/ 64)* 64+(j%2)* 32+(j% 64)/2;
    perms[3][j]=(j/ 32)* 32+(j%2)* 16+(j% 32)/2;
    perms[4][j]=(j/ 16)* 16+(j%2)*  8+(j% 16)/2;
    perms[5][j]=(j/  8)*  8+(j%2)*  4+(j%  8)/2;
    perms[6][j]=(j/  4)*  4+(j%2)*  2+(j%  4)/2;
  };
  for(i=7;i<14;i++)
    for(j=0;j<256;j++)
      perms[i][perms[13-i][j]]=j;

  for(j=0;j<256;j++)
    perms[14][j]=j;
}

void benes_equiv_perms()
                      /* sets the net-permutations Q to a benes-equivalent net */
                      /* every row of switching elements is shifted cyclic     */
{
  int i,j;

  for(j=0;j<256;j++){
```

```
    perms[0][j]           =((j/256)*256+(j%2)*128+(j%256)/2+ 10)%256;
    perms[1][(j+ 10)%256] =((j/128)*128+(j%2)* 64+(j%128)/2+180)%256;
    perms[2][(j+180)%256] =((j/ 64)* 64+(j%2)* 32+(j% 64)/2+ 70)%256;
    perms[3][(j+ 70)%256] =((j/ 32)* 32+(j%2)* 16+(j% 32)/2+120)%256;
    perms[4][(j+120)%256] =((j/ 16)* 16+(j%2)*  8+(j% 16)/2+ 60)%256;
    perms[5][(j+ 60)%256] =((j/  8)*  8+(j%2)*  4+(j%  8)/2+210)%256;
    perms[6][(j+210)%256] =((j/  4)*  4+(j%2)*  2+(j%  4)/2+110)%256;
  };
  for(i=7;i<14;i++)
    for(j=0;j<256;j++)
      perms[i][perms[13-i][j]]=j;

  for(j=0;j<256;j++)
    perms[14][j]=j;
}


void xinit()
{
  XGCValues gcvalue;
  XSizeHints* sizehints;

  if(! (display=XOpenDisplay(""))){
    printf("Sorry, could'nt open display\n");
    exit(1);
  };

  root = DefaultRootWindow(display);

  /* init window for displaying the key and inversion matrix*/
  ik_atts.event_mask        = ExposureMask|ButtonPressMask;
  ik_atts.background_pixel  = WhitePixel(display, DefaultScreen(display));
  ik_atts.backing_store     = Always;
  ik_window = XCreateWindow(display, root, 300, 0, 250, 680, 2, 0,
   InputOutput, CopyFromParent,
   CWEventMask |  CWBackPixel | CWBackingStore,
   &ik_atts);
  XStoreName(display, ik_window, "key and inversion matrix");
  XMapWindow(display, ik_window);


  /* init button window */
  buttons_atts.event_mask        = ExposureMask|ButtonPressMask;
  buttons_atts.background_pixel  = WhitePixel(display, DefaultScreen(display));
  buttons_atts.backing_store     = Always;
  buttons_window = XCreateWindow(display, root, 0, 0, 113, 443, 2, 0,
 InputOutput, CopyFromParent,
```

```
 CWEventMask |  CWBackPixel | CWBackingStore,
 &buttons_atts);
  XStoreName(display,buttons_window,"menu");
  XMapWindow(display, buttons_window);

  /* Set the graphical context */
  gcvalue.function          = GXcopy;
  gcvalue.plane_mask        = AllPlanes;
  gcvalue.background        = WhitePixel(display,DefaultScreen(display));
  gcvalue.foreground        = BlackPixel(display,DefaultScreen(display));
  gcvalue.line_width        = 1;
  gcvalue.line_style        = LineSolid;
  gcvalue.fill_style        = FillSolid;
  gc_draw = XCreateGC(display, ik_window,
 GCFunction|GCPlaneMask|GCForeground|GCBackground|
 GCLineWidth|GCFillStyle|GCLineStyle, &gcvalue);
  gc_buttons_draw = XCreateGC(display, buttons_window,
      GCFunction|GCPlaneMask|GCForeground|GCBackground|
      GCLineWidth|GCFillStyle|GCLineStyle, &gcvalue);
  gcvalue.foreground        = WhitePixel(display,DefaultScreen(display));
  gc_erase = XCreateGC(display, ik_window,
  GCFunction|GCPlaneMask|GCForeground|GCBackground|
  GCLineWidth|GCFillStyle|GCLineStyle, &gcvalue);
  gc_buttons_erase = XCreateGC(display, buttons_window,
       GCFunction|GCPlaneMask|GCForeground|GCBackground|
       GCLineWidth|GCFillStyle|GCLineStyle, &gcvalue);

  XFlush(display);
}

void draw_key_pixel(int row,int column)
{
  if( (row<0) || (row>127) || (column<1) || (column>15) ) return;

  XFillRectangle(display,ik_window,gc_draw,xkm+1+5*(column-1),ykm+1+5*row,4,4);
  XFlush(display);
}

void erase_key_pixel(int row,int column)
{
  if( (row<0) || (row>127) || (column<1) || (column>15) ) return;

  XFillRectangle(display,ik_window,gc_erase,xkm+1+5*(column-1),ykm+1+5*row,4,4);
  XFlush(display);
}

void draw_inversion_pixel(int row,int column)
```

```
{
  if( (row<0) || (row>127) || (column<1) || (column>15) ) return;

  XFillRectangle(display,ik_window,gc_draw,xim+1+5*(column-1),yim+1+5*row,4,4);
  XFlush(display);
}

void erase_inversion_pixel(int row,int column)
{
  if( (row<0) || (row>127) || (column<1) || (column>15) ) return;
  XFillRectangle(display,ik_window,gc_erase,xim+1+5*(column-1),yim+1+5*row,4,4);
  XFlush(display);
}

void flip_key_pixel(int row,int column)
{
  if( (row<0) || (row>127) || (column<1) || (column>15) ) return;
  if(keys[column-1][row]){
    keys[column-1][row]=0;
    XFillRectangle(display,ik_window,gc_erase,xkm+1+5*(column-1),ykm+1+5*row,4,4);
  } else {
    keys[column-1][row]=1;
    XFillRectangle(display,ik_window,gc_draw,xkm+1+5*(column-1),ykm+1+5*row,4,4);
  };
  XFlush(display);
}

void draw_keys_matrix()
{
  int x,y;

  for(x=0;x<15;x++)
    for(y=0;y<128;y++)
      if(keys[x][y])
draw_key_pixel(y,x+1);
      else
erase_key_pixel(y,x+1);
  XFlush(display);
}

void draw_inversion_matrix()
{
  int x,y;

  for(x=0;x<15;x++)
    for(y=0;y<128;y++)
      if(inversion[x][y])
```

```
draw_inversion_pixel(y,x+1);
      else
erase_inversion_pixel(y,x+1);
  XFlush(display);
}

void draw_matrices()
{
  int x,y;

  XDrawRectangle(display,ik_window,gc_draw,xkm-1,ykm-1,77,642);
  XDrawRectangle(display,ik_window,gc_draw,xim-1,yim-1,77,642);

  for(x=0; x<=75; x+=5){
    XDrawLine(display,ik_window,gc_draw,x+xkm,ykm,x+xkm,ykm+640);
    XDrawLine(display,ik_window,gc_draw,x+xim,yim,x+xim,yim+640);
  }
  for(y=0;y<=640; y+=5){
    XDrawLine(display,ik_window,gc_draw,xkm,y+ykm,xkm+76,y+ykm);
    XDrawLine(display,ik_window,gc_draw,xim,y+yim,xim+76,y+yim);
  }

  XDrawString(display,ik_window,gc_draw,xkm+31,ykm+652,"key",3);
  XDrawString(display,ik_window,gc_draw,xim+11,yim+652,"inversion",9);

  for(y=0;y<=640;y+=32*5){
    XDrawLine(display,ik_window,gc_draw,xkm-1,ykm-1+y,xkm-13,ykm-1+y);
    XDrawLine(display,ik_window,gc_draw,xkm-1,ykm+y,xkm-13,ykm+y);
    XDrawLine(display,ik_window,gc_draw,xkm-1,ykm+1+y,xkm-13,ykm+1+y);

    XDrawLine(display,ik_window,gc_draw,xim+77,yim-1+y,xim+88,yim-1+y);
    XDrawLine(display,ik_window,gc_draw,xim+77,yim+y,xim+88,yim+y);
    XDrawLine(display,ik_window,gc_draw,xim+77,yim+1+y,xim+88,yim+1+y);
  }

  for(y=0;y<=640;y+=16*5){
    XDrawLine(display,ik_window,gc_draw,xkm-1,ykm+y,xkm-13,ykm+y);
    XDrawLine(display,ik_window,gc_draw,xim+77,yim+y,xim+88,yim+y);
  }

  for(y=0;y<=640;y+=8*5){
    XDrawLine(display,ik_window,gc_draw,xkm-1,ykm+y,xkm-10,ykm+y);
    XDrawLine(display,ik_window,gc_draw,xim+77,yim+y,xim+85,yim+y);
  }

  for(y=0;y<=640;y+=4*5){
    XDrawLine(display,ik_window,gc_draw,xkm-1,ykm+y,xkm-7,ykm+y);
```

```
    XDrawLine(display,ik_window,gc_draw,xim+77,yim+y,xim+82,yim+y);
  }

  draw_keys_matrix();
  draw_inversion_matrix();

  XFlush(display);
}

void process_quit()
{
  exit(0);
}

void process_clear()
{
  clear_stat();
  clear_keys();
  clear_inversion();
  XClearWindow(display,ik_window);
  draw_matrices();
}

void inversion_0()           /* simple path modifikation  */
{
  int i;

  clear_inversion();
  for(i=0;i<15;i++)
    inversion[i][path[i]]=1;
}

void inversion_1()           /* diagonal path modifikation  */
{
  int i,s;

  clear_inversion();
  for(i=0;i<15;i++)
    for(s=0;s<15;s++)
      {
inversion[s][path[i]]=1;
inversion[s][(path[i]+i-s+128)%128]=1;
inversion[s][(path[i]-i+s+128)%128]=1;
      }
}

void inversion_2()           /* invert leftright-neighbours of path */
```

```
{
  int i;

  clear_inversion();

  for(i=0;i<15;i++)
      {
inversion[(i+14)%15][path[i]]=1;
inversion[(i+1)%15][path[i]]=1;
      }
}

void inversion_3()           /* inverty all (eight) path-neighbours  */
{
  int i,s,z;

  clear_inversion();

    for(i=0;i<15;i++)
      for(s=0;s<3;s++)
for(z=0;z<3;z++)
    inversion[(i+14+s)%15][(path[i]+127+z)%128]=1;
}

void inversion_4()          /* Matrix inversion as in  game of life  */
                            /* makes no sense in kryptology          */
{
  int i,j,sum;

  for(i=0;i<15;i++)
    for(j=0;j<128;j++)
      {
sum = keys[(i+14)%15][(j+127)%128] + keys[(i+14)%15][(j)%128]
            + keys[(i+14)%15][(j+1)%128]   + keys[(i)][(j+127)%128]
            + keys[(i)%15][(j+1)%128     ] + keys[(i+1)%15][(j+127)%128]
            + keys[(i+1)%15][(j)%128]      + keys[(i+1)%15][(j+1)%128] ;
if(sum==2)
  inversion[i][j]=0;
else
  if(sum==3)
    inversion[i][j]=1-keys[i][j];
  else
    inversion[i][j]=keys[i][j];;
      }
  for(i=0;i<15;i++)
    inversion[i][path[i]]=1- inversion[i][path[i]];
}
```

```
void inversion_5()          /*  Modified game of life  */
{
  int i,j,sum;

  for(i=0;i<15;i++)
    for(j=0;j<128;j++)
      {
sum = keys[(i+14)%15][(j+127)%128] + keys[(i+14)%15][(j)%128]
            + keys[(i+14)%15][(j+1)%128]   + keys[(i)][(j+127)%128]
            + keys[(i)%15][(j+1)%128     ] + keys[(i+1)%15][(j+127)%128]
            + keys[(i+1)%15][(j)%128]      + keys[(i+1)%15][(j+1)%128] ;
switch(sum)
  {
  case 0: inversion[i][j]=1; break;
  case 1: inversion[i][j]=0; break;
  case 2: inversion[i][j]=1; break;
  case 3: inversion[i][j]=keys[i][j]; break;
  case 4: inversion[i][j]=1-keys[i][j]; break;
  case 5: inversion[i][j]=keys[i][j]; break;
  case 6: inversion[i][j]=0; break;
  case 7: inversion[i][j]=1; break;
  case 8: inversion[i][j]=0; break;
  }
      }
}

void inversion_6()    /* simple path modifikation with cyclic row-shift  */
{
  int i,j;

  clear_inversion();
  for(i=0;i<15;i++)
    inversion[i][path[i]]=1;
  for(i=0;i<15;i++)
    for(j=0;j<128;j++)
      inversion[i][j] = (inversion[i][j] + keys[i][j] + keys[i][(j+127)%128]) %2 ;
}

void modify_keys()    /* adds inversion matrix to key matrix */
{
  int i,j;
  (*inversion_functions[inv])();

  for(i=0;i<15;i++)
    for(j=0;j<128;j++)
      keys[i][j]=(keys[i][j]+inversion[i][j])%2;
```

```
  if(m==1)
    {
      draw_keys_matrix();
      draw_inversion_matrix();
    }
}

void draw_buttons()   /* Draws buttons in  menu-window */
{ char buf[10];

  XClearWindow(display,buttons_window);

  XDrawRectangle(display,buttons_window,gc_buttons_draw,5,5,50,50);
  XDrawString(display,buttons_window,gc_buttons_draw,14,27,"Manual",6);
  XDrawString(display,buttons_window,gc_buttons_draw,17,42,"Input",5);

  XDrawRectangle(display,buttons_window,gc_buttons_draw,60,5,50,50);
  XDrawString(display,buttons_window,gc_buttons_draw,69,27,"Random",6);
  XDrawString(display,buttons_window,gc_buttons_draw,72,42,"Input",5);

  if(randommode)
    {XDrawRectangle(display,buttons_window,gc_buttons_draw,62,7,46,46);}
  else
    {XDrawRectangle(display,buttons_window,gc_buttons_draw,7,7,46,46);}

  XDrawRectangle(display,buttons_window,gc_buttons_draw,5,60,50,50);
  XDrawString(display,buttons_window,gc_buttons_draw,20,82,"Save",4);
  XDrawString(display,buttons_window,gc_buttons_draw,23,97,"Key",3);

  XDrawRectangle(display,buttons_window,gc_buttons_draw,60,60,50,50);
  XDrawString(display,buttons_window,gc_buttons_draw,75,82,"Save",4);
  XDrawString(display,buttons_window,gc_buttons_draw,65,97,"Invert.",7);

  XDrawRectangle(display,buttons_window,gc_buttons_draw,5,115,50,50);
  XDrawString(display,buttons_window,gc_buttons_draw,20,145,"Step",4);

  XDrawRectangle(display,buttons_window,gc_buttons_draw,60,115,50,50);
  XDrawString(display,buttons_window,gc_buttons_draw,75,137,"Step",4);
  XDrawString(display,buttons_window,gc_buttons_draw,84,152,"8",1);

  XDrawRectangle(display,buttons_window,gc_buttons_draw,5,170,50,50);
  XDrawString(display,buttons_window,gc_buttons_draw,20,192,"Step",4);
  XDrawString(display,buttons_window,gc_buttons_draw,26,207,"64",2);

  XDrawRectangle(display,buttons_window,gc_buttons_draw,60,170,50,50);
  XDrawString(display,buttons_window,gc_buttons_draw,75,192,"Step",4);
  XDrawString(display,buttons_window,gc_buttons_draw,75,207,"8192",4);
```

```
  XDrawRectangle(display,buttons_window,gc_buttons_draw,5,225,50,50);
  XDrawString(display,buttons_window,gc_buttons_draw,20,247,"Save",4);
  XDrawString(display,buttons_window,gc_buttons_draw,17,262,"Stat.",5);

  XDrawRectangle(display,buttons_window,gc_buttons_draw,60,225,50,50);
  XDrawString(display,buttons_window,gc_buttons_draw,72,247,"Clear",5);
  XDrawString(display,buttons_window,gc_buttons_draw,72,262,"Stat.",5);

  XDrawRectangle(display,buttons_window,gc_buttons_draw,5,280,50,50);
  XDrawRectangle(display,buttons_window,gc_buttons_draw,7,282,46,46);
  XDrawString(display,buttons_window,gc_buttons_draw,23,302,"Net",3);
  sprintf(buf,"Nr.%.1d",net);
  XDrawString(display,buttons_window,gc_buttons_draw,20,317,buf,strlen(buf));

  XDrawRectangle(display,buttons_window,gc_buttons_draw,60,280,50,50);
  XDrawRectangle(display,buttons_window,gc_buttons_draw,62,282,46,46);
  XDrawString(display,buttons_window,gc_buttons_draw,65,302,"Invert.",7);
  sprintf(buf,"Nr.%.1d",inv);
  XDrawString(display,buttons_window,gc_buttons_draw,75,317,buf,strlen(buf));

  XDrawRectangle(display,buttons_window,gc_buttons_draw,5,335,50,50);
  XDrawString(display,buttons_window,gc_buttons_draw,20,357,"Show",4);
  XDrawString(display,buttons_window,gc_buttons_draw,17,372,"Matr.",5);

  if(m==1)
    {XDrawRectangle(display,buttons_window,gc_buttons_draw,7,337,46,46);}

  XDrawRectangle(display,buttons_window,gc_buttons_draw,5,390,50,50);
  XDrawString(display,buttons_window,gc_buttons_draw,17,422,"Clear",5);

  XDrawRectangle(display,buttons_window,gc_buttons_draw,60,390,50,50);
  XDrawString(display,buttons_window,gc_buttons_draw,75,422,"Quit",4);

  XFlush(display);
}

static int encrypt(int plaintext);

void process_step()
{
  int plaintext, ciphertext;
  if(randommode){
    plaintext=random()&0xff;
  printf("Plaintext : %d\n",plaintext);
  } else
    do {
```

```
      printf("Plaintext : ");
      scanf("%d",&plaintext);
    } while ( (plaintext<0) || (plaintext>255) );
  ciphertext=encrypt(plaintext);
  modify_keys();
  printf("Ciphertext: %d\n",ciphertext);
}

void process_step1()
{
  XDrawRectangle(display,buttons_window,gc_buttons_draw,7,117,46,46);
  XFlush(display);
  process_step();
}

void process_step8()
{
  int i;
  XDrawRectangle(display,buttons_window,gc_buttons_draw,62,117,46,46);
  for(i=0;i<8;i++){
    if(XCheckWindowEvent(display,ik_window,ButtonPressMask,&xevent))
      break;
    process_step();
  }
}

void process_step64()
{
  int i;
  XDrawRectangle(display,buttons_window,gc_buttons_draw,7,172,46,46);
  for(i=0;i<64;i++){
    if(XCheckWindowEvent(display,ik_window,ButtonPressMask,&xevent))
      break;
    process_step();
  }
}

void process_step8192()
{
  int i;
  XDrawRectangle(display,buttons_window,gc_buttons_draw,62,172,46,46);
  for(i=0;i<8192;i++){
    if(XCheckWindowEvent(display,ik_window,ButtonPressMask,&xevent))
      break;
    process_step();
  }
}
```

```
void process_save_k()      /* saves all black pixels (i,j) of key matrix */
                           /* for LaTex-picture environment              */
                           /* format: \put(i,127-j){\rule{1.5mm}{1.5mm}} */
                           /* not very efficently !!!                    */
{
  int i,j;
  FILE *fopen(),*fp;
  XDrawRectangle(display,buttons_window,gc_buttons_draw,7,62,46,46);
  XFlush(display);
  fp = fopen("matrix","w");
  fprintf(fp," This is a keymatrix of invertion %d, with net %d  \n",inv,net);
  for(i=0;i<15;i++)
    for(j=0;j<128;j++)
      if(keys[i][j])
{
  fprintf(fp,"\\put(%d,%d){\\rule{1.5mm}{1.5mm}}\n",i,127-j);
}
  fclose(fp);
}

void process_save_i()       /* saves all black pixels of inversion matrix */
{
  int i,j;
  FILE *fopen(),*fp;
  XDrawRectangle(display,buttons_window,gc_buttons_draw,62,62,46,46);
  XFlush(display);
  fp = fopen("matrix","w");
  fprintf(fp," This is a inversionmatrix of inv %d, with net %d  \n",inv,net);
  for(i=0;i<15;i++)
    for(j=0;j<128;j++)
      if(inversion[i][j])
{
  fprintf(fp,"\\put(%d,%d){\\rule{1.5mm}{1.5mm}}\n",i,127-j);
}
  fclose(fp);
}

void process_save_stat()       /* saves the statistic in Latex-tabular c column */
{
  int i,j,c;
  FILE *fopen(),*fp;
  XDrawRectangle(display,buttons_window,gc_buttons_draw,7,227,46,46);
  XFlush(display);
  fp = fopen("statistic","w");
  fprintf(fp,"%% This is a statistic of the output with inversion %d  \n",inv);
  c=8;
```

```
  for(i=0;i<(256/c);i++)
    {
      for(j=0;j<c;j++)
{
  fprintf(fp,"%d & %d ",i+256/c*j,statistic[i+256/c*j]);
  if(j<7)
    fprintf(fp," &");
}
      fprintf(fp,"\\\\\n");
    }
  for(i=0;i<1000;i++)
    stat2[i]==0;
  for(i=0;i<256;i++)
    if(statistic[i]<1000)
      stat2[statistic[i]]++;
  for(i=0;i<1000;i++)
    fprintf(fp,"%% %d :  %d \n",i,stat2[i]);
  fclose(fp);
}

void process_choose_n()    /* select benes-net or benes-equiv. net */
{
  net=1-net;
   if(net)
     {
       benes_equiv_perms();
     }
   else
     {
       benes_perms();
     }
}

void process_choose_i()    /* select inversion-function */
{
  inv=(inv+1)%7;
}

void process_display_m()    /* m=1 z display matrices */
{
  m=1-m;
  if(m==1)
    {
      draw_keys_matrix();
      draw_inversion_matrix();
    }
}
```

```
void process_manual()
{ randommode=0; }

void process_random()
{ randommode=1; }

void process_clear_stat()
{ clear_stat(); }

void process_void()
{}

                           /* menu */
static void (*dispatch_table[2][8])()={
  { process_manual, process_save_k,   process_step1,  process_step64,
      process_save_stat, process_choose_n, process_display_m,   process_clear},
  { process_random, process_save_i,   process_step8,  process_step8192,
      process_clear_stat, process_choose_i, process_void,   process_quit}};

int encrypt(int p)
{
  int i;
  for(i=0; i<15; i++){
    path[i]=p/2;
    if(keys[i][p/2]){
      p=perms[i][2*(p/2)+(1-p%2)];
    } else {
      p=perms[i][p];
    }
  };
  statistic[p]++;
  return p;
}

void flip_key_row(int row)
{
  int x;
  for(x=1;x<16;x++)
    flip_key_pixel(row,x);
}

void flip_key_column(int column)
{
  int y;
  for(y=0;y<128;y++)
    flip_key_pixel(y,column);
```

```
}

void dispatcher()
{
  int x,y;

  XNextEvent(display, &xevent);
  if( xevent.xbutton.window==buttons_window) {
    if(xevent.type == Expose) {
      draw_buttons();
    } else if(xevent.type == ButtonPress) {
      if((xevent.xbutton.x-5)%55 > 50 ||
 (xevent.xbutton.y-5)%55 > 50) return;
      x=(xevent.xbutton.x-5)/55;
      y=(xevent.xbutton.y-5)/55;
      if( (x<0) || (x>1) || (y<0) || (y>7) ) return;
      (*dispatch_table[x][y])();
      draw_buttons();
    }
  } else {
    if(xevent.type == ButtonPress) {
      x=(xevent.xbutton.x-xkm)/5+1;
      y=(xevent.xbutton.y-ykm)/5;
      if(xevent.xbutton.button == 1)
flip_key_pixel(y,x);
      else if(xevent.xbutton.button == 2)
flip_key_row(y);
      else if(xevent.xbutton.button == 3)
flip_key_column(x);
    } else if(xevent.type == Expose) {
      draw_matrices();
    }
  }
}

main()
{
  xinit();
  process_clear();
  process_choose_n();
  draw_buttons();
  while(1)
    dispatcher();
}
```

# C Nomenklatur

| | |
|---|---|
| $Z, S$ | Zeilen, Spalten der Matrix bzw. des Netzwerks |
| $P_i$ | Permutation der $i$.-ten Tauscherspalte |
| $V_i$ | Vernetzungspermutation zwischen $i$.-ter und $i+1$.-ter Tauscherspalte |
| $K \in \mathcal{K}$ | Schlüsselmatrix, Schlüsselraum |
| $K_t$ | Schlüsselmatrix nach $t$ Verschlüsselungsschritten |
| $k_t(z, s)$ | Element der Schlüsselmatrix |
| $\mathcal{V}$ | Vernetzung ( $\mathcal{V} = (V_0, V_1, \ldots V_S)$) |
| $n$ | Blockbreite der Chiffre |
| $A$ | Ein- Ausgabealphabet ( $A = \{0,1\}^n$) |
| $\delta(K_t, m_t)$ | Folgezustandsfunktion |
| $l$ | Länge des Klartextes und des Chiffrates (in $n$-bit Blöcken) |
| $M$ | Klartext ($M = m_0, m_1, \ldots m_{l-1}$) |
| $C$ | Schlüsseltext ($C = c_0, c_1, \ldots c_{l-1}$) |
| $e_{K_t}(m_t)$ | Blockverschlüsselungsfunktion |
| $d_{K_t}(c_t)$ | Blockentschlüsselungsfunktion |
| $E_K(M)$ | Entschlüsselungsfunktion |
| $D_K(M)$ | Verschlüsselungsfunktion |
| $H(M)$ | Hashfunktion |
| $h$ | Länge des Hashwertes |
| $B_n$ | Beneš-Netzwerk mit $2^n$ Ein- und Ausgängen |
| $\mathrm{SINC}_n$ | Selbstmodifizierendes Netzwerk mit $2^n$ Ein- und Ausgängen |
| $\mathcal{V}_{Benes}$ | Beneš-Vernetzung ($\mathcal{V}_{Benes} = (id, Q_1, Q_2, \ldots Q_{S-1}, id)$) |
| $IP = V_0$ | Eingangspermutation des Netzwerks |
| $\mathcal{P}$ | Untergruppe der $S_{2^n}$, Menge der Permutationen die nur Zyklen der Form $(i)$ und $(2i, 2i+1)$ enthalten |
| $\mathcal{T}$ | Untergruppe der $S_{2^n}$, Menge der Permutationen die paarweise Elemente $(2i, 2i+1)$ permutieren |

# Literatur

[Bene65] V.E. Beneš: *Mathematical Theory of Connecting Networks and Telefon Traffic*, Academic Press, New York, 1965.

[Akl85] Selim G. Akl: *Parallel Sorting Algorithms*, Academic Press, Orlando, 1985.

[Hors85] Patrick Horster: *Kryptologie*, Reihe Informatik, Bibliographisches Institut, Mannheim, 1985.

[SePi89] J. Seberry, J. Pieprzyk: *Cryptography: An Introduction to Computer Science*, Prentice Hall, 1989.

[Port91] Portz, Michael: *On the Use of Interconnection Networks in Cryptography*, Advances in Cryptology- EUROCRYPT'91, Lecture Notes in Computer Science 547, 302-315, 1991.

[HoNP91] Patrick Horster, Armin Nückel, Michael Portz: *The design of a VLSI cryptochip using intelligent design environment.* Europäisches Institut für Systemsicherheit, E.I.S.S. Report 91/14, 1991.

[Port92] Portz, Michael: *Verbindungsnetzwerke als Permutationsgeneratoren im kryptologischen Kontext.* Dissertation der Mathematisch-Naturwissenschaftlichen Fakultät der Technischen Hochschule Aachen, 1992.

[Vick92] Vickus, Martin: *Verbindungsnetzwerke für kryptologische Anwendungen – Entwurf, Implementation und Analyse.* Diplomarbeit am Lehrstuhl für angewandte Mathematik, RWTH Aachen, 1992.

[Hors93] Horster, Patrick: *Selbstmodifizierende Verbindungsnetzwerke – Ein neuartiges Konzept zur Realisierung kryptographischer Basisfunktionen.* Europäisches Institut für Systemsicherheit, Proceedings zur 3. GI-Fachtagung Verläßliche Informationssysteme (VIS'93), 11. - 13 Mai 1993, München, erscheint im Vieweg-Verlag, außerdem E.I.S.S. Report 93/3, 1993.