

FBDD-basierte Kryptanalyse des A5/1 Schlüsselstromgenerators

Diplomarbeit

vorgelegt am

Lehrstuhl für Theoretische Informatik
Prof. Dr. Matthias Krause
Universität Mannheim

im

Juni 2004

von

Dirk Stegemann

Inhaltsverzeichnis

1	Einleitung	1
2	Flusschiffren und LFSR-basierte Schlüsselstromgeneratoren	2
2.1	Allgemeine Funktionsweise von Flusschiffren	2
2.2	Kryptographische Sicherheit von Flusschiffren	3
2.3	Linear rückgekoppelte Schieberegister (LFSR)	3
2.4	Lineare Bitstromgeneratoren	6
2.5	LFSR-basierte Schlüsselstromgeneratoren	8
2.5.1	<i>best case</i> Kompressionsrate γ	8
2.5.2	Informationsrate α	9
3	Kryptanalyse LFSR-basierter Schlüsselstromgeneratoren	12
3.1	Angriffsmodell	12
3.2	Allgemeiner Kryptanalyseansatz	13
3.3	Bestimmung des Initialzustands x	13
3.4	Bestimmung der Eindeutigkeitsgrenze m^*	15
4	Repräsentation Boolescher Funktionen mit FBDDs	16
4.1	Operationen auf Booleschen Funktionen	16
4.2	Binäre Entscheidungsgraphen (BDDs)	17
4.3	Geordnete binäre Entscheidungsgraphen (OBDDs)	18

4.4	Freie binäre Entscheidungsgraphen (FBDDs)	19
4.5	Operationen auf FBDDs	20
4.5.1	Minimierung	20
4.5.2	m -äre Synthese	21
4.5.3	SAT-ENUM	23
5	Kryptanalyse mit Hilfe von FBDDs	25
5.1	Repräsentation der Sprachen U_m , W_m und T_m durch FBDDs	25
5.2	Bestimmung von $ R_m $ und $ S_m $	26
5.3	Bestimmung von $ P_m $	28
5.4	Gesamtresultat	29
6	Der Algorithmus A5/1	30
6.1	Verschlüsselung der Sprachdaten	30
6.2	Aufbau des A5/1 Schlüsselstromgenerators	31
6.3	Arbeitsweise des A5/1 Schlüsselstromgenerators	32
7	Kryptanalyse des A5/1 Schlüsselstromgenerators	33
7.1	Simulation des Algorithmus A5/1 durch einen LFSR-basierten Schlüsselstromgenerator	33
7.1.1	Vorbemerkungen	33
7.1.2	Konstruktion einer Simulation mit <i>read-once</i> Kompressionsfunktion	34
7.2	Konstruktion des Steuerungsgraphen G_m^C	35
7.3	Konstruktion des G_m^C -FBDDs Q_m	39
7.4	Konstruktion der G_m^C -FBDDs R_m und S_m	42
7.5	Statistische Eigenschaften des modifizierten A5/1 Schlüsselstromgenerators	44
7.6	Bestimmung der Informationsrate α des modifizierten A5/1 Schlüsselstromgenerators	48
7.7	Gesamtresultat für den A5/1 Schlüsselstromgenerator	50
8	Implementation des FBDD-Pakets	52

8.1	Designkonzepte von OBDD-Implementationen	52
8.1.1	<i>shared</i> OBDDs und <i>Manager</i>	52
8.1.2	<i>unique-table</i>	53
8.1.3	<i>computed-table</i>	53
8.1.4	komplementierte Kanten	54
8.1.5	Implementation von Knoten und Referenzen	55
8.1.6	Speicherverwaltung	55
8.2	Auswahl des OBDD-Basispakets	56
8.2.1	Anforderungen an OBDD-Pakete	57
8.2.2	Frei verfügbare OBDD-Pakete	57
8.3	Design und Implementation des FBDD-Pakets	58
8.3.1	BDD-Adapter	59
8.3.2	Schnittstelle des FBDD-Pakets	62
9	Implementation der FBDDs für die Kryptanalyse	65
9.1	Berechnung des Steuerungsgraphen G_m^C und des G_m^C -FBDDs Q_m	65
9.2	Berechnung des G_m^C -FBDDs S_m	66
9.3	Implementation des Kryptanalysealgorithmus	66
10	Empirische Resultate	71
10.1	Entwicklung der maximalen Größe von P_m in Abhängigkeit der Schlüssellänge	71
10.1.1	Datengenerierung	71
10.1.2	Ergebnisse	72
10.2	$ P_m $ in Abhängigkeit der Iterationen	72
10.2.1	Datengenerierung	73
10.2.2	Ergebnisse	73
10.3	Verwendete Hardware und Systemumgebung	74

<i>INHALTSVERZEICHNIS</i>	5
11 Zusammenfassung und Ausblick	76
Literaturverzeichnis	77
A Änderungen am Quellcode von CUDD	79
B Ehrenwörtliche Erklärung	81

Kapitel 1

Einleitung

Die vorliegende Arbeit beschäftigt sich mit einem kryptographischen Angriff auf den im GSM-Mobilfunknetz zur Verschlüsselung der Sprachdaten verwendeten A5/1 Schlüsselstromgenerator. Wie bei kryptographischen Angriffen allgemein geht es auch hier darum, in den Besitz einer Geheiminformation zu kommen, auf deren Basis zwei Kommunikationspartner ihren Datenverkehr - in diesem Fall ein Mobilfunktelefonat - verschlüsseln. Mit Hilfe der Geheiminformation kann der Angreifer den Datenverkehr dann zumindest abhören und in einigen Fällen sogar unbemerkt manipulieren.

Da die Menge der möglichen Geheiminformationen, der Schlüsselraum, aus Berechenbarkeitsgründen endlich ist, hat ein Angreifer immer die Möglichkeit, systematisch alle Schlüssel auszuprobieren, bis er den korrekten Schlüssel gefunden hat. Um diesen trivialen Angriff zu verhindern, wählt man die Schlüsselräume so groß, dass das Betrachten aller möglichen Schlüssel mit sinnvollem Berechnungsaufwand nicht mehr durchführbar ist. An dieser Stelle kommen nichttriviale Angriffstechniken ins Spiel, die in der Lage sind, Teile des Schlüsselraums auszuschließen und damit den Suchraum zu verkleinern. [Kra02] hat ein solches nichttriviales Verfahren im Jahre 2002 veröffentlicht, das im Rahmen dieser Arbeit implementiert werden soll, einerseits um die theoretischen Resultate empirisch zu stützen, aber auch um festzustellen, wie gefährlich dieser Angriff der Chiffre in der Praxis tatsächlich werden kann.

Die ersten vier Kapitel der Arbeit beschäftigen sich zunächst unabhängig vom A5/1 Schlüsselstromgenerator mit den theoretischen Grundlagen der in [Kra02] entwickelten Kryptanalyse. In Kapitel 2 betrachten wir einige grundlegende Eigenschaften und Sicherheitsparameter von Flusschiffren, die wie der A5/1 Generator auf linear rückgekoppelten Schieberegistern (LFSR) basieren. Anschließend stellen wir den Angriff in abstrakter Weise als Manipulation von Booleschen Funktionen vor und präsentieren *Free Binary Decision Diagrams* (FBDDs) als eine effiziente Möglichkeit, diese Booleschen Funktionen zu verwalten. Mit FBDDs als Hilfsmitteln sind wir in der Lage, in Kapitel 5 ein theoretisches Laufzeitresultat für die Kryptanalyse herzuleiten.

Die folgenden zwei Kapitel konkretisieren den Angriff für den A5/1 Generator und ermitteln eine theoretische Laufzeitschranke für den Angriff dieser Chiffre.

Da keine frei verfügbaren Programmpakete für FBDDs existieren, gleichzeitig aber zahlreiche Pakete für die mit FBDDs verwandten *Ordered Binary Decision Diagrams* (OBDDs) erhältlich sind, zeigen wir in Kapitel 8 auf, wie man ein solches OBDD-Paket benutzen kann, um eine effiziente Programmbibliothek für die Verwaltung von FBDDs zu erstellen. Mit dieser Bibliothek sind wir schließlich in der Lage, die Kryptanalyse des A5/1 Generators praktisch durchzuführen und stellen im letzten Kapitel den zu Beginn formulierten theoretischen Resultaten einige empirische Ergebnisse gegenüber.

Kapitel 2

Flusschiffren und LFSR-basierte Schlüsselstromgeneratoren

Als Grundlage für die Darstellung der FBDD-basierten Kryptanalyse wollen wir in diesem einführenden Kapitel zunächst die Zielobjekte unseres Angriffs untersuchen und die wesentlichen Eigenschaften von Flusschiffren und LFSR-basierten Schlüsselstromgeneratoren zusammentragen.

2.1 Allgemeine Funktionsweise von Flusschiffren

Das Hauptanwendungsgebiet für Flusschiffren ist das folgende Kommunikationsszenario:

Eine als Bitstrom $p = (p_0, \dots, p_{|p|-1}) \in \{0, 1\}^*$ gegebene Klartext-Nachricht soll in Echtzeit verschlüsselt und über einen unsicheren Kanal zu einem Empfänger übermittelt werden. Bei den zu übermittelnden Daten kann es sich, etwa wie im Anwendungskontext des A5/1 Schlüsselstromgenerators, zum Beispiel um digitalisierte Sprachdaten handeln, die von einem Mobiltelefon über die Luftschnittstelle abhörgeschützt zur Basisstation des Netzbetreibers oder umgekehrt übertragen werden sollen.

Die Nachricht p wird verschlüsselt, indem ein geheimer, d.h. nur dem Sender und dem legalen Empfänger bekannter Schlüsselbitstrom

$$y = (y_0, \dots, y_{|p|-1}) \in \{0, 1\}^{|p|}$$

bitweise zum Klartextstrom addiert wird. Die verschlüsselte Nachricht (das Kryptogramm) ergibt sich also als

$$e = (e_0, \dots, e_{|p|-1}) \in \{0, 1\}^{|p|} \text{ mit } e_i := p_i \oplus y_i \text{ für alle } i \in \{0, \dots, |p| - 1\}$$

wobei die Operation \oplus die binäre Addition bezeichnet, d.h. $a \oplus b := a + b \pmod 2$ für alle $a, b \in \{0, 1\}$.

Für alle $a, b \in \{0, 1\}$ gilt $(a \oplus b) \oplus b = a$, wie man mit Hilfe einer Wertetabelle leicht nachprüft. Der legale Empfänger kann daher das erhaltene Kryptogramm analog zur Verschlüsselung durch bitweise Addition des Schlüsselstroms wieder entschlüsseln und erhält somit den Klartext als

$$\underbrace{(p_0 \oplus y_0 \oplus y_0, \dots, p_{|p|-1} \oplus y_{|p|-1} \oplus y_{|p|-1})}_{e_0 \quad e_{|p|-1}} = (p_0, \dots, p_{|p|-1}) = p$$

2.2 Kryptographische Sicherheit von Flusschiffren

Wie durch das Kerckhoffsche Prinzip gefordert und in der Kryptanalyse allgemein üblich, gehen wir davon aus, dass ein Angreifer, der aus einem gegebenen Chiffretext e und ggf. weiteren Zusatzinformationen den Klartext p bzw. den Schlüsselbitstrom y rekonstruieren will, bis auf den Schlüsselbitstrom alle Details des Verschlüsselungsalgorithmus kennt. Obwohl der Angreifer den einem konkreten Chiffretext e zugrunde liegenden Schlüsselstrom y nicht genau kennt, weiß er immerhin, wie y im Prinzip aufgebaut ist.

Für den Fall, dass die einzelnen Bits des Schlüsselstroms echt zufällig sind, d.h. unabhängig aus $\{0, 1\}$ mit gleicher Wahrscheinlichkeit für 0 bzw. 1 gezogen werden, hat [Sha49] gezeigt, dass ein Angreifer, dem ein Kryptogramm e in die Hände fällt, keine Möglichkeit hat, aus e Informationen über den Klartext p zu gewinnen. Selbst wenn er einige Klartext-Chiffretext Paare $(p_{i_1}, e_{i_1}), \dots, (p_{i_s}, e_{i_s})$ und damit auch die Schlüsselbits y_{i_1}, \dots, y_{i_s} kennt, kann er daraus keine Informationen über Klartextbits p_i mit $i \notin \{i_1, \dots, i_s\}$ ableiten.

Dieser als *One-Time-Pad* bezeichneter Schlüsselbitstrom führt zwar zu maximaler kryptographischer Sicherheit, aber Sender und Empfänger müssen sich vor der Übermittlung der eigentlichen Nachricht in geeigneter Weise auf einen Schlüsselbitstrom verständigen, der genauso lang ist wie die Nachricht selbst. Dies ist für viele Anwendungen wie zum Beispiel die mobile Kommunikation offensichtlich nicht praktikabel. Man verwendet daher statt des *One-Time-Pads* in den meisten Fällen sogenannte *Pseudo-One-Time-Pad* Schlüsselstromgeneratoren, die basierend auf einem geheimen Schlüssel fixierter Länge, den Sender und Empfänger zu Beginn der Datenübermittlung aushandeln, deterministisch einen beliebig langen pseudozufälligen Schlüsselstrom erzeugen können. Wichtigstes sicherheitsrelevantes Designkriterium solcher Schlüsselstromgeneratoren ist eine möglichst gute Approximation des *One-Time-Pads*, d.h. der erzeugte Bitstrom sollte einem echt zufällig erzeugten Bitstrom möglichst ähnlich sehen. Wie man diese Ähnlichkeit definieren und messen könnte, wird in der Literatur ausführlich diskutiert. In diesem Kapitel werden einige elementare Kriterien und deren Auswirkungen auf das Design von Schlüsselstromgeneratoren vorgestellt.

Trotz eines weitgehend zufälligen Erscheinungsbilds besitzen die deterministisch erzeugten Schlüsselströme der *Pseudo-One-Time-Pad* Generatoren immer eine mehr oder weniger deutliche innere Struktur, die sie von echt zufälligen Bitströmen unterscheidet und damit die theoretische Grundlage kryptographischer Angriffe bildet.

Viele in der Praxis verwendete Schlüsselstromgeneratoren, darunter auch der A5/1-Schlüsselstromgenerator, verwenden linear rückgekoppelte Schieberegister (LFSR) als Grundbausteine für die Erzeugung eines pseudozufälligen internen Bitstroms, der anschließend mit Hilfe einer Kompressionsfunktion zu einem Ausgabeschlüsselstrom verdichtet wird. Wir wollen nun den Aufbau solcher LFSR-basierter Schlüsselstromgeneratoren genauer betrachten.

2.3 Linear rückgekoppelte Schieberegister (LFSR)

Definition 1. Ein linear rückgekoppeltes Schieberegister (*Linear Feedback Shift Register*, kurz *LFSR*) L der Länge n mit einem Koeffizientenvektor $c = (c_1, \dots, c_n) \in \{0, 1\}^n$ besteht aus n binären Speicherzellen q_1, \dots, q_n , die durch einen Rückkopplungskanal verbunden sind. Das LFSR wird periodisch getaktet, wobei in jedem Takt $t > 0$ der Inhalt der Speicherzelle q_1 ausgegeben und für $i \in \{2, \dots, n\}$ der Inhalt von Zelle q_i in die Zelle q_{i-1} kopiert wird. Der neue Wert der Speicherzelle q_n wird mit Hilfe des Rückkopplungskanals aus den aktuellen Inhalten der Speicherzellen berechnet als

$$q_n^{neu} := c_1 q_1 \oplus c_2 q_2 \oplus \dots \oplus c_n q_n$$

Zu Beginn der Berechnung, d.h. in Takt $t = 0$, werden die Speicherzellen q_1, \dots, q_n mit einer Anfangsbelegung $x = (x_0, \dots, x_{n-1}) \in \{0, 1\}^n$ initialisiert.

Abbildung 2.1 illustriert die Funktionsweise von linear rückgekoppelten Schieberegistern anhand eines LFSR der Länge $n = 4$ mit der Anfangsbelegung $x = (1, 0, 1, 1)$ und dem Koeffizientenvektor $c = (1, 1, 0, 1)$.

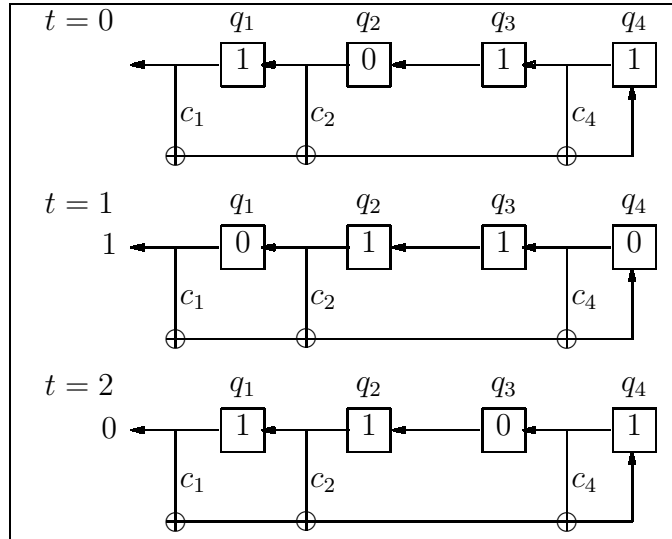


Abbildung 2.1: Berechnungsverhalten von LFSR

Für den Ausgabebitstrom eines LFSRs ergibt sich aus der Definition unmittelbar die folgende Rekurrenzdarstellung:

Beobachtung 2. Ein LFSR L der Länge n erzeugt in Abhängigkeit eines Koeffizientenvektors $c = (c_1, \dots, c_n) \in \{0, 1\}^n$ und einer Anfangsbelegung $x = (x_0, \dots, x_{n-1}) \in \{0, 1\}^n$ den Bitstrom

$$L(x) = L_0(x), L_1(x), \dots, L_i(x), \dots$$

$$\text{mit } L_i(x) := \begin{cases} x_i & \text{falls } 0 \leq i \leq n-1 \\ c_1 L_{i-n}(x) \oplus c_2 L_{i-n+1}(x) \oplus \dots \oplus c_n L_{i-1}(x) & \text{falls } i > n-1 \end{cases}$$

d.h. es gilt $L_i(x) = \bigoplus_{k=0}^{n-1} c_{k+1} \cdot L_{i-n+k}(x)$ für $i > n-1$.

Diese Schreibweise verdeutlicht, dass die ersten n Bits $L_0(x), \dots, L_{n-1}(x)$ des Bitstroms $L(x)$ den Bits der Anfangsbelegung entsprechen und alle weiteren Bits $L_i(x)$, $i > n-1$ durch Linearkombination der n Vorgängerbits $L_{i-n}(x), \dots, L_{i-1}(x)$ gebildet werden.

Gleichzeitig hängen alle Bits $L_i(x)$, $i > n-1$, linear von den Bits der Anfangsbelegung ab, wie die folgende Beobachtung zeigt:

Beobachtung 3. Die Bits $L_i(x)$ des durch ein LFSR L erzeugten Bitstroms $L(x)$ können als Linearkombination von $x = (x_0, \dots, x_{n-1})$ dargestellt werden. Es gilt für $i \geq 0$

$$L_i(x) = \bigoplus_{j=0}^{n-1} L_{j,i} \cdot x_j$$

mit den Koeffizienten

$$L_{j,i} = \begin{cases} 1 & \text{falls } i \leq n-1 \text{ und } i = j \\ 0 & \text{falls } i \leq n-1 \text{ und } i \neq j \\ \bigoplus_{k=0}^{n-1} c_{k+1} \cdot L_{j,i-n+k} & \text{sonst} \end{cases}$$

Beweis. Wir beweisen die behauptete Aussage mittels vollständiger Induktion über i .

Für $i \leq n-1$ gilt

$$L_i(x) = \bigoplus_{j=0}^{n-1} L_{j,i} \cdot x_j = L_{i,i} \cdot x_i = x_i$$

d.h. die Behauptung gilt für $i \leq n-1$.

Für $i \geq n$ erhalten wir

$$\begin{aligned} L_i(x) &= \bigoplus_{k=0}^{n-1} c_{k+1} \cdot L_{j,i-n+k}(x) \\ &= \bigoplus_{k=0}^{n-1} c_{k+1} \cdot \left(\bigoplus_{j=0}^{n-1} L_{j,i-n+k} \cdot x_j \right) \text{ gemäß Induktionsvoraussetzung} \\ &= \bigoplus_{k=0}^{n-1} \bigoplus_{j=0}^{n-1} c_{k+1} \cdot L_{j,i-n+k} \cdot x_j \\ &= \bigoplus_{j=0}^{n-1} x_j \underbrace{\bigoplus_{k=0}^{n-1} c_{k+1} \cdot L_{j,i-n+k}}_{L_{j,i}} \end{aligned}$$

Somit gilt die Behauptung auch für $i \geq n$. □

Man kann ein LFSR L der Länge n als einen endlichen Automaten auffassen, dessen Zustandsmenge Q durch alle 2^n möglichen Belegungskombinationen der Speicherzellen q_1, \dots, q_n gegeben ist, d.h. es gilt für die Menge Q der inneren Zustände

$$Q = \{(q_1, \dots, q_n) \mid q_i \in \{0, 1\} \forall i \in \{1, \dots, n\}\} \text{ und } |Q| = 2^n$$

Ausgehend vom durch die Anfangsbelegung gegebenen Initialzustand $q(0) = (x_0, \dots, x_{n-1})$ geht das LFSR in jedem Takt t vom Zustand $q(t-1)$ in einen durch den Koeffizientenvektor c und $q(t-1)$ eindeutig bestimmten Nachfolgezustand $q(t)$ über. Offensichtlich erzeugt die Anfangsbelegung $x = (0, \dots, 0) \in \{0, 1\}^n$ für alle Koeffizientenvektoren einen trivialen, nur aus Nullen bestehenden Ausgabebitstrom und es gilt $q(t) = q(0)$ für alle Takte t . Wir wollen daher im Folgenden voraussetzen, dass der Initialzustand von Null verschieden ist.

Da der Ausgabebitstrom von L durch den Initialzustand x und den Koeffizientenvektor c bereits vollständig determiniert ist, erhalten wir:

Beobachtung 4. Ein LFSR der Länge n mit einem fixierten Koeffizientenvektor c kann höchstens $2^n - 1$ verschiedene nichttriviale Bitströme erzeugen.

Da $|Q| = 2^n$ endlich ist, wird nach spätestens 2^n Takten ein Zustand $q \in Q$ zum zweiten Mal erreicht, d.h. jeder durch L erzeugte Ausgabebitstrom ist zwangsläufig periodisch. Genauer gilt (vgl. [Gol82], Kapitel 2, Theorem 2.1)

Lemma 5. *Unabhängig vom Initialzustand x ist jeder durch ein LFSR der Länge n erzeugte Bitstrom periodisch mit einer Periode $p \leq 2^n - 1$, d.h. es gilt $L_r(x) = L_{r+k \cdot p}(x)$ für genügend große $r \geq 0$ und alle $k \geq 0$. \square*

Ein Bitstrom, der sich periodisch wiederholt, erweckt schon intuitiv einen äußerst nichtzufälligen Eindruck und kommt daher höchstens während der ersten Periode, d.h. bevor die ersten Wiederholungen auftreten, als Schlüsselbitstrom in Frage. Der Koeffizientenvektor c sollte also so gewählt werden, dass die Periode des Ausgabebitstroms möglichst groß wird, damit ein möglichst langer Präfix der Ausgabe genutzt werden kann. Für jede Registerlänge n kann man die Koeffizienten tatsächlich so wählen, dass der Ausgabebitstrom die maximale Periode von $2^n - 1$ erreicht:

Lemma 6 ([Rue86]). *Ein LFSR L der Länge n mit dem Koeffizientenvektor $c = (c_1, \dots, c_n)$ produziert für jeden Initialzustand $x \in \{0, 1\}^n$, $x \neq 0$, einen Bitstrom mit einer Periode $p = 2^n - 1$, falls es sich bei dem mit L korrespondierenden charakteristischen Polynom*

$$F(x) = x^n + \sum_{i=1}^n c_i x^{i-1}$$

um ein primitives Polynom handelt.¹ \square

Neben einer maximalen Periode besitzt der Ausgabebitstrom eines LFSR, dessen charakteristisches Polynom primitiv ist, drei für zufällige Bitströme typische Eigenschaften (vgl. [Gol82], Kapitel 3, Theoreme 4.1, 4.2 und 4.4):

Lemma 7. *Es seien L ein LFSR der Länge n mit primitivem charakteristischem Polynom und $x \in \{0, 1\}^n$, $x \neq 0$. Ferner bezeichne*

$$a := (a_0, \dots, a_{2^p-1}) \text{ mit } a_i = L_i(x) \text{ und } p := 2^n - 1$$

die ersten beiden Perioden des Ausgabebitstroms $L(x)$. Dann gilt

- (i) $\sum_{i=0}^p a_i = 2^{r-1}$, d.h. jede Periode von $L(x)$ enthält genau 2^{r-1} Einsen und $2^{r-1} - 1$ Nullen.
- (ii) Die Hälfte aller Folgen gleicher Ziffern in einer Periode hat die Länge 1, ein Viertel aller Folgen hat die Länge 2 usw., bis die Anzahl der Folgen der Länge i den Wert 1 erreicht. Insbesondere existieren für jede Folgenlänge gleich viele 0- und 1-Folgen.
- (iii) Für die Autokorrelationsfunktion² $C(\tau) = \frac{1}{p} \sum_{i=1}^p b_i b_{i+\tau}$ mit $b_i := 1 - 2a_i$ gilt

$$C(0) = 1 \text{ und } C(\tau) = -\frac{1}{p} \text{ für } 0 < \tau < p \quad \square$$

2.4 Lineare Bitstromgeneratoren

Viele LFSR-basierte Schlüsselstromgeneratoren verwenden mehrere LFSR zur Erzeugung des internen Bitstroms, der schließlich zum Schlüsselbitstrom verdichtet wird. Um die Darstellung der Kryptanalyse

¹Auf die Definition und die Bestimmung primitiver Polynome für gegebenen Grad n soll an dieser Stelle nicht näher eingegangen werden, hierzu sei etwa auf [Rue86] und [Gol82] verwiesen. Für kleine n stellt [Cha] Listen aller primitiven Polynome zur Verfügung.

²Die Autokorrelationsfunktion für eine periodische Folge $\{c_n\}$ mit der Periode p ist definiert als $C(\tau) := \frac{1}{p} \sum_{i=1}^p c_i c_{i+\tau}$. Es gilt $\max_{\tau} \{C(\tau)\} = C(0)$, und falls $\{c_n\}$ eine echt zufällige Folge ist, nimmt $C(\tau)$ relativ kleine Werte für die meisten $\tau \in \{1, \dots, p-1\}$ an.

zu vereinfachen, wollen wir die Gesamtheit dieser LFSR zu einem linearen Bitstromgenerator in folgendem Sinn zusammenfassen:

Definition 8. Ein linearer Bitstromgenerator L mit einem Initialzustand $x \in \{0, 1\}^n$ erzeugt mit Hilfe von $k \geq 1$ parallelen LFSR L^r der Länge n_r , $r = 0, \dots, k-1$, einen linearen Bitstrom

$$L(x) = L_0(x), L_1(x), \dots, L_i(x), \dots$$

mit

$$L_i(x) = L_{s(i)}^{r(i)}(x^{r(i)}), \text{ wobei } \begin{matrix} r(i) = i \bmod k \\ s(i) = i \operatorname{div} k \end{matrix} \text{ d.h. } i = k \cdot s + r$$

und $n_0 + \dots + n_{k-1} = n$, d.h. das i -te Ausgabebit von L entspricht dem $s(i)$ -ten Ausgabebit des LFSRs $L^{r(i)}$. Der Initialzustand x von L ist zusammengesetzt aus den Initialzuständen $x^p \in \{0, 1\}^{n_p}$, $p = 0, \dots, k-1$, der LFSR L^0, \dots, L^{k-1} .

Abbildung 2.2 illustriert diese Definition anhand der Ausgabebitströme von drei LFSR L^0 , L^1 und L^2 mit $n_0 = n_2 = 4$ sowie $n_1 = 3$. Aus Gründen der Übersichtlichkeit wurden die Initialzustände in der Darstellung weggelassen, d.h. die L_s^r sind als $L_s^r(x^r)$ und die L_m als $L_m(x)$ aufzufassen.

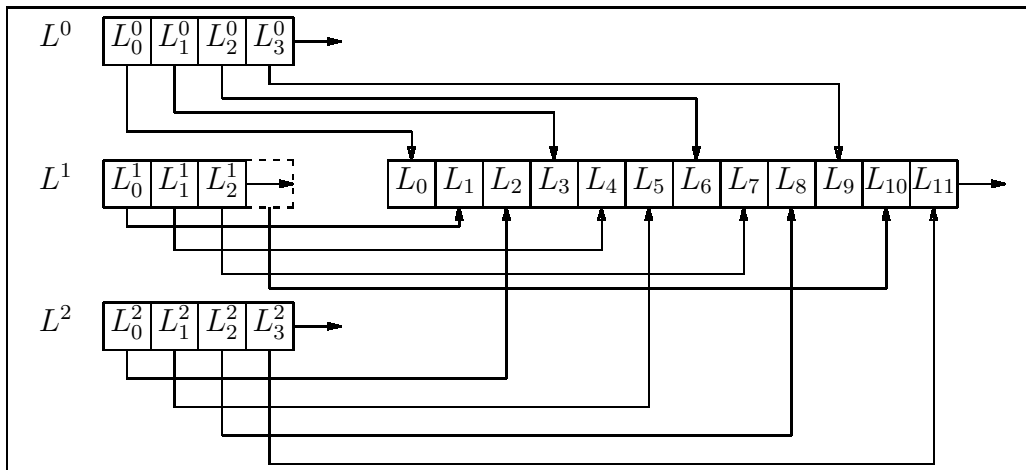


Abbildung 2.2: Linearer Bitstromgenerator

Auch der Ausgabebitstrom eines linearen Bitstromgenerators kann analog zu einem einzelnen LFSR in Initialzustandsbits und linear kombinierte Bits partitioniert werden:

Definition 9. Für alle $x \in \{0, 1\}^n$ und $m \geq 1$ bezeichne $L_{\leq m}(x)$ die ersten m Bits von $L(x)$, d.h.

$$L_{\leq m}(x) := (L_0(x), L_1(x), \dots, L_{m-1}(x))$$

Ferner seien $I_m(L)$ die Menge der Indizes von Initialzustandsbits und $C_m(L)$ die Menge der Indizes von Linearkombinationsbits in der Indexmenge $\{0, \dots, m-1\}$ d.h.

$$\begin{aligned} I_m(L) &= \{j \in \{0, \dots, m-1\} \mid s(j) < n_{r(j)}\} \\ \text{und } C_m(L) &= \{j \in \{0, \dots, m-1\} \mid s(j) \geq n_{r(j)}\} = \{0, \dots, m-1\} \setminus I_m \end{aligned}$$

Für einen gegebenen Bitstrom $z = (z_0, \dots, z_{m-1})$ bezeichnen $i_m(L, z)$ und $c_m(L, z)$ die Initialzustands- bzw. Linearkombinationsbits in z , d.h.

$$\begin{aligned} i_m(L, z) &= (z_{j_1}, \dots, z_{j_{|I_m(L)|}}) \text{ mit } j_p \in I_m(L) \forall p \in \{1, \dots, |I_m(L)|\} \text{ und } j_p < j_q \forall p < q \\ c_m(L, z) &= (z_{j_1}, \dots, z_{j_{|C_m(L)|}}) \text{ mit } j_p \in C_m(L) \forall p \in \{1, \dots, |C_m(L)|\} \text{ und } j_p < j_q \forall p < q \end{aligned}$$

Schließlich benötigen wir für unsere Kryptanalyse die Position des ersten linear kombinierten Bits in $L(x)$ und definieren daher

Definition 10. *Es sei $n' := \min_i \{i \in C_m(L)\}$ der Index des ersten linear kombinierten Bits im von L erzeugten Bitstrom.*

In Abbildung 2.2 ist beispielsweise das Bit $L_{10}(x)$ das erste linear kombinierte Bit des Bitstroms $L(x)$.

Aus der Definition linearer Bitstromgeneratoren folgt unmittelbar:

Beobachtung 11. *Für jeden linearen Bitstromgenerator L der Länge n ist $n' \leq n$. Gleichheit gilt genau dann, wenn $n_r = \frac{n}{k}$ für alle $r \in \{0, \dots, k-1\}$, d.h. wenn alle LFSR in L dieselbe Länge besitzen.*

2.5 LFSR-basierte Schlüsselstromgeneratoren

Definition 12. *Ein LFSR-basierter Schlüsselstromgenerator $K = (L, C)$ erzeugt mit Hilfe eines linearen Bitstromgenerators L mit den LFSR L^0, \dots, L^{k-1} und einem Initialzustand $x \in \{0, 1\}^n$ den internen Bitstrom $z = L(x)$ und berechnet hieraus den Schlüsselstrom $y \in \{0, 1\}^*$ als*

$$y = C(z) = C(L(x))$$

wobei C eine nichtlineare Kompressionsfunktion $C : \{0, 1\}^* \rightarrow \{0, 1\}^*$ darstellt. C berechnet die Bits des Schlüsselstroms online, d.h. es existiert eine Funktion $\delta : N \rightarrow N$ mit $\delta(i) < \delta(j)$ für $i < j$, so dass der Wert des i -ten Schlüsselbits nur von den ersten $\delta(i)$ Bits des internen Bitstroms abhängt. Insbesondere werden die Bits des internen Bitstroms durch C unter Beachtung der Reihenfolge gelesen, in der sie durch die jeweiligen LFSR produziert werden, d.h. für $s > 0$ und alle $r \in \{0, \dots, k-1\}$ wird das Bit $L_{k \cdot s + r}(x)$ nicht vor dem Bit $L_{k \cdot (s-1) + r}(x)$ von C eingelesen.

Abbildung 2.3 veranschaulicht diese Definition und integriert den linearen Bitstromgenerator aus dem letzten Abschnitt in einen LFSR-basierten Schlüsselstromgenerator.

Definition 13. *Zwei Initialzustände $x, x' \in \{0, 1\}^n$ eines LFSR-basierten Schlüsselstromgenerators $K = (L, C)$ heißen äquivalent, falls*

$$C(L(x)) = C(L(x'))$$

d.h. falls K aus x und x' denselben Schlüsselstrom erzeugt.

Grundlegendes Designziel der Kompressionsfunktion C ist es, bei der Transformation des internen Bitstroms z in den Schlüsselstrom y die geringe lineare Komplexität von z zu erhöhen, ohne dabei die guten Pseudozufälligkeitseigenschaften zunichte zu machen. Auf diesem Hintergrund wollen wir nun einige wichtige Parameter der Kompressionsfunktion C untersuchen.

2.5.1 best case Kompressionsrate γ

Da die Verschlüsselung in Echtzeit stattfindet, sollte der Zeitabstand zwischen der Ausgabe zweier Schlüsselbits klein sein und nicht zu stark schwanken, d.h. es gilt:

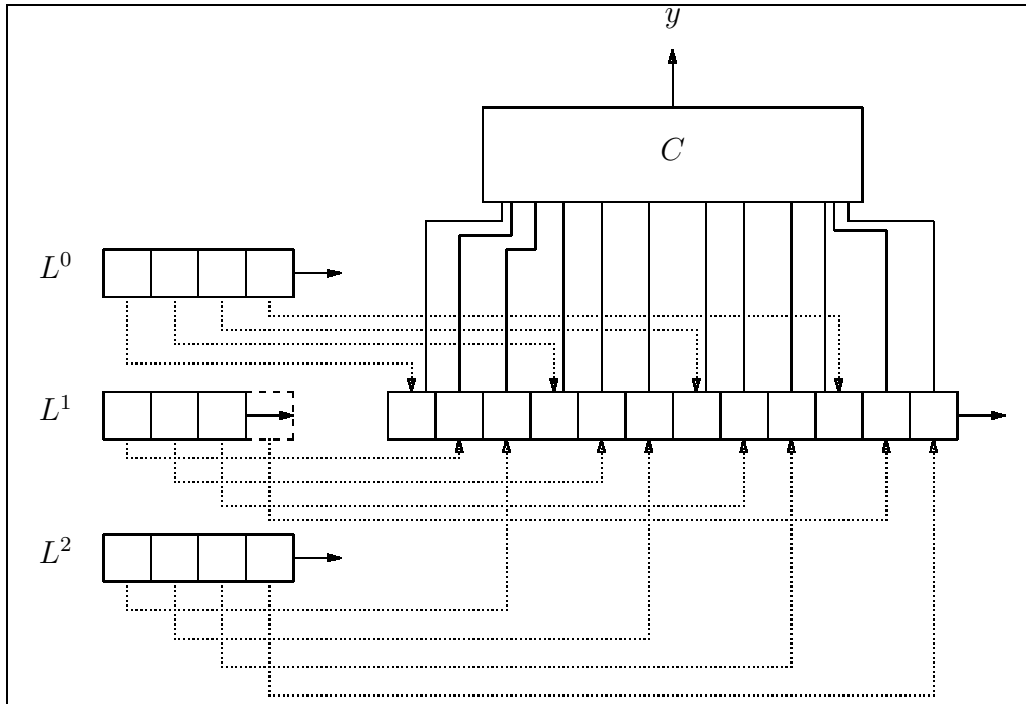


Abbildung 2.3: LFSR-basierter Schlüsselstromgenerator

Annahme 1 (Partitionierungsregel). Jeder interne Bitstrom $z = L(x)$ eines LFSR-basierten Schlüsselstromgenerators kann in Elementarblöcke z^0, z^1, \dots, z^{l-1} mit $|z^j| \geq 1$ aufgeteilt werden, so dass

$$C(z) = y^0, y^1, \dots, y^{l-1}$$

mit $y^j = C(z^j)$ und $|y^j| = 1$ für $j = 0, \dots, l-1$. Für die durchschnittliche Länge β der Elementarblöcke gilt

$$\beta = \frac{\sum_{j=0}^{l-1} |z^j|}{l} \geq 1$$

d.h. es werden im Durchschnitt β Bits des internen Bitstroms für die Berechnung eines Schlüsselbits verwendet.

Definition 14. Es sei γ die best case Kompressionsrate von C und damit γm die maximale Anzahl von Schlüsselbits, die die Kompressionsfunktion C aus internen Bitströmen der Länge m erzeugt.

Für die best case Kompressionsrate γ folgt aus der Partitionierungsregel (Annahme 1) unmittelbar $\gamma \in (0, 1]$.

2.5.2 Informationsrate α

Ein entscheidender Sicherheitsparameter eines LFSR-basierten Schlüsselstromgenerators ist die durchschnittliche Information, die ein beobachteter Schlüsselstrom y über den internen Bitstrom z liefert, aus dem y berechnet worden ist. Je mehr Information aus dem Schlüsselstrom über z gewonnen werden kann, desto einfacher ist es, den internen Bitstrom z und daraus den Initialzustand x zu rekonstruieren.

Aufgrund der Pseudozufälligkeitseigenschaften des internen Bitstroms nehmen wir eine Gleichverteilung auf der Menge der internen Bitströme an und betrachten den Wahrscheinlichkeitsraum

$$\mathcal{Z} = (\{0, 1\}^m, \text{Prob}[z] = 2^{-m} \text{ für alle } z \in \{0, 1\}^m)$$

sowie die Zufallsvariablen

$$\begin{aligned} Z^{(m)} : \mathcal{Z} &\rightarrow \{0, 1\}^m \\ Y : \mathcal{Z} &\rightarrow \{0, 1\}^* \end{aligned}$$

die der Auswahl des internen Bitstroms bzw. dem Auftreten des Schlüsselstroms Y entsprechen.

$\text{Prob}[Z^{(m)} = z | Y = y]$ gibt also die Wahrscheinlichkeit für das Ereignis an, dass der interne Bitstrom z zur Produktion des beobachteten Schlüsselstroms y verwendet wurde. Es gilt demnach

$$\begin{aligned} \text{Prob}[Z^{(m)} = z] &= 2^{-m} \\ \text{Prob}[Y = y] &= \sum_{z \in \tilde{Z}} \text{Prob}[Z = z] \text{ mit } \tilde{Z} := \{z \in \{0, 1\}^m | C(z) \text{ ist Präfix von } y\} \end{aligned}$$

Wir können nun die durchschnittliche Information in folgender Weise definieren:

Definition 15. Für einen zufällig gemäß Gleichverteilung gewählten internen Bitstrom $Z^{(m)} \in \{0, 1\}^m$ und einen zufälligen Schlüsselstrom Y sei

$$\alpha := \frac{1}{m} I(Z^{(m)}, Y) \in (0, 1]$$

die durchschnittliche Information, die Y über $Z^{(m)}$ liefert.

Diese durchschnittliche Information wollen wir nun genauer bestimmen. Wir betrachten zunächst die Wahrscheinlichkeit für das Ereignis, dass für einen zufällig gemäß Gleichverteilung gewählten internen Bitstrom $z \in \{0, 1\}^m$ die Schlüsselbits $C(z)$ Präfix eines gegebenen Schlüsselstroms $y \in \{0, 1\}^*$ sind. Diese Wahrscheinlichkeit kann dargestellt werden als

$$\begin{aligned} \text{Prob}_{z \in \{0, 1\}^m} [C(z) \text{ ist Präfix von } y] &= \\ \sum_{i=0}^{\lceil \gamma m \rceil} \text{Prob}_{z \in \{0, 1\}^m} [|C(z)| = i] \cdot \text{Prob}_{z \in \{0, 1\}^m, |C(z)|=i} [C(z) = (y_0, \dots, y_{i-1})] \end{aligned}$$

Wir wollen für die Bestimmung von α zunächst die folgende Annahme treffen:

Annahme 2 (Unabhängigkeitsannahme). Für alle $m \geq 1$, ein zufällig gemäß Gleichverteilung gewähltes $z \in \{0, 1\}^m$ und alle Schlüsselströme $y \in \{0, 1\}^*$ gilt

$$\text{Prob}_z [C(z) \text{ ist Präfix von } y] = p_C(m)$$

d.h. die Wahrscheinlichkeit, dass für den internen Bitstrom z die Schlüsselbits $C(z)$ Präfix von y sind, ist für alle Schlüsselströme y gleich groß.

Dass die Unabhängigkeitsannahme durch die Kompressionsfunktion des A5/1 Schlüsselstromgenerators erfüllt wird, werden wir in Kapitel 7.1 nachweisen.

Wir sind nun in der Lage, die folgende allgemeine Darstellung der Informationsrate α herzuleiten:

Lemma 16. *Unter der Unabhängigkeitsannahme (Annahme 2) gilt für die Informationsrate*

$$\alpha = -\frac{1}{m} \log_2 p_C(m)$$

Beweis. Es gilt wegen der allgemeinen Definition von Information und Entropie

$$\alpha = \frac{1}{m} I(Z^{(m)}, Y) = \frac{1}{m} \left(H(Z^{(m)}) - H(Z^{(m)}|Y) \right) = \frac{1}{m} \left(m - H(Z^{(m)}|Y) \right)$$

sowie

$$H(Z^{(m)}|Y) = \sum_{y \in \{0,1\}^*} \text{Prob}[Y = y] \left(- \sum_{z \in \{0,1\}^m} \text{Prob}[Z^{(m)} = z|Y = y] \cdot \log_2 \text{Prob}[Z^{(m)} = z|Y = y] \right)$$

Unter der Unabhängigkeitsannahme (Annahme 2) gilt für $z \in \{0,1\}^m$ und $y \in \{0,1\}^*$

$$\text{Prob}[Z^{(m)} = z|Y = y] = \begin{cases} \frac{1}{p_C(m) \cdot 2^m} & \text{falls } C(z) \text{ Präfix von } y \text{ ist} \\ 0 & \text{sonst} \end{cases}$$

Damit folgt mit $\tilde{Z} := \{z \in \{0,1\}^m | C(z) \text{ ist Präfix von } y\}$

$$H(Z^{(m)}|Y) = \sum_{y \in \{0,1\}^*} \text{Prob}[Y = y] \underbrace{\left(- \sum_{z \in \tilde{Z}} (p_C(m) 2^m)^{-1} \cdot \log_2((p_C(m) 2^m)^{-1}) \right)}_{\log_2(p_C(m) 2^m)} = \log_2(p_C(m) 2^m)$$

und schließlich

$$\alpha = -\frac{1}{m} (m - \log_2(p_C(m) 2^m)) = \frac{1}{m} (m - \log_2 p_C(m) - m) = -\frac{1}{m} \log_2 p_C(m) \quad \square$$

Gemäß der Partitionierungsregel (Annahme 1) verwendet die Kompressionsfunktion C im Durchschnitt $\beta \geq 1$ Bits des internen Bitstroms für die Berechnung eines Schlüsselbits, d.h. aus einem internen Bitstrom der Länge m werden durchschnittlich $\lfloor \beta^{-1} m \rfloor$ Schlüsselbits $\tilde{y}_0, \tilde{y}_1, \dots, \tilde{y}_{\frac{m}{\beta}-1}$ erzeugt.

Falls alle Elementarblöcke z^0, z^1, \dots, z^{l-1} von z dieselbe Länge haben, d.h. $|z^j| = k$ für alle $j = 0, \dots, l-1$, gilt auch für die durchschnittliche Länge der Elementarblöcke $\beta = k$ und damit $\alpha = \gamma = \frac{1}{k}$. Wenn die Länge der Elementarblöcke nicht konstant ist, kann α bestimmt werden aus

$$2^{-\alpha m} = p_C(m) = \text{Prob}_z[C(z) \text{ ist Präfix von } y] \quad (2.1)$$

Diesen Zusammenhang werden wir für den A5/1 Schlüsselstromgenerator ausnutzen.

Schließlich wollen wir noch die folgende, für die Kryptanalyse grundlegende Annahme treffen:

Annahme 3 (Pseudozufälligkeitsannahme). *Für alle Schlüsselströme y und alle $m \leq \lceil \alpha^{-1} n \rceil$ gilt*

$$\text{Prob}_z[C(z) \text{ ist Präfix von } y] \approx \text{Prob}_x[C(L_{\leq m}(x)) \text{ ist Präfix von } y]$$

wobei z und x zufällige gemäß Gleichverteilung gewählte Elemente aus $\{0,1\}^m$ bzw. $\{0,1\}^{|L_m(L)|}$ darstellen.

Aus einer groben Verletzung der Pseudozufälligkeitsannahme würde folgen, dass viele äquivalente Initialzustände existieren und damit bestimmte Schlüsselströme von mehr als einem Initialzustand x erzeugt werden, während andere Schlüsselströme nie als Ausgabe von K auftreten können. Die auf diese Weise entstehende statistische Auffälligkeit des Schlüsselstroms könnte etwa durch einen Korrelationsangriff ausgenutzt werden.

Kapitel 3

Kryptanalyse LFSR-basierter Schlüsselstromgeneratoren

3.1 Angriffsmodell

Nach der Diskussion des allgemeinen Aufbaus LFSR-basierter Schlüsselstromgeneratoren wollen wir nun den von [Kra02] gewählten Kryptanalyseansatz betrachten. Wir gehen dabei von folgendem Angriffsmodell aus:

Der Sender und der legale Empfänger benutzen einen LFSR-basierten Schlüsselstromgenerator $K = (L, C)$ mit einem geheimen Initialzustand $x \in \{0, 1\}^n$ zur verschlüsselten Übertragung einer Klartext-Nachricht $p \in \{0, 1\}^*$ entsprechend dem in Kapitel 2.1 dargestellten Verfahren. Der Angreifer kennt sämtliche Parameter von K , also insbesondere die Koeffizienten der LFSR und die Definition der Kompressionsfunktion C , und er ist in der Lage, den gesamten Chiffretext $(e_0, \dots, e_{|p|-1})$ abzuhören. Darüber hinaus gelingt es ihm, die ersten s Bits des Klartexts zu bestimmen, so dass er über s Paare $(p_0, e_0), \dots, (p_{s-1}, e_{s-1})$ mit Klartextbits (*known plaintext*) und zugehörigem Chiffretext verfügt. Aus diesen Informationen versucht der Angreifer, die Klartext-Nachricht p zu rekonstruieren.

Wir wollen uns am Beispiel des A5/1 Schlüsselstromgenerators kurz von der Plausibilität dieser Annahmen überzeugen. Trotz teilweise intensiver Bemühungen gelingt es den meisten Betreibern kryptographischer Systeme nicht, die verwendeten Verschlüsselungsalgorithmen lange geheim zu halten. So ist auch die Spezifikation des A5/1 Generators in Form von [BGW99] in die Öffentlichkeit gelangt.

Der A5/1 Generator wird zur Verschlüsselung von Gesprächsdaten verwendet, die zwischen Mobiltelefon und Basisstation über die Luftschnittstelle ausgetauscht werden. Diese Luftschnittstelle abzuhören und ein verschlüsseltes Gespräch aufzuzeichnen erfordert zwar einigen elektro- und nachrichtentechnischen Aufwand, ist aber im Prinzip möglich. Insbesondere stellt der Zugang zum Übertragungsmedium im Vergleich zum Anzapfen von Telefonleitungen (*wire-tapping*) ein weitaus kleineres Problem dar.

[Bri99] gibt an, dass ungefähr im ersten Zehntel eines Telefongesprächs digitalisierte Stille verschlüsselt wird, d.h. man kennt tatsächlich für die ersten etwa 1300 Bits des aufgezeichneten verschlüsselten Gesprächs den zugrundeliegenden Klartext.

3.2 Allgemeiner Kryptanalyseansatz

Nachdem wir festgelegt haben, welche Informationen dem Angreifer zur Verfügung stehen, versetzen wir uns nun in die Rolle des Kryptanalysten und versuchen, diese Informationen möglichst effizient zur Rekonstruktion des Klartexts auszunutzen.

Offensichtlich genügt es, den Schlüsselbitstrom zu rekonstruieren, denn in diesem Fall verfügen wir über dieselben Informationen wie der legale Empfänger der Nachricht und können p direkt aus dem Schlüsselbitstrom und dem abgehörten Chiffretext bestimmen. Aus der Betrachtung der LFSR-basierten Schlüsselstromgeneratoren wissen wir weiterhin, dass der gesamte Schlüsselstrom bereits vollständig durch den Initialzustand x des Schlüsselstromgenerators K determiniert wird. Wir können das Kryptanalyseproblem der Rekonstruktion des Klartexts p also auf die Bestimmung des Initialzustands x von K reduzieren.

Zunächst nutzen wir die zur Verfügung stehenden Klartext-Chiffretext Paare $(p_0, e_0), \dots, (p_{s-1}, e_{s-1})$ aus. Für alle $i \in \{0, \dots, s-1\}$ gilt

$$\begin{aligned} e_i \oplus p_i &= (p_i \oplus y_i) \oplus p_i \\ &= \underbrace{p_i \oplus p_i}_{=0} \oplus y_i \\ &= y_i \end{aligned}$$

d.h. wir erhalten aus $(p_0, e_0), \dots, (p_{s-1}, e_{s-1})$ die Schlüsselbits $(y_0, \dots, y_{s-1}) =: y$.

Weil wir die Definition von K und mit dem Schlüsselpräfix y einen Teil der Ausgabe von K kennen, stellen wir nun systematisch fest, welche Initialzustände x zu der von uns beobachteten Ausgabe geführt haben können.

3.3 Bestimmung des Initialzustands x

Wir wissen, dass LFSR-basierte Schlüsselstromgeneratoren anhand des Initialzustands x und dem linearen Bitstromgenerator L zunächst einen internen Bitstrom $z = L(x) = z_0, z_1, \dots$ berechnen und die Bits von z schließlich mit Hilfe der Kompressionsfunktion C zum Schlüsselstrom $y = C(z)$ verdichten. Da der Initialzustand x unmittelbar aus den Initialzustandsbits $b_j \in i_{|z|}(L, z)$ abgelesen werden kann, können wir unser Kryptanalyseproblem weiterhin auf die Bestimmung desjenigen durch L erzeugbaren internen Bitstroms $z \in \{0, 1\}^*$ reduzieren, für den $C(z) = y$ gilt.

Obwohl z beliebig lang sein kann, existieren höchstens $2^n - 1$ verschiedene Kandidaten (vgl. Beobachtung 4). Gesucht ist also ein Bitstrom $z \in \{0, 1\}^*$, der folgende Anforderungen erfüllt:

- (i) z ist ein durch L erzeugter interner Bitstrom, d.h. $z \in U$ mit

$$U := \{z = (z_0, \dots, z_{|z|-1}) \in \{0, 1\}^* \mid z = L(i_{|z|}(L, z))\}$$

- (ii) Der beobachtete Schlüsselstrom y kann mit Hilfe von C aus z berechnet werden, d.h. $z \in W$ mit

$$W := \{z = (z_0, \dots, z_{|z|-1}) \in \{0, 1\}^* \mid C(z) \text{ ist Präfix von } y\}$$

Insgesamt muss also $z \in T$ gelten mit

$$T := \{z = (z_0, \dots, z_{|z|-1}) \in \{0, 1\}^* \mid z = L(i_{|z|}(L, z)) \text{ und } C(z) \text{ ist Präfix von } y\} = U \cap W$$

Wir wollen nun die Mengen U , W und T für einen fest gewählten linearen Bitstromgenerator L mit geeigneten Folgen Boolescher Funktionen identifizieren:

Definition 17. Es seien \mathcal{U} , \mathcal{V} , \mathcal{W} , \mathcal{T} Folgen Boolescher Funktionen, die definiert sind durch

$$\begin{aligned} \mathcal{U} &:= (u_m : \{0, 1\}^m \rightarrow \{0, 1\})_{m>0} \text{ mit } u_m(z) = \begin{cases} 1 & \text{falls } z = L_{\leq m}(i_m(L, z)) \\ 0 & \text{sonst} \end{cases} \\ \mathcal{V} &:= (v_m : \{0, 1\}^m \rightarrow \{0, 1\})_{m>0} \text{ mit } v_m(z) = \begin{cases} 1 & \text{falls } z_{m-1} = L_{m-1}(i_m(L, z)) \\ 0 & \text{sonst} \end{cases} \\ \mathcal{W} &:= (w_m : \{0, 1\}^m \rightarrow \{0, 1\})_{m>0} \text{ mit } w_m(z) = \begin{cases} 1 & \text{falls } C(z) \text{ Prafix von } y \text{ ist} \\ 0 & \text{sonst} \end{cases} \\ \mathcal{T} &:= (t_m : \{0, 1\}^m \rightarrow \{0, 1\})_{m>0} \text{ mit } t_m(z) = (u_m \wedge w_m)(z) \end{aligned}$$

Ferner seien $U_m, V_m, W_m, T_m \subseteq \{0, 1\}^m$ die mit u_m, v_m, w_m bzw. t_m korrespondierenden Sprachen, d.h. $U_m := u_m^{-1}(1)$, $W_m := w_m^{-1}(1)$ und $T_m := t_m^{-1}(1)$.

Die Sprache T_m der sowohl mit L als auch mit C und y konsistenten internen Bitstrome kann in folgender Weise durch V_m und W_m ausgedruckt werden:

Lemma 18. Fur $m > 0$ gilt

$$T_m = \begin{cases} (T_{m-1} \times \{0, 1\}) \cap V_m \cap W_m & \text{fur } m > n' \\ W_m & \text{fur } m \leq n' \end{cases}$$

Beweis. Da die ersten n' Bit eines internen Bitstroms $z = (z_0, \dots, z_{m-1}) \in \{0, 1\}^m$ Initialzustandsbits sind, ist fur $m \leq n'$ offensichtlich $T_m = W_m$. Fur $m > n'$ gilt

$$\begin{aligned} z \in U_m &\Rightarrow (z_0, \dots, z_{m-2}) \in U_{m-1} \text{ wegen Definition 8} \\ z \in W_m &\Rightarrow (z_0, \dots, z_{m-2}) \in W_{m-1} \text{ wegen Definition 12} \end{aligned}$$

und damit auch

$$z \in T_m \Rightarrow (z_0, \dots, z_{m-2}) \in T_{m-1} \text{ oder aquivalent } (z_0, \dots, z_{m-2}) \notin T_{m-1} \Rightarrow z \notin T_m$$

d.h. $(z_0, \dots, z_{m-2}) \in T_{m-1}$ ist eine notwendige Bedingung fur $z \in T_m$.

Zusatzlich folgt aus der Definition von \mathcal{U} und \mathcal{V} , dass

$$U_m = \bigcap_{i=n'+1}^m (V_i \times \{0, 1\}^{m-i})$$

□

Fur hinreichend kleine m gilt $|T_m| > 1$, d.h. es existieren mehrere interne Bitstrome $z \in \{0, 1\}^m$, die durch L erzeugt werden konnen und fur die $C(z)$ Prafix von y ist. Andererseits produziert L hochstens $2^n - 1$ verschiedene interne Bitstrome, und die Anzahl der zu erfullenden Bedingungen wachst mit m , so dass fur ein genugend groes m^* die Sprache $T_{m^*} \subseteq \{0, 1\}^{m^*}$ mit hoher Wahrscheinlichkeit nur noch aus einem Element z^* besteht. Die in z^* enthaltenen Initialzustandsbits $i_m(L, z^*)$ stellen dann den gesuchten Initialzustand x dar.

Wir erhalten also basierend auf Lemma 18 den generischen Algorithmus 1 zur Berechnung von x .

Die Laufzeit dieses Algorithmus hangt im Wesentlichen von der Reprasentation der Sprachen T_m , W_m und U_m , der Effizienz der Syntheseoperation und der Anzahl der Schleifendurchlaufe ab. Wir wollen daher im nachsten Abschnitt den Wert m^* bestimmen.

Algorithmus 1 COMPUTE-x (L,C)

```

 $T \leftarrow W_{n'}$ 
for  $m \leftarrow n' + 1$  to  $m^*$  do
     $T \leftarrow T \cap U_m \cap W_m$ 
return  $i_{m^*}(L, z^*)$  for a  $z^* \in T$ 

```

3.4 Bestimmung der Eindeutigkeitsgrenze m^*

Mit Hilfe der Pseudozufälligkeitsannahme (Annahme 3) kann unmittelbar die Anzahl der möglichen Anfangszustände bestimmt werden, die zur Generierung des beobachteten Schlüsselstroms y geführt haben können:

Lemma 19. *Falls für K die Pseudozufälligkeitsannahme zutrifft, gilt für alle Schlüsselströme y und alle $m \leq \lceil \alpha^{-1}n \rceil$*

$$|\{x \in \{0, 1\}^{|I_m(L)|} : C(L_{\leq m}(x)) \text{ ist Präfix von } y\}| \approx 2^{|I_m(L)| - \alpha m} \leq 2^{n - \alpha m}$$

d.h. es existieren ungefähr $2^{n - \alpha m}$ verschiedene Anfangszustände x , so dass $C(L_{\leq m}(x))$ Präfix von y ist.

Beweis. Aus der Pseudozufälligkeitsannahme und (16) folgt, dass für $m \leq \lceil \alpha^{-1}n \rceil$ und einen zufällig gemäß Gleichverteilung gewählten Initialzustand $x \in \{0, 1\}^{|I_m(L)|}$ gilt

$$\text{Prob}_x[L_{\leq m}(x) \text{ ist Präfix von } y] \approx \text{Prob}_z[C(z) \text{ ist Präfix von } y] = p_C(m) = 2^{-\alpha m}$$

Da die Anzahl der für die Berechnung von $L_{\leq m}(x)$ relevanten Initialzustände $|\{0, 1\}^{|I_m(L)|}| = 2^{|I_m(L)|}$ beträgt, muss die Anzahl der Initialzustände $x \in \{0, 1\}^{|I_m(L)|}$, so dass $C(L_{\leq m}(x))$ Präfix von y ist, etwa gleich $2^{|I_m(L)|} \cdot 2^{-\alpha m} = 2^{|I_m(L)| - \alpha m} \leq 2^{n - \alpha m}$ sein. \square

Da Lemma 19 für alle, also insbesondere auch für den beobachteten Schlüsselstrom y gilt, ist der gesuchte Wert m^* dasjenige $m \leq \lceil \alpha^{-1}n \rceil$, für das mit hoher Wahrscheinlichkeit nur ein Initialzustand x existiert, so dass $C(L_{\leq m}(x))$ Präfix von y ist. Dies ist der Fall für

$$|\{x \in \{0, 1\}^{|I_m(L)|} : C(L_{\leq m}(x)) \text{ ist Präfix von } y\}| = 1$$

d.h. $m = \lceil \alpha^{-1}|I_m(L)| \rceil \leq \lceil \alpha^{-1}n \rceil$.

Da aus den ersten $\lceil \alpha^{-1}n \rceil$ Bits des internen Bitstroms höchstens $\lceil \gamma \alpha^{-1}n \rceil$ Schlüsselbits berechnet werden, folgt insgesamt:

Theorem 20. *Es sei $K = (L, C)$ ein LFSR-basierter Schlüsselstromgenerator mit der Informationsrate α und der best case Kompressionsrate γ . Falls K die Unabhängigkeitsannahme (Annahme 2) sowie die Pseudozufälligkeitsannahme erfüllt und die Partitionierungsregel für K gültig ist, kann der gesuchte Initialzustand x aus den ersten $\lceil \alpha^{-1}n \rceil$ Bits des internen Bitstroms $L(x)$ und aus den ersten $\lceil \gamma \alpha^{-1}n \rceil$ Bits des beobachteten Schlüsselstroms y bestimmt werden. \square*

Kapitel 4

Repräsentation Boolescher Funktionen mit FBDDs

Neben der Eindeutigkeitsgrenze m^* haben wir bereits die effiziente Repräsentation der Mengen T_m , W_m und U_m und ihrer korrespondierenden Booleschen Funktionen als entscheidenden Einflussfaktor der Laufzeit des Kryptanalysealgorithmus (vgl. Algorithmus 1) identifiziert. In diesem Kapitel sollen nun Binäre Entscheidungsgraphen (BDDs) als eine für unsere Zwecke besonders geeignete Datenstruktur zur Repräsentation Boolescher Funktionen vorgestellt werden.

4.1 Operationen auf Booleschen Funktionen

Eine Datenstruktur G_f , die eine Boolesche Funktion

$$f \in B_n = \{f \mid f : \{0, 1\}^n \rightarrow \{0, 1\}\}$$

über der Variablenmenge $X_n = \{x_1, \dots, x_n\}$ repräsentiert, sollte eine kompakte Repräsentation möglichst vieler Funktionen aus B_n ermöglichen und gleichzeitig verschiedene Auswertungs- und Manipulationsoperationen effizient unterstützen.

In unserem Kontext sind insbesondere die folgenden Operationen von Bedeutung:

(i) **Minimierung**

Eingabe: G_f

Ausgabe: G_f^* vom selben Repräsentationstyp wie G_f mit minimaler Größe für f

(ii) **m -äre Synthese**

Eingabe: G_1, \dots, G_m , die die Funktionen $f_1, \dots, f_m \in B_n$ repräsentieren, sowie eine m -äre Syntheseoperation $\otimes : \{0, 1\}^m \rightarrow \{0, 1\}$

Ausgabe: G_f mit $f = \otimes(f_1, \dots, f_m)$

(iii) **SAT-ENUM**

Eingabe: G_f

Ausgabe: Alle erfüllenden Variablenbelegungen, d.h. alle $a \in \{0, 1\}^n$ mit $f(a) = 1$

Im Folgenden sollen nun binäre Entscheidungsgraphen als Datenstrukturen für Boolesche Funktionen vorgestellt und Algorithmen für die genannten Operationen entwickelt werden.

4.2 Binäre Entscheidungsgraphen (BDDs)

Binäre Entscheidungsgraphen (*Binary Decision Diagrams*, kurz BDDs) und ihre verschiedenen Varianten können als für Theorie und Praxis gleichermaßen grundlegende Datenstrukturen für Boolesche Funktionen betrachtet werden. Für diese Datenstruktur ist vor allem im Kontext der Komplexitätstheorie die Bezeichnung *Branching Program* (BP) geläufig, während der Name BDD eher im Anwendungsbereich Verwendung findet.

Definition 21. Ein Binärer Entscheidungsgraph (BDD) bzw. ein Branching Program (BP) über der Variablenmenge $X_n = \{x_1, \dots, x_n\}$ ist ein gerichteter azyklischer Graph $G = (V, E)$ mit $E \subseteq V \times V \times \{0, 1\}$. Jeder innere Knoten v besitzt genau zwei ausgehende Kanten, eine 0-Kante $(v, v_0, 0)$, die zum 0-Nachfolger v_0 führt, und eine 1-Kante $(v, v_1, 1)$, die den 1-Nachfolger v_1 als Endknoten besitzt. Ein BDD enthält genau 2 Knoten ohne ausgehende Kanten, die Senken s_0 und s_1 . Jeder Knoten $v \in V$ trägt eine Bezeichnung $v.label$. Für die inneren Knoten gilt $v.label \in X_n$, während die Senken s_0 und s_1 mit 0 bzw. 1 beschriftet werden. Es existiert genau ein innerer Knoten ohne eingehende Kanten, die Wurzel des BDD. Die Größe $|G|$ eines BDDs G entspricht der Anzahl der Knoten in G , d.h. $|G| := |V|$. Jeder Knoten v eines BDDs über X_n repräsentiert eine Boolesche Funktion $f_v \in B_n$ in folgender Weise: Die Berechnung von $f_v(a)$, $a = (a_1, \dots, a_n) \in \{0, 1\}^n$, beginnt im Knoten v . In einem Knoten mit Bezeichnung x_i wird diejenige ausgehende Kante gewählt, die mit a_i beschriftet ist. Der Wert $f_v(a)$ entspricht dann der Bezeichnung der Senke, zu der der durch a bestimmte Pfad führt.

Ein beispielhafter BDD ist in Abbildung 4.1 dargestellt. Für die 0-Kanten werden gepunktete Linien und für die 1-Kanten durchgezogene Linien verwendet. Für die Eingabe $(x_1, x_2, x_3) = (1, 1, 0)$ verläuft der Berechnungspfad vom x_1 -Knoten zum x_2 -Knoten und von dort aus zur 1-Senke. Damit wissen wir, dass $f(0, 1, 1) = 1$ gilt.

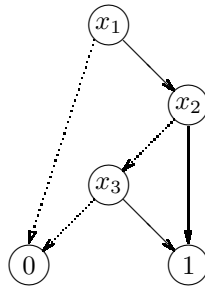


Abbildung 4.1: Eine mögliche BDD-Darstellung der Funktion $f(x_1, x_2, x_3) = x_1x_2 + x_1x_3$

Definition 22. Für einen BDD G über X_n sei $One(G) \subseteq \{0, 1\}^n$ die Menge aller Eingaben $a \in \{0, 1\}^n$, für die der in der Wurzel von G beginnende, durch a bestimmte Pfad in G zur 1-Senke führt.

Definition 23. Zwei BDDs G_1 und G_2 über X_n heißen isomorph, falls bei einer simultanen Tiefensuche ausgehend von der Wurzel in G_1 bzw. G_2 , die zunächst die 1-Kanten besucht, in G_1 und G_2 in jedem Schritt Knoten mit derselben Bezeichnung erreicht werden, d.h. wenn für die Folgen der in G_1 besuchten Knoten $(u_i^1)_{1 \leq i \leq p_1}$ und der in G_2 besuchten Knoten $(u_i^2)_{1 \leq i \leq p_2}$ gilt $p_1 = p_2$ und $u_i^1.label = u_i^2.label$.

Definition 24. Falls die Repräsentation minimaler Größe G_f^* für jede Boolesche Funktion f innerhalb eines Repräsentationstyps eindeutig bis auf isomorphe Repräsentationen ist, wird G_f^* reduzierte Repräsentation und die Minimierungsoperation Reduktion genannt.

Im Allgemeinen sind BDDs in der Lage, viele Boolesche Funktionen in kompakter Weise darzustellen. Allerdings die Minimierung eines gegebenen BDDs NP-schwer (vgl. [Weg00], Theorem 2.4.2), so dass allgemeine BDDs für viele Anwendungen ungeeignet sind. Durch geeignete Restriktionen, insbesondere durch eine festgelegte Einleseordnung der Variablen, kann man jedoch die effiziente Ausführbarkeit vieler wichtiger Operationen erreichen. Diese Eigenschaft wird durch die nun zu betrachtenden BDD-Varianten ausgenutzt.

4.3 Geordnete binäre Entscheidungsgraphen (OBDDs)

Die von [Bry86] eingeführten geordneten binären Entscheidungsgraphen (*Ordered Binary Decision Diagrams*, kurz OBDDs) schränken allgemeine BDDs dahingehend ein, dass auf jedem Pfad eine bestimmte Einleseordnung der Variablen x_1, \dots, x_n eingehalten werden muss. Genauer definiert man:

Definition 25. Eine Variablenordnung π über der Variablenmenge $X_n = \{x_1, \dots, x_n\}$ ist eine Permutation der Indexmenge $I = \{1, \dots, n\}$. Die Position der Variablen x_i in der π -geordneten Variablenliste ist $\pi(i)$. Damit ergibt sich die π -geordnete Variablenliste als $x_{\pi^{-1}(1)}, \dots, x_{\pi^{-1}(n)}$.

Definition 26.

(i) Ein π -OBDD über der Variablenmenge $X_n = \{x_1, \dots, x_n\}$ bezüglich einer gegebenen Variablenordnung π ist ein BDD, in dem auf jedem Pfad von der Wurzel zu einer Senke die Variablenordnung π respektiert wird, d.h. wenn eine gerichtete Kante von einem x_i -Knoten zu einem x_j -Knoten existiert, dann gilt $\pi(i) < \pi(j)$.

(ii) Ein BDD G heisst OBDD, falls eine Variablenordnung π existiert, so dass G ein π -OBDD ist.

Abbildung 4.2 zeigt einen beispielhaften π -OBDD.

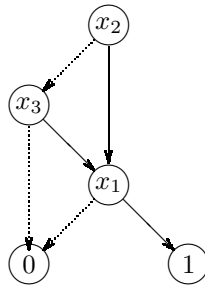


Abbildung 4.2: π -OBDD-Darstellung der Funktion $f(x_1, x_2, x_3) = x_1x_2 + x_1x_3$ mit $\pi(2) = 1$, $\pi(1) = 3$ und $\pi(3) = 2$

Im Unterschied zu allgemeinen BDDs gilt (vgl. [Weg00], Theoreme 3.1.4 und 3.3.4):

Lemma 27. Für jede Funktion $f \in B_n$ und eine gegebene Variablenordnung π kann aus einem π -OBDD G_f in Zeit $O(|G_f|)$ der minimale π -OBDD G_f^* berechnet werden. G_f^* ist bis auf isomorphe Repräsentationen eindeutig bestimmt. \square

Damit besitzt jede Boolesche Funktion $f \in B_n$ für eine gegebene Variablenordnung π eine kanonische Darstellung in Form eines reduzierten π -OBDDs.

Auch für die übrigen in unserem Kontext benötigten Operationen existieren effiziente Algorithmen: (vgl. [Weg00], Theoreme 3.3.6 und 3.3.1):

Lemma 28. *Es seien $f_1, \dots, f_m \in B_n$ gegeben durch die minimierten π -OBDDs G_1, \dots, G_m über X_n . Dann sind die Operationen*

- (i) *m-äre Synthese von G_1, \dots, G_m hinsichtlich einer m-ären Operation $\otimes \in B_m$ in Zeit und mit Speicherplatzbedarf $O(\prod_{i=1}^m |G_i|)$*
- (ii) *SAT-ENUM für G_f in Zeit $O(|G_f|)$*

ausführbar. □

OBDDs haben für viele praktische Probleme wie Schaltkreisverifikation und automatische Testfallgenerierung große Bedeutung erlangt. In unserem Kryptanalysekontext, aber auch in anderen Anwendungsfällen hängt die Einlesereihenfolge der Variablen von der Variablenbelegung ab und kann nicht wie von der OBDD-Definition gefordert global festgelegt werden. In den meisten solcher Fälle stellt die Verwendung von freien binäre Entscheidungsgraphen an Stelle von OBDDs eine geeignete Alternative dar.

4.4 Freie binäre Entscheidungsgraphen (FBDDs)

Freie binäre Entscheidungsgraphen (*Free Binary Decision Diagrams*, kurz FBDDs), die unter dem Namen *Read-Once Branching Programs* erstmals von [Mas76] untersucht wurden, verallgemeinern OBDDs dahingehend, dass die Reihenfolge, in der die Variablen x_1, \dots, x_n gelesen werden, nicht global durch eine Permutation der Indexmenge definiert und damit für alle Eingabevektoren $a \in \{0, 1\}^n$ gleich ist, sondern mit Hilfe eines Steuerungsgraphen für jeden Eingabevektor individuell festgelegt werden kann.

Definition 29. *Ein Steuerungsgraph (oracle graph) $G_0 = (V, E)$ über der Variablenmenge $X_n = \{x_1, \dots, x_n\}$ ist ein in folgender Weise modifizierter BDD:*

- (i) *G_0 enthält nur eine Senke s . Diese ist mit $*$ bezeichnet.*
- (ii) *Für alle $x_i \in X_n$ existiert auf jedem Pfad von der Wurzel zur Senke genau ein Knoten, der mit x_i bezeichnet ist.*

Abbildung 4.3 zeigt einen Steuerungsgraphen über X_3 .

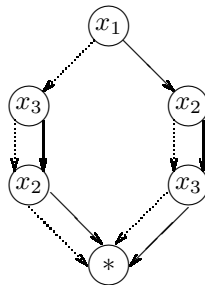


Abbildung 4.3: Steuerungsgraph G_0 über X_3

Bemerkung 30. Ein Steuerungsgraph G_0 über der Variablenmenge $X_n = \{x_1, \dots, x_n\}$ kann also als Funktion

$$\begin{aligned} \pi_{G_0} : \{0, 1\}^n &\rightarrow \mathcal{S}_n \\ b &\mapsto \pi_{G_0}(b) \end{aligned}$$

aufgefasst werden, die jeder Variablenbelegung $b \in \{0, 1\}^n$ eine Variablenordnung in Form einer Permutation der Menge $\{1, \dots, n\}$ zuordnet.

Definition 31.

(i) Ein G_0 -FBDD G über der Variablenmenge $X_n = \{x_1, \dots, x_n\}$ bezüglich eines Steuerungsgraphen G_0 über X_n ist ein BDD, der die folgende Bedingung für alle Eingabevektoren $a \in \{0, 1\}^n$ erfüllt: Die Liste $G_0(a)$ enthalte die Variablen $x_i \in X_n$ in der Reihenfolge, in der sie auf dem durch a bestimmten Pfad in G_0 abgefragt werden. Analog enthalte die Liste $G(a)$ die in G auf dem durch a bestimmten Pfad abgefragten Variablen $x_i \in X_n$ in der entsprechenden Reihenfolge. Wenn die Variablen x_i und x_j in $G(a)$ enthalten sind, dann treten x_i und x_j in $G(a)$ in derselben Reihenfolge wie in $G_0(a)$ auf.

(ii) Ein BDD G heisst FBDD, falls ein Steuerungsgraph G_0 existiert, so dass G ein G_0 -FBDD ist.

OBDDs können somit als spezielle FBDDs aufgefasst werden, deren Steuerungsgraph zu einer Liste entartet ist.

Der in Abbildung 4.1 dargestellte BDD ist beispielsweise ein G_0 -FBDD bezüglich des in Abbildung 4.3 definierten Steuerungsgraphen G_0 .

Aus Definition 31 folgt unmittelbar die folgende wichtige Eigenschaft:

Bemerkung 32 (read-once Eigenschaft). In jedem FBDD G über einer Variablenmenge $X_n = \{x_1, \dots, x_n\}$ wird jede Variable $x_i \in X_n$ auf jedem Pfad in G höchstens einmal eingelesen.

Schließlich benötigen wir für unsere Untersuchungen folgenden Zusammenhang zwischen der Größe eines minimalen G_0 -FBDDs und der Anzahl der erfüllenden Variablenbelegungen:

Lemma 33. Für jeden minimalen G_0 -FBDD G über X_n gilt $|G| \leq n \cdot |\text{One}(G)|$.

Beweis. Es existieren höchstens $|\text{One}(G)|$ Pfade von der Wurzel zur 1-Senke in G , die wegen der *read-once* Eigenschaft (Bemerkung 32) jeweils höchstens n Variablen einlesen. Aus der Minimalität von G folgt, dass alle Kanten, die nicht Bestandteil eines Pfades zur 1-Senke sind, direkt auf die 0-Senke verweisen, denn andernfalls könnte man durch Weglassen der auf dem Weg zur 0-Senke besuchten Knoten einen kleineren G_0 -FBDD als G konstruieren. Somit enthält G höchstens $n \cdot |\text{One}(G)|$ verschiedene Knoten. \square

4.5 Operationen auf FBDDs

4.5.1 Minimierung

Analog zu Eindeutigkeit minimaler OBDDs gilt (vgl. [Weg00], Theorem 6.4.4):

Lemma 34. Es seien G_0 ein Steuerungsgraph und $f \in B_n$ eine Boolesche Funktion. Dann ist der minimale G_0 -FBDD G_f^* für f bis auf isomorphe Repräsentationen eindeutig bestimmt. \square

Es existiert also für jede Funktion $f \in B_n$ und einen gegebenen Steuerungsgraphen G_0 eine kanonische Darstellung in Form eines minimalen G_0 -FBDDs. Diese Darstellung kann etwa durch die Erweiterung des Steuerungsgraphen G_0 zu einem Entscheidungsbaum für f und die anschließende Reduktion des so konstruierten G_0 -FBDDs erzeugt werden.

Der Reduktionsalgorithmus für G_0 -FBDDs basiert auf den folgenden zwei Reduktionsregeln:

Definition 35. Es seien G_0 ein Steuerungsgraph und $G = (V, E)$ ein G_0 -FBDD.

- (i) *Elimination rule:* Wenn für zwei Knoten $u, v \in V$ gilt $u_0 = u_1 = v$, dann können alle Kanten (\cdot, u, i) in (\cdot, v, i) umgewandelt werden, d.h. alle eingehenden Kanten von u werden zu eingehenden Kanten von v . u kann anschließend entfernt werden (vgl. Abbildung 4.4).
- (ii) *Merging rule:* Wenn für zwei Knoten $u, v \in V$ gilt $u.\text{label} = v.\text{label}$ und $u_0 = v_0$ sowie $u_1 = v_1$, dann können u und v zusammengefasst werden, d.h. alle Kanten (\cdot, v, i) werden in (\cdot, u, i) umgewandelt und v wird entfernt (vgl. Abbildung 4.5).

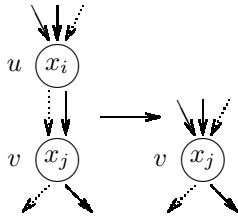


Abbildung 4.4: *elimination rule*

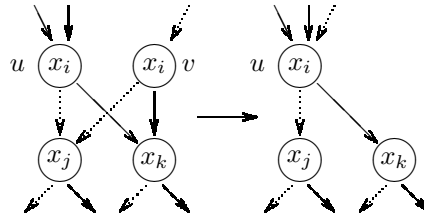


Abbildung 4.5: *merging rule*

Offensichtlich ist ein Graph G' , der durch Anwendung der *elimination rule* oder der *merging rule* aus einem G_0 -FBDD G erzeugt wurde, ebenfalls ein G_0 -FBDD und G' repräsentiert dieselbe Funktion wie G . Die Nichtanwendbarkeit dieser beiden Regeln und die Erreichbarkeit aller Knoten von der Wurzel sind ein notwendiges und hinreichendes Kriterium für die Minimalität eines G_0 -FBDDs (vgl. [Weg00], Theorem 6.4.5):

Lemma 36. Für einen G_0 -FBDD $G_f = (V, E)$, der eine Boolesche Funktion $f \in B_n$ repräsentiert und in dem alle Knoten $v \in V$ von der Wurzel aus erreichbar sind, gilt: G_f ist genau dann reduziert, wenn weder die *merging rule* noch die *elimination rule* anwendbar ist. \square

Basierend auf dieser Aussage zeigt [Weg00] die Existenz eines Linearzeitalgorithmus für die Minimierung eines gegebenen G_0 -FBDDs:

Lemma 37. Aus einem G_0 -FBDD G_f für eine Boolesche Funktion $f \in B_n$ kann in einer Laufzeit und mit einem Speicherplatzbedarf von $O(|G_f|)$ der reduzierte G_0 -FBDD G_f^* für f bestimmt werden. \square

Dieser Algorithmus soll an dieser Stelle nicht im Detail betrachtet werden¹, da sich die Minimierung in die im folgenden Abschnitt dargestellte Syntheseoperation integrieren lässt.

4.5.2 m -äre Synthese

Wir betrachten aus Gründen der Übersichtlichkeit zunächst den Spezialfall der binären Synthese, der sich leicht auf den m -ären Fall verallgemeinern lässt.

¹Für eine ausführlichere Darstellung vgl. [Ste03]

Es sei $G_0 = (V, E)$ ein Steuerungsgraph und $G_1 = (V_1, E_1)$, $G_2 = (V_2, E_2)$ seien G_0 -FBDDs über der Variablenmenge $X_n = \{x_1, \dots, x_n\}$. G_1 repräsentiere die Boolesche Funktion $f_1 \in B_n$ und G_2 die Funktion $f_2 \in B_n$. Die Synthese von G_1 und G_2 bezüglich einer Operation $\otimes \in B_2$ muss einen G_0 -FBDD G erzeugen, der die Funktion $f = f_1 \otimes f_2$, d.h. $f(a) = f_1(a) \otimes f_2(a)$ für $a \in \{0, 1\}^n$, repräsentiert.

Daher liegt es nahe, G *bottom-up* mittels simultaner Tiefensuche in G_0 , G_1 und G_2 zu berechnen und für jede partielle Eingabe $a \in \{0, 1\}^m$, $m \leq n$, den in G_0 , G_1 und G_2 erreichten Knoten zu speichern. Werden durch eine Eingabe a in G_1 und G_2 jeweils Senken erreicht, kann der Funktionswert $f(a) = f_1(a) \otimes f_2(a)$ anhand der Bezeichnungen der Senken bestimmt werden.

Es seien $\tilde{v}^0, \tilde{v}^1, \tilde{v}^2$ die Quellen von G_0 , G_1 bzw. G_2 . Ebenso bezeichnen s^0 sowie s_0^1, s_1^1 und s_0^2, s_1^2 die Senken in G_0 , G_1 bzw. G_2 . Wie von [SW95] vorgeschlagen definiert man $G := (V, E)$ mit $V := V' \cup \{(s_0, -, -), (s_1, -, -)\}$, wobei für die Knotenmenge V' gilt

$$V' \subseteq (V_0 \setminus \{s^0\}) \times (V_1 \setminus \{s_0^1, s_1^1\}) \times (V_2 \setminus \{s_0^2, s_1^2\})$$

Die Quelle von G entspricht dem Knoten $v = (\tilde{v}^0, \tilde{v}^1, \tilde{v}^2)$. Für die Bezeichnung jedes Knotens $v = (v^0, v^1, v^2) \in V \setminus \{(s_0, -, -), (s_1, -, -)\}$ wählt man $v.label := v^0.label$. Den c -Nachfolger $v_c = (v^0, v^1, v^2)_c$, $c \in \{0, 1\}$, von v definiert man wie folgt: Falls $v_c^1 \notin \{s_0^1, s_1^1\}$ oder $v_c^2 \notin \{s_0^2, s_1^2\}$, d.h. nicht gleichzeitig in G_1 und G_2 Senken erreicht werden, setzt man

$$(v^0, v^1, v^2)_c := (v_c^0, v^1, v^2) \quad \text{mit} \quad v^i := \begin{cases} v_c^i & : v^i.label = v^0.label \\ v^i & : \text{sonst} \end{cases} \quad \text{für } i = 1, 2$$

um der Tatsache Rechnung zu tragen, dass auf den Pfaden in G_1 und G_2 nicht notwendigerweise alle Variablen aus X_n eingelesen werden. Wenn andererseits $v_c^1 = s_i^1$ und $v_c^2 = s_j^2$ für ein $i \in \{0, 1\}$ und ein $j \in \{0, 1\}$ gilt, d.h. in G_1 und in G_2 Senken erreicht werden, definiert man $(v^0, v^1, v^2)_c := (s_w, -, -)$ mit $w = s_i^1.label \otimes s_j^2.label$.

Offensichtlich ist G ein G_0 -FBDD, und G berechnet die Funktion $f_1 \otimes f_2$.

Die vollständig berechneten Knoten, d.h. die Knoten (v^0, v^1, v^2) , für die die Tiefensuche abgeschlossen ist, werden in einer *computed-table* gespeichert, um die mehrfache Berechnung desselben Teilgraphen zu vermeiden.

Da G *bottom-up* berechnet wird, gilt für alle Knoten auf den Pfaden von einem vollständig berechneten Knoten $v \in V$ zu einer Senke, dass diese Knoten ebenfalls vollständig berechnet sind. Damit kann direkt nach Abschluss der Traversierung des Teilgraphen mit der Wurzel v geprüft werden, ob v durch Anwendung der *elimination rule* gelöscht werden kann. Dies ist durch einen Vergleich von v_0 und v_1 in elementarer Weise möglich.

Zusätzlich kann die Anwendbarkeit der *merging rule* beurteilt werden, wenn in einer *unique-table* für jedes Tripel $(l, u_0, u_1) \in X_n \times V \times V$ ein Repräsentant $u \in V$ gespeichert wird. Falls ein solcher Repräsentant für v existiert, wird v direkt mit u verschmolzen, andernfalls wird v als Repräsentant für das Tripel $(v.label, v_0, v_1)$ der *unique-table* hinzugefügt.

Nach Abschluss der Berechnung von G und dem Entfernen aller nicht erreichbaren Knoten ist also weder die *merging rule* noch die *elimination rule* auf G anwendbar. Lemma 34 impliziert somit die Minimalität von G .

Die dargestellte Vorgehensweise ist in den Algorithmen 2 und 3 nochmals formalisiert.

Insgesamt folgt also:

Lemma 38. *Die binäre Synthese bezüglich einer Operation $\otimes \in B_2$ von zwei Funktionen $f, g \in B_n$,*

die durch die G_0 -FBDDs G_f und G_g repräsentiert werden, sowie die Reduktion des entstehenden G_0 -FBDDs G können in einer Laufzeit und mit einem Speicherplatzbedarf von $O(|G_0||G_f||G_g|)$ durchgeführt werden. \square

Algorithmus 2 SYNTH (G_0, G_1, G_2, \otimes)

```

computed – table  $\leftarrow$  empty lookup-table
unique – table  $\leftarrow$  empty lookup-table
G.source  $\leftarrow$  SYNTH-VISIT( $G_0$ .source,  $G_1$ .source,  $G_2$ .source)

```

Algorithmus 3 SYNTH-VISIT((u^0, u^1, u^2))

```

if  $u^1 = s_i^1 \wedge u^2 = s_j^2$  then // sink reached
  return  $(s_{i \otimes j}, -, -)$ 
 $v' \leftarrow$  computed – table.lookup( $(u^0, u^1, u^2)$ )
if  $v' \neq \text{NIL}$  then
  return  $v'$  // already computed
 $v \leftarrow (u^0, u^1, u^2)$ 
 $v$ .label  $\leftarrow v^0$ .label
for all  $c \in \{0, 1\}$  do
  for all  $i \in \{1, 2\}$  do // compute successors in  $G_1$  and  $G_2$ 
    if  $v^i$ .label =  $v^0$ .label then
       $v^{i,c} \leftarrow v_c^i$ 
    else // test of  $V^0$ .label omitted in  $G_i$ 
       $v^{i,c} \leftarrow v^i$ 
   $v_c \leftarrow$  SYNTH-VISIT( $v_c^0, v^{i,c}, v^{j,c}$ )
if  $v_0 = v_1$  then // elimination rule applicable
  computed – table.put( $(u^0, u^1, u^2), v_0$ )
  return  $v_0$ 
 $v' \leftarrow$  unique – table.lookup( $v^0$ .label,  $v_1, v_0$ )
if  $v' = \text{NIL}$  then // no representative present
  unique – table.put( $(v^0$ .label,  $v_1, v_0), v$ )
   $v' \leftarrow v$ 
computed – table.put( $(u^0, u^1, u^2), v'$ )
return  $v'$ 

```

Offensichtlich lassen sich die zur binären Synthese angestellten Überlegungen unmittelbar auf die Synthese von m Funktionen übertragen:

Korollar 39. Die m -äre Synthese bezüglich einer Operation $\otimes \in B_m$ von m Funktionen f_1, \dots, f_m , die durch die G_0 -FBDDs G_1, \dots, G_m repräsentiert werden, sowie die Reduktion des entstehenden G_0 -FBDDs G können in einer Laufzeit und mit einem Speicherplatzbedarf von $O(|G_0| \prod_{i=1}^m |G_i|)$ durchgeführt werden. \square

4.5.3 SAT-ENUM

Zur Bestimmung der erfüllenden Variablenbelegungen einer als G_0 -FBDD G_f gegebenen Booleschen Funktion f über $X_n = \{x_1, \dots, x_n\}$ werden zunächst mittels Tiefensuche die Pfade von der Wurzel zur 1-Senke in G_f bestimmt.

Es seien x_{i_1}, \dots, x_{i_k} die auf einem solchen Pfad p eingelesenen Variablen und $b_{i_1}, \dots, b_{i_k} \in \{0, 1\}$ die durch p bestimmten Belegungen dieser Variablen. Wegen der *read-once* Eigenschaft von G_f (Bemerkung 32)

gilt $k \leq n$. Zu p korrespondieren alle Eingaben $a \in \{0, 1\}^n$ mit $a_{i_j} = b_{i_j}$ für $1 \leq j \leq k$, d.h. die auf dem Pfad p nicht eingelesenen Variablen können beliebige Werte annehmen. Durch systematisches Durchlaufen aller Belegungskombinationen dieser Variablen können schließlich alle zu p korrespondierenden und damit erfüllenden Variablenbelegungen erzeugt werden.

Für den Laufzeit- und Speicherplatzbedarf dieses Algorithmus ergibt sich:

Lemma 40. *Für eine Funktion $f \in B_n$, die als FBDD G_f gegeben ist, können die erfüllenden Variablenbelegungen, d.h. diejenigen $a \in \{0, 1\}^n$ mit $f(a) = 1$, in einer Laufzeit und mit einem Speicherplatzbedarf von $O(n \cdot |One(G_f)|)$ berechnet werden.*

Beweis. Es existieren in G_f höchstens $|One(G_f)|$ Pfade von der Wurzel zur 1-Senke, die mittels Tiefensuche in Laufzeit und Speicherplatz $O(|G_f|)$ bestimmt werden können (vgl. [CLRS01], S. 540ff). Um die erfüllenden Belegungen auszugeben, werden Zeit und Speicherplatz $O(n \cdot |One(G_f)|)$ benötigt. Aus Lemma 33 folgt insgesamt für den Laufzeit- und Speicherplatzbedarf

$$O(|G_f| + n \cdot |One(G_f)|) = O(n \cdot |One(G_f)|) \quad \square$$

Kapitel 5

Kryptanalyse mit Hilfe von FBDDs

5.1 Repräsentation der Sprachen U_m , W_m und T_m durch FBDDs

Die Effizienz des generischen Algorithmus zur Bestimmung des Initialzustands x (Algorithmus 1) hängt neben der Anzahl der Schleifendurchläufe maßgeblich von der effizienten Repräsentation der Sprachen U_m , W_m und T_m ab. [Kra02] verwendet hierzu die folgenden FBDD-Konstruktionen:

Definition 41. Für $m > 0$ sei G_m^C der Steuerungsgraph, der für jeden internen Bitstrom $z \in \{0, 1\}^m$ die Reihenfolge definiert, in der die Bits von z durch die Kompressionsfunktion C eingelesen werden.

Definition 42. Für $m > 0$ sei R_m der minimale G_m^C -FBDD, der für $z \in \{0, 1\}^m$ entscheidet, ob $z = L_{\leq m}(i_m(L, z))$ gilt.

Definition 43. Für $m > 0$ sei S_m der minimale G_m^C -FBDD, der für $z = (z_0, \dots, z_{m-1}) \in \{0, 1\}^m$ entscheidet, ob $z_{m-1} = L_{m-1}(i_m(L, z))$ gilt.

Bemerkung 44. Im Allgemeinen ist es möglich, dass nicht alle Bits des internen Bitstroms z tatsächlich von der Kompressionsfunktion C für die Berechnung der Schlüsselbits verarbeitet werden. Abweichend von der formalen Definition eines Steuerungsgraphen (vgl. Definition 29) existieren damit in G_m^C Pfade, auf denen nicht alle Bits z_i eingelesen werden. Die Korrektheit der in diesem Kapitel dargestellten Überlegungen wird dadurch jedoch nicht beeinflusst.

Definition 45. Für $m > 0$ sei Q_m der minimale G_m^C -FBDD, der für $z \in \{0, 1\}^m$ entscheidet, ob $C(z)$ Präfix von y ist.

Definition 46. Für $m > 0$ sei P_m der minimale G_m^C -FBDD, der für $z \in \{0, 1\}^m$ entscheidet, ob $C(z)$ Präfix von y ist und ob $z = L_{\leq m}(i_m(L, z))$ gilt.

Für U_m, V_m, W_m, T_m aus Definition 17 gilt offensichtlich $\text{One}(R_m) = U_m$, $\text{One}(S_m) = V_m$, $\text{One}(Q_m) = W_m$, $\text{One}(P_m) = T_m$.

Der generische Algorithmus 1 kann mit Hilfe der G_m^C -FBDDs P_m , Q_m und S_m demnach wie folgt in einen FBDD-Algorithmus transformiert werden.

Offensichtlich gilt $P = P_m$ nach jeder Iteration m . Ferner impliziert die Definition der Kompressionsfunktion C , dass für $m' \geq m$ jeder G_m^C -FBDD auch ein $G_{m'}^C$ -FBDD ist. Hieraus folgt die Korrektheit der Operation $\min(P \wedge Q_m \wedge S_m)$.

Algorithmus 4 FBDD-COMPUTE-x

```

 $P \leftarrow Q_{n'}$ 
for  $m \leftarrow n' + 1$  to  $\lceil \alpha^{-1}n \rceil$  do
   $P \leftarrow \min(P \wedge Q_m \wedge S_m)$ 
return  $i_m(L, z^*)$  for a  $z^* \in \text{One}(P)$ 

```

Für die Laufzeit $t(n)$ der *for*-Schleife in Algorithmus 4 gilt gemäß Korollar 39:

$$t(n) \leq \sum_{m=n'+1}^{\lceil \alpha^{-1}n \rceil} |G_m^C| |P_{m-1}| |Q_m| |S_m|$$

Für eine genauere Laufzeitanalyse benötigen wir daher Größenabschätzungen für den Steuerungsgraphen G_m^C und die G_m^C -FBDDs P_m , Q_m und S_m .

Bezüglich G_m^C und Q_m soll zunächst die folgende Annahme getroffen werden:

Annahme 4 (FBDD-Annahme). Für die Kompressionsfunktion C und alle $m \geq n'$ gilt

$$|G_m^C| \in m^{O(1)} \text{ und } |Q_m| \in m^{O(1)}$$

d.h. die Größen des Steuerungsgraphen G_m^C und des G_m^C -FBDDs Q_m sind polynomiell in m .

Wir werden in Kapitel 7 nachweisen, dass der A5/1 Schlüsselstromgenerator diese Annahme tatsächlich erfüllt.

5.2 Bestimmung von $|R_m|$ und $|S_m|$

Lemma 47. Es sei L ein linearer Bitstromgenerator bestehend aus k LFSR L^0, \dots, L^{k-1} . Für $r \in \{0, \dots, k-1\}$ besitze das LFSR L^r die Länge n_r und den Initialzustand x^r . Ferner sei $n := \sum_{r=0}^{k-1} n_r$. Dann existiert für $m > n'$ ein G_m^C -FBDD R_m mit $|R_m| \leq |G_m^C| 2^{m-|I_m(L)|}$, der für ein $z \in \{0, 1\}^m$ entscheidet, ob $z = L_{\leq m}(i_m(L, z))$.

Beweis. Es seien $r(i) := i \bmod k$ und $s(i) := i \operatorname{div} k$ für $i \in \{0, \dots, m-1\}$. Gemäß Definition 8 kann ein Bit z_i eines Bitstroms

$$z = L_{\leq m}(x) = (z_0, \dots, z_{m-1})$$

dargestellt werden als $z_i = L_{s(i)}^{r(i)}(x^{r(i)})$. Ferner ist wegen Definition 9 das Bit z_i genau dann ein Initialzustandsbit, wenn $i \in I_m(L)$ und genau dann ein linear kombiniertes Bit, wenn $i \in C_m(L)$.

Für die linear kombinierten Bits gilt gemäß Beobachtung 3

$$z_i = \bigoplus_{j=0}^{n_{r(i)}-1} L_{j, s(i)}^{r(i)} \cdot x_j^{r(i)} \text{ für alle } i \in C_m(L) \quad (5.1)$$

Um zu überprüfen, ob ein gegebener Bitstrom $z = (z_0, \dots, z_{m-1})$ durch L erzeugt werden kann, d.h. um zu entscheiden, ob $z = L_{\leq m}(i_m(L, z))$ gilt, verwalten wir für alle $i \in C_m(L)$ den Wert

$$b_i := \bigoplus_{j=0}^{n_{r(i)}-1} L_{j, s(i)}^{r(i)} \cdot z_{k \cdot j + r(i)}$$

Der Bitstrom z kann genau dann durch L erzeugt werden, wenn $z_i = b_i$ für alle $i \in C_m(L)$ gilt, d.h. wenn die linear kombinierten Bits mit den aus den Initialzustandsbits berechneten Bits übereinstimmen.

Ein Algorithmus, der die Bits z_0, \dots, z_{m-1} in der durch G_m^C festgelegten Reihenfolge einliest, verfährt in folgender Weise: Immer wenn ein z_i mit $i \in I_m(L)$ gelesen wurde, wird jeder Vergleichswert b_j mit $j \in C_m(L)$ und $r(j) = r(i)$ durch

$$b_j := b_j \oplus L_{s(i),s(j)}^{r(i)} \cdot z_i$$

aktualisiert, d.h. der von z_i abhängige Summand wird zu b_j hinzuaddiert. Falls $i \in C_m(L)$ wird geprüft, ob $z_i = b_i$ erfüllt ist. Die Korrektheit dieser Operation folgt aus der Tatsache, dass gemäß Definition 12 die Ausgabebits eines fixierten LFSR in der Reihenfolge ihrer Produktion gelesen werden. Im positiven Fall stimmen der Wert, der aus dem in z enthaltenen Initialzustand berechnet wurde, und der tatsächliche Wert überein. Der Algorithmus fährt dann mit dem nächsten Bit des Bitstroms z fort. Falls $z_i \neq b_i$ gilt, kann der Bitstrom z nicht durch L erzeugt worden sein, so dass die Überprüfung mit der Ausgabe 0 abbricht. Diese Vorgehensweise ist in Algorithmus 5 nochmals formalisiert.

Algorithmus 5 *test – linearity*(z, G_m^C)

```

b = ( $b_{j_1}, \dots, b_{j_{|C_m(L)|}}$ )  $\leftarrow \vec{0}$  where  $j_l \in C_m(L)$  for all  $l \in \{1, \dots, |C_m(L)|\}$ 
Let  $\pi_{G_m^C}(z)$  be the reading-order of the  $z_i, i \in \{0, \dots, m-1\}$ , as defined by  $G_m^C$ 
for  $l \leftarrow 0$  to  $m-1$  do
   $i \leftarrow (\pi_{G_m^C}(z))(l)$ 
  if  $i \in I_m(L)$  then
    for all  $j \in \{j \in C_m(L) | r(j) = r(i)\}$  do
       $b_j \leftarrow b_j \oplus L_{s(i),s(j)}^{r(i)} \cdot z_i$ 
    else //  $i \in C_m(L)$ 
      if  $b_i \neq z_i$  then
        stop(0)
  stop(1)

```

Algorithmus 5 kann in folgender Weise in einen G_m^C -FBDD R_m transformiert werden: Die Knotenmenge $V(R_m)$ von R_m sei definiert als

$$V(R_m) \subseteq \{(v, b) | v \in V(G_m^C) \wedge b \in \{0, 1\}^{|C_m(L)|}\} = V(G_m^C) \times \{0, 1\}^{|C_m(L)|}$$

Die Wurzel von R_m sei der Knoten $(G_m^C.root, \vec{0})$, wobei $G_m^C.root$ die Wurzel von G_m^C bezeichnet. Für alle $(v, b) \in V(R_m)$ gelte $(v, b).label := v.label$. Die Kantenmenge $E(R_m)$ sei wie folgt definiert: Für einen Knoten $(v, b) \in V(R_m)$ mit $b = (b_{j_1}, \dots, b_{j_{|C_m(L)|}})$ und $j_l \in C_m(L)$ für alle $l \in \{1, \dots, |C_m(L)|\}$ bezeichne $v_c, c \in \{0, 1\}$, den c -Nachfolger von v in G_m^C und $(v, b)_c$ den c -Nachfolger von (v, b) .

Zur Definition von $(v, b)_c$ unterscheiden wir zwei Fälle:

1. Fall: $v.label = z_i, i \in I_m(L)$

$$(v, b)_c := (v_c, b \oplus b') \text{ mit } b' = (b'_{j_1}, \dots, b'_{j_{|C_m(L)|}}) \text{ und } b'_{j_i} = \begin{cases} L_{s(i),s(j_i)}^{r(i)} \cdot z_i & \text{für } r(j_i) = r(i) \\ 0 & \text{sonst} \end{cases}$$

2. Fall: $v.label = z_j, j \in C_m(L)$

$$(v, b)_c := \begin{cases} 0 - sink & \text{für } b_j \neq c \\ \begin{cases} 1 - sink & \text{für } v_c = 1 - sink \\ (v_c, b) & \text{für } v_c \neq 1 - sink \end{cases} & \text{für } b_j = c \end{cases}$$

Offensichtlich berechnet R_m die Sprache $\{z \in \{0,1\}^m \mid z = L_{\leq m}(i_m(L, z))\}$. Da höchstens $2^{|C_m(L)|}$ verschiedene Belegungen des Vektors b existieren, gilt nach dem Entfernen nicht erreichbarer Knoten $|R_m| = |V(R_m)| \leq |G_m^C| 2^{|C_m(L)|} \leq |G_m^C| 2^{m-|I_m(L)|}$. \square

Im Unterschied zum G_m^C -FBDD R_m prüft der G_m^C -FBDD S_m nicht für alle z_j , $j \in C_m(L)$, sondern nur für z_{m-1} , ob $z_{m-1} = L_{m-1}(i_m(L, z))$ gilt. S_m kann daher aus R_m konstruiert werden, indem nur der Vergleichswert b_{m-1} und nicht alle b_j für $j \in C_m(L)$ verwaltet und aktualisiert werden. Aus Lemma 47 folgt somit für die Größe von S_m unmittelbar:

Korollar 48. *Für einen beliebigen linearen Bitstromgenerator L und alle $m > 0$ existiert ein G_m^C -FBDD S_m mit $|S_m| \leq 2|G_m^C|$, der für ein $z = (z_0, \dots, z_{m-1}) \in \{0,1\}^m$ entscheidet, ob $z_{m-1} = L_{m-1}(i_m(L, z))$ gilt.* \square

5.3 Bestimmung von $|P_m|$

Mit Hilfe der Größenabschätzungen für G_m^C und die G_m^C -FBDDs Q_m , R_m und S_m können wir nun die Größe von P_m bestimmen.

Lemma 49. *Falls L die FBDD-Annahme erfüllt, gilt $|P_m| \leq n^{O(1)} 2^{\frac{1-\alpha}{1+\alpha}n}$ für alle $n' \leq m \leq \lceil \alpha^{-1}n \rceil$.*

Beweis. Die Definitionen von P_m , R_m und Q_m implizieren $P_m = R_m \wedge Q_m$ für $n' \leq m \leq \lceil \alpha^{-1}n \rceil$. Wegen Korollar 39 gilt daher

$$|P_m| \leq |G_m^C| |R_m| |Q_m|$$

Zur Vereinfachung der Notation sei $n^* := |I_m(L)|$. Mit Hilfe von Lemma 47 und Annahme 4 sowie $P_{n'} = Q_{n'} \in n'^{O(1)}$ ergibt sich

$$|P_m| \leq \underbrace{|G_m^C|^2}_{\in m^{O(1)}} 2^{m-n^*} \underbrace{|Q_m|}_{\in m^{O(1)}} \leq p(m) \cdot 2^{m-n^*} \quad (5.2)$$

mit $p(m) = |G_m^C|^2 \cdot |Q_m|$.

Andererseits folgt aus Lemma 33, dass $|P_m| \leq m \cdot |One(P_m)|$. In Verbindung mit $|One(P_m)| \approx 2^{n^* - \alpha m}$ (Lemma 19) gilt damit für alle $n' \leq m \leq \lceil \alpha^{-1}n \rceil$

$$|P_m| \leq m |One(P_m)| \approx m \cdot 2^{n^* - \alpha m} = m \cdot 2^{(1-\alpha)n^* - \alpha(m-n^*)} \quad (5.3)$$

Aus (5.2) und (5.3) folgt für $n' \leq m \leq \lceil \alpha^{-1}n \rceil$

$$\begin{aligned} |P_m| &\leq \min \left\{ p(m) \cdot 2^{m-n^*}, m \cdot 2^{(1-\alpha)n^* - \alpha(m-n^*)} \right\} \\ &= \min \left\{ p(m) \cdot 2^{r(n^*)}, m \cdot 2^{(1-\alpha)n^* - \alpha r(n^*)} \right\} \text{ mit } r(n^*) := m - n^* \\ &\leq \min \left\{ p(m) \cdot 2^{r(n^*)}, p(m) \cdot 2^{(1-\alpha)n^* - \alpha r(n^*)} \right\} \text{ wegen } m \leq p(m) \\ &\leq p(m) \cdot 2^{r^*(n^*)} \end{aligned}$$

wobei $r^*(n^*)$ die Lösung von $2^{r(n^*)} = 2^{(1-\alpha)n^* - \alpha r(n^*)}$ bezeichnet. Auflösen ergibt $r^*(n^*) = \frac{1-\alpha}{1+\alpha}n^*$.

Wegen $n' \leq m \leq \lceil \alpha^{-1}n \rceil$ und der FBDD-Annahme ist $p(m) \in n^{O(1)}$, und mit $n^* = |I_m(L)| \leq n$ folgt schließlich

$$|P_m| \leq n^{O(1)} 2^{\frac{1-\alpha}{1+\alpha}n} \text{ für alle } n \leq m \leq \lceil \alpha^{-1}n \rceil \quad \square$$

5.4 Gesamtresultat

Wir kennen nun mit der Anzahl der Schleifendurchläufe und der Größe der beteiligten FBDDs die entscheidenden Einflussgrößen der Laufzeit und des Speicherplatzbedarfs von Algorithmus 4 und erhalten:

Korollar 50. *Für die Laufzeit $t(n)$ und den Speicherplatzbedarf $s(n)$ der for-Schleife in Algorithmus 4 gilt*

$$t(n), s(n) \leq n^{O(1)} 2^{\frac{1-\alpha}{1+\alpha}n}$$

Beweis. Wegen Korollar 39 besitzt eine einzelne Synthese- und Minimierungsoperation eine Laufzeit und einen Speicherplatzbedarf von höchstens $|G_m^C| |P_{m-1}| |Q_m| |S_m|$. Aus der FBDD-Annahme (Annahme 4) und Korollar 48 ergibt sich

$$t(n) \leq \sum_{m=n'+1}^{\lceil \alpha^{-1}n \rceil} |P_{m-1}| \underbrace{|G_m^C| |Q_m| |S_m|}_{\leq m^{O(1)}} \leq \sum_{m=n'+1}^{\lceil \alpha^{-1}n \rceil} m^{O(1)} |P_{m-1}| \leq n^{O(1)} \max_{n'+1 \leq m \leq \lceil \alpha^{-1}n \rceil} \{|P_m|\}$$

Aus $|P_m| \leq n^{O(1)} 2^{\frac{1-\alpha}{1+\alpha}n}$ für alle $n' \leq m \leq \lceil \alpha^{-1}n \rceil$ (Lemma 49) folgt schließlich

$$t(n) \leq n^{O(1)} 2^{\frac{1-\alpha}{1+\alpha}n}$$

Die Argumentation für $s(n)$ verläuft analog. □

Für den Startwert $Q_{n'}$ gilt $|Q_{n'}| \leq n^{O(1)}$ gemäß Annahme 4, und die Bestimmung der erfüllenden Belegungen für $P_{\lceil \alpha^{-1}n \rceil}$ ist gemäß Lemma 40 möglich in einer Laufzeit und mit einem Speicherplatzbedarf von

$$O(\lceil \alpha^{-1}n \rceil \cdot \text{One}(P_{\lceil \alpha^{-1}n \rceil})) \approx O(\lceil \alpha^{-1}n \rceil \cdot 1) = O(\lceil \alpha^{-1}n \rceil)$$

d.h. die Laufzeit und der Speicherplatzbedarf der übrigen Operationen in Algorithmus 4 sind polynomiell in n .

Wir erhalten somit das folgende Gesamtresultat:

Theorem 51. *Es sei $K = (L, C)$ ein LFSR-basierter Schlüsselstromgenerator der Schlüssellänge n mit der Informationsrate α und der best case Kompressionsrate γ . Falls K die Unabhängigkeitsannahme, die Pseudozufälligkeitsannahme und die FBDD-Annahme erfüllt, kann der Initialzustand x in einer Laufzeit und mit einem Speicherplatzbedarf von $n^{O(1)} 2^{\frac{1-\alpha}{1+\alpha}n}$ aus den ersten $\lceil \gamma \alpha^{-1}n \rceil$ aufeinanderfolgenden Bits des Schlüsselstroms $y = C(L(x))$ bestimmt werden. □*

Die Laufzeit und der Speicherplatzbedarf von Algorithmus 4 werden also entscheidend von der Informationsrate α beeinflusst. Offensichtlich ist $\lim_{\alpha \rightarrow 1} \frac{1-\alpha}{1+\alpha} = 0$, so dass der Algorithmus umso effizienter ist, je mehr Information der Schlüsselstrom y über den internen Bitstrom z preisgibt.

Kapitel 6

Der Algorithmus A5/1

Der Algorithmus A5/1 wird im GSM-Mobilfunknetz (*Global System for Mobile Communications*) zur Verschlüsselung der Sprachdaten verwendet, die zwischen Mobiltelefon und Basisstation über die Luft-schnittstelle ausgetauscht werden.

Die genaue Spezifikation wurde durch die Betreiber zunächst geheimgehalten, aber [BGW99] haben durch *Reverse Engineering* einen Algorithmus gefunden, dessen Korrektheit laut [BSW01] von der GSM-Organisation bestätigt worden ist. Definitiv bekannt - weil durch die Betreiber veröffentlicht - ist lediglich das während eines Telefonats verwendete allgemeine Kommunikations- und Sicherheitsprotokoll, das im Folgenden zunächst kurz skizziert werden soll.

6.1 Verschlüsselung der Sprachdaten

Zur Übermittlung der Sprachdaten versenden das Mobiltelefon und die Basisstation abwechselnd verschlüsselte Datenpakete mit einer Nutzdatenmenge von 114 Bits, wobei Pakete von der Basisstation zum Mobiltelefon (*downstream*) und darauffolgende Pakete vom Mobiltelefon zur Basisstation (*upstream*) jeweils zu einem (verschlüsselten) Übertragungsrahmen der Länge $114 + 114 = 228$ Bits zusammengefasst werden (vgl. Abbildung 6.1)

Ein solcher Übertragungsrahmen $f \in \{0, 1\}^{228}$ kann also dargestellt werden als

$$f = (\underbrace{e_0^b, \dots, e_{113}^b}_{e^b}, \underbrace{e_0^m, \dots, e_{113}^m}_{e^m})$$

wobei $(e_0^b, \dots, e_{113}^b) =: e^b$ den *downstream* Chiffretext und $(e_0^m, \dots, e_{113}^m) =: e^m$ den *upstream* Chiffretext bezeichnen. Für jeden Rahmen erzeugen das Mobiltelefon und die Basisstation mit Hilfe des A5/1 Schlüsselstromgenerators einen Schlüsselstrom

$$y = (\underbrace{y_0, \dots, y_{113}}_{y^b}, \underbrace{y_{114}, \dots, y_{227}}_{y^m}) \in \{0, 1\}^{228}$$

Die Basisstation berechnet e^b aus dem zu versendenden *downstream* Klartext $p^b := (p_0^b, \dots, p_{113}^b)$ durch bitweises XOR mit den ersten 114 Bits von y und entschlüsselt den empfangenen *upstream* Chiffretext

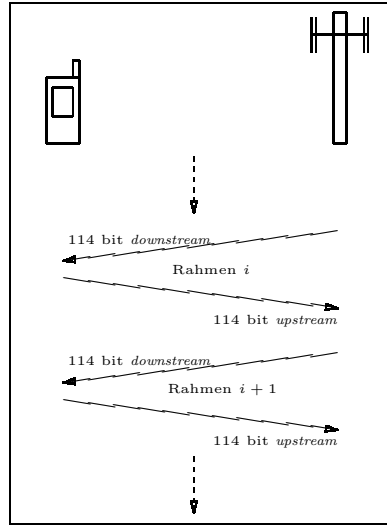


Abbildung 6.1: Kommunikation im GSM-Mobilfunknetz

e^m durch bitweises XOR mit den letzten 114 Bits von y , d.h.

$$\begin{aligned} p^b \oplus y^b &= e^b \\ e^m \oplus y^m &= p^m \end{aligned}$$

Entsprechend entschlüsselt das Mobiltelefon mit der ersten Hälfte von y den von der Basisstation übermittelten *downstream* Chiffretext und verschlüsselt mit der zweiten Hälfte von y den für die Basisstation bestimmten *upstream* Klartext $p^m := (p_0^m, \dots, p_{113}^m)$, d.h.

$$\begin{aligned} e^b \oplus y^b &= p^b \\ p^m \oplus y^m &= e^m \end{aligned}$$

6.2 Aufbau des A5/1 Schlüsselstromgenerators

Der A5/1 Schlüsselstromgenerator besteht aus den drei LFSR R_0 , R_1 und R_2 der Länge n_0 , n_1 bzw. n_2 sowie einer Taktkontrolle, die eine lineare Abhängigkeit der Schlüsselbits vom Initialzustand der LFSR verhindert. Als Eingabe für die Taktkontrolle wird in jedem LFSR R_r , $r \in \{0, 1, 2\}$, ein Zustandsbit $q_{N^k}^r$ mit $N^r \in \{0, \dots, n_r - 1\}$ als Kontrollbit festgelegt (vgl. Abbildung 6.2).

In der von [BGW99] angegebenen Version des Algorithmus gilt

$$(n_0, n_1, n_2) = (19, 22, 23) \quad \text{sowie} \quad (N^0, N^1, N^2) = (11, 12, 13)$$

Der innere Zustand $q(t)$ des Schlüsselstromgenerators im Zeitpunkt t , d.h. vor der Produktion des Schlüsselbits y_t , ist zusammengesetzt aus den inneren Zuständen der drei LFSR zu diesem Zeitpunkt, d.h.

$$q(t) = (q^0(t), q^1(t), q^2(t))$$

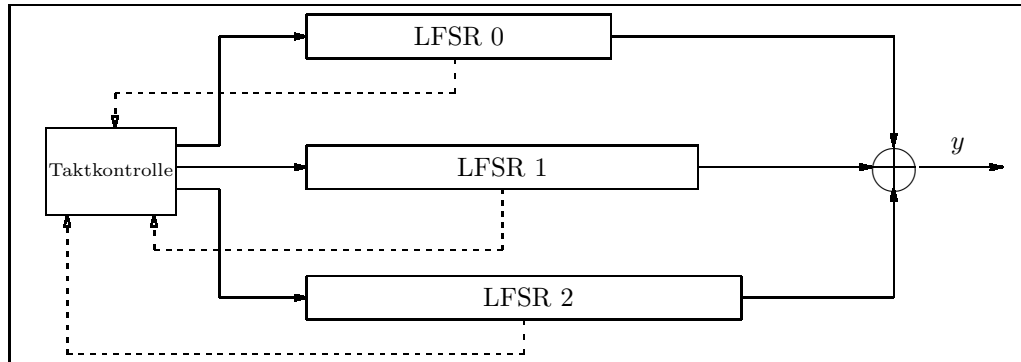


Abbildung 6.2: Aufbau des A5/1 Schlüsselstromgenerators

6.3 Arbeitsweise des A5/1 Schlüsselstromgenerators

Für jeden Übertragungsrahmen wird aus einem zu Beginn des Gesprächs generierten geheimen Sitzungsschlüssel Kc der Länge 64 Bit und der öffentlich zugänglichen Rahmennummer F_n ($|F_n|=22$ Bit) ein geheimer Rahmenschlüssel $Kf = fKey(Kc, F_n)$ ($|Kf| = 64$ Bit) erzeugt, indem Kc und F_n in bestimmter Weise in die drei LFSR eingespeist werden. Kf ergibt sich dann als der Initialzustand $q(0)$, d.h. als der innere Zustand unmittelbar vor der Produktion des ersten Schlüsselbits.

In jeder Iteration bestimmt der Algorithmus zunächst aus den Ausgabebits der drei LFSR das Schlüsselbit y_t , $t \geq 0$, als

$$y_t := q_0^0 \oplus q_0^1 \oplus q_0^2$$

Anschließend betrachtet der Algorithmus die Kontrollbits $q_{N^0}^0$, $q_{N^1}^1$ und $q_{N^2}^2$ und taktet das LFSR R_r genau dann, wenn

$$q_{N^r}^r = \text{maj}_3(q_{N^0}^0, q_{N^1}^1, q_{N^2}^2) \quad (6.1)$$

Hierbei ist die Funktion maj_3 definiert durch

$$\begin{aligned} \text{maj}_3 : \{0, 1\}^3 &\rightarrow \{0, 1\} \\ (a, b, c) &\mapsto \begin{cases} 1 & \text{falls } a + b + c \geq 2 \\ 0 & \text{sonst} \end{cases} \end{aligned}$$

d.h. maj_3 gibt genau dann p zurück, wenn mindestens zwei der drei Argumente den Wert p besitzen.

Weil (6.1) immer für mindestens zwei $r \in \{0, 1, 2\}$ erfüllt ist, werden in jeder Iteration mindestens zwei der drei LFSR getaktet.

Kapitel 7

Kryptanalyse des A5/1 Schlüsselstromgenerators

7.1 Simulation des Algorithmus A5/1 durch einen LFSR-basier- ten Schlüsselstromgenerator

7.1.1 Vorbemerkungen

Um das im Kapitel 3 dargestellte Kryptanalyseverfahren auf den Algorithmus A5/1 anwenden zu können, benötigen wir zunächst eine Darstellung des Algorithmus als LFSR-basierter Schlüsselstromgenerator im Sinne der Definition 12. Wir konstruieren hierzu den folgenden Schlüsselstromgenerator $K' = (L', C')$ mit dem linearen Bitstromgenerator L' und der nichtlinearen Kompressionsfunktion C' :

L' besteht aus den drei LFSR L^0 , L^1 und L^2 , die den LFSR R_0 , R_1 und R_2 des Algorithmus A5/1 entsprechen, d.h. für $r \in \{0, 1, 2\}$ entsprechen die Länge von L^r der Länge von R_r und der Initialzustand von L^r dem von R_r . L' erzeugt also den internen Bitstrom

$$z = L(x) = z_0^0, z_0^1, z_0^2, \dots, z_s^0, z_s^1, z_s^2, \dots \quad \text{mit } z_s^r = L_s^r(x^r)$$

Die Kompressionsfunktion $C' : \{0, 1\}^* \rightarrow \{0, 1\}^*$ speichert für $r = 0, 1, 2$ in $i[r]$ die aktuelle Ausgabe-
position und in $j[r]$ die aktuelle Kontrollposition im durch L^r erzeugten Bitstrom. In jeder Iteration
 t gibt C' den Wert $y_t := z_{i[0]}^0 \oplus z_{i[1]}^1 \oplus z_{i[2]}^2$ als neues Schlüsselbit aus, berechnet das neue Kontrollbit
 $p := \text{maj}_3(z_{j[0]}^0, z_{j[1]}^1, z_{j[2]}^2)$ und setzt für $r \in \{0, 1, 2\}$

$$i[r] := i[r] + 1 \text{ falls } z_{j[r]}^k = p \quad \text{sowie} \quad j[r] := j[r] + 1 \text{ falls } z_{j[r]}^k = p$$

Die Ausgabe-
positionen $i[r]$ werden mit 0 und die Kontrollpositionen $j[r]$ mit N^r initialisiert. Offensichtlich
ist die Funktion maj_3 und damit auch C' nicht linear.

Man sieht leicht, dass der von K' erzeugte Schlüsselstrom für jeden möglichen Initialzustand mit dem
Schlüsselstrom übereinstimmt, den der Algorithmus A5/1 für diesen Initialzustand berechnet.

7.1.2 Konstruktion einer Simulation mit *read-once* Kompressionsfunktion

Die im ersten Ansatz entwickelte Kompressionsfunktion C' verwendet einzelne interne Bits z_i zunächst zur Bestimmung des Kontrollbits p und später erneut zur Berechnung des Schlüsselstroms y . Um die Konstruktion des Steuerungsgraphen G_m^C , der für einen internen Bitstrom $z = (z_0, \dots, z_{m-1})$ die Einleseihenfolge der z_i durch die Kompressionsfunktion angibt, zu vereinfachen, konstruieren wir in diesem Abschnitt einen zu K' äquivalenten LFSR-basierten Schlüsselstromgenerator $K = (L, C)$, dessen Kompressionsfunktion C jedes Bit des internen Bitstroms höchstens einmal einliest (*read-once* Eigenschaft). Diese Darstellung des A5/1 Schlüsselstromgenerators werden wir für alle weiteren Betrachtungen verwenden.

Konstruktion des linearen Bitstromgenerators L

Der lineare Bitstromgenerator L erweitert L' um drei auf insgesamt sechs LFSR L^0, \dots, L^5 . Die ersten drei LFSR L^0, L^1, L^2 werden zur Berechnung der Ausgabebits verwendet und entsprechen den LFSR in L' und damit den LFSR R_0, R_1, R_2 des zu simulierenden Algorithmus A5/1. Insbesondere stimmen die Initialzustände von L^r und R_r für $r \in \{0, 1, 2\}$ überein.

Die drei zusätzlichen LFSR L^3, L^4, L^5 dienen zur Berechnung der Kontrollbits und entsprechen den um N^0, N^1 bzw. N^2 verschobenen LFSR L^0, L^1 bzw. L^2 , d.h. es gilt für $r \in \{0, 1, 2\}$, dass $n_{3+r} = n_r$ und

$$\begin{aligned} L_s^{3+r}(x) &= L_{s+N^r}^r(x) \text{ für alle } s \geq 0 \\ \text{bzw. } L_s^r(x) &= L_{s-N^r}^{3+r}(x) \text{ für alle } s \geq N^r \end{aligned}$$

Damit ist auch der Initialzustand von L^{3+r} der um N^r Positionen verschobene Initialzustand von L^r , d.h. wenn das LFSR L^r den Initialzustand $x^r = (x_0, \dots, x_{n_r-1})$ besitzt, ist der Initialzustand des LFSR L^{3+r} gegeben durch

$$\begin{aligned} x^{3+r} &= (L_{N^r}^r(x^r), \dots, L_{n_r+N^r-1}^r(x^r)) \\ &= (x_{N^r}, \dots, x_{n_r-1}, L_{n_r}^r(x^r), \dots, L_{n_r+N^r-1}^r(x^r)) \end{aligned}$$

L erzeugt also den internen Bitstrom

$$z = L(x) = z_0^0, \dots, z_0^5, \dots, z_s^0, \dots, z_s^5, \dots$$

mit $z_s^r = L_s^r(x^r)$ und $z_s^r = z_{s-N^r}^{3+r}$ für alle $r \in \{0, 1, 2\}$ und $s \geq N^r$.

Konstruktion der Kompressionsfunktion C

Im Unterschied zu C' speichert die Kompressionsfunktion C nur noch die aktuelle Leseposition $i[r]$ für $r \in \{0, 1, 2\}$ und berechnet in jeder Iteration t das Schlüsselbit y_t als

$$y_t := z_{i[0]}^0 \oplus z_{i[1]}^1 \oplus z_{i[2]}^2$$

Das neue Kontrollbit ergibt sich als

$$p := \text{maj}_3(z_{i[0]}^3, z_{i[1]}^4, z_{i[2]}^5)$$

und die aktuelle Leseposition $i[r]$ wird angepasst durch

$$i[r] := i[r] + 1 \text{ falls } z_{i[r]}^{3+r} = p$$

Auf diese Weise ist zwar sichergestellt, dass dasselbe Bit des internen Bitstroms nicht zur Kontrollbitberechnung und später erneut für die Berechnung der Schlüsselbits eingelesen wird, aber es werden noch immer einzelne interne Bits mehrfach verwendet, nämlich wenn für ein r in einer Iteration t die Leseposition $i[r]$ nicht inkrementiert wird. Das interne Bit $z_{i[r]}^r$ geht dann in die Berechnung von y_t und auch in die Berechnung von y_{t+1} ein, darf aber in Iteration $t+1$ nicht nochmals gelesen werden. Gleiches gilt für das Kontrollbit $z_{i[r]}^{3+r}$. Um die *read-once* Eigenschaft nicht zu verletzen, müssen solche Bits in geeigneter Weise zwischengespeichert werden.

Algorithmus 6 berücksichtigt diese Einschränkung und berechnet für einen gegebenen internen Bitstrom z der Länge m den Schlüsselstrom $C(z)$, ohne ein Bit z_s^r mehrfach einzulesen. Hierzu wird die Berechnung des Schlüsselstroms aufgespalten in die Berechnung der Schlüsselbits durch die Funktion *output* und in die Berechnung der Kontrollbits mit Hilfe der Funktion *control*. Aus der Definition von *maj₃* folgt, dass in jeder Iteration für höchstens ein $r \in \{0, 1, 2\}$ die Leseposition $i[r]$ nicht inkrementiert wird. Diesen Index r speichert der Algorithmus in der Variablen u (*unchanged index*) und verwaltet das zugehörige Eingabebit für die Kontrollbitberechnung $z_{i[r]}^{3+r}$ in der Variablen v (*unchanged control value*) sowie das Eingabebit für die Schlüsselbitberechnung $z_{i[r]}^r$ in der Variablen w (*unchanged output value*).

Um das nächste Schlüsselbit berechnen zu können, genügt es, die aktuellen Lesepositionen $i[0..2]$ und die Werte u und w zu kennen. Zusätzlich wird der Wert v benötigt, um nach der Berechnung des Schlüsselbits mit der Bestimmung des nächsten Kontrollbits fortfahren zu können. Damit ergibt sich die Parameterliste der Funktion *output* als (i, u, v, w) . Für die Berechnung des nächsten Kontrollbits sind zunächst die aktuellen Lesepositionen $i[0..2]$ und die Werte u und v notwendig. Darüber hinaus erhält die Funktion *control* die drei aktuellen Ausgabebits der LFSR L^0 , L^1 und L^2 als Eingabeparameter, damit nach der Kontrollbitberechnung dasjenige Ausgabebit *newW* ermittelt werden kann, das auch in die Berechnung des nächsten Schlüsselbits eingeht.

Die Ausgabebits der einzelnen LFSR werden in beiden Funktionen stets entsprechend der LFSR-Nummerierung eingelesen, d.h. wenn in einer Iteration Ausgabebits der LFSR L^{r_1} und L^{r_2} mit $r_1 < r_2$ verarbeitet werden, liest der Algorithmus stets das Ausgabebit von L^{r_1} vor dem Ausgabebit von L^{r_2} ein. Diese Konvention ist zwar nicht zwingend erforderlich, führt jedoch in der Implementation zu einem einheitlicheren Berechnungsverhalten.

Der Übersicht halber geht Algorithmus 6 von einer komponentenweisen Definition der Summe $r = p + q$ zweier Felder p und q der Länge n aus, d.h. $r[i] = (p + q)[i] := p[i] + q[i]$ für $i = 0, \dots, n - 1$.

Aus Algorithmus 6 folgt unmittelbar, dass die Anzahl der Schlüsselbits, die aus einem internen Bitstrom der Länge m berechnet werden, maximal ist, falls für jedes Schlüsselbit vier interne Bits eingelesen werden, d.h. es gilt

Beobachtung 52. Die best case Kompressionsrate γ des A5/1 Schlüsselstromgenerators beträgt $\gamma = \frac{1}{4}$.

7.2 Konstruktion des Steuerungsgraphen G_m^C

Mit Hilfe der Funktionen *output* und *control* des *read-once* Algorithmus 6 konstruieren wir nun den Steuerungsgraphen G_m^C des A5/1 Schlüsselstromgenerators. Die zwischengespeicherten Ausgabebits *newW* und

Algorithmus 6 $C(z)$

```
// compute the output bitstream  $y$  from the given internal bitstream  $z$  of length  $m$ 
//  $z$  is interpreted as  $z = z_0^0, \dots, z_0^5, \dots, z_s^0, \dots, z_s^5, \dots, z_{m-1}^{m-1 \bmod 6}$ 
//  $z_{m-1}^{m-1 \bmod 6}$ 
output([0, 0, 0], NIL, NIL, NIL)
```

```
function output( $i, u, v, w$ )
//  $i$ =current read position
//  $u$ =unchanged index  $\in \{0, 1, 2, NIL\}$ ,  $v$ =unchanged control value  $\in \{0, 1, NIL\}$ 
//  $w$ =unchanged output value  $\in \{0, 1, NIL\}$ 
if  $\exists r \in \{0, 1, 2\} \setminus \{u\} : 6 \cdot i[r] + r \geq m$  then
  stop
Read  $\leftarrow \{0, 1, 2\} \setminus \{u\}$ 
Let  $r_0, \dots, r_{|Read|-1}$  the elements of Read in ascending order, i.e.  $r_m < r_n$  for  $m < n$ 
newOut[ $r_0$ ]  $\leftarrow z_{i[r_0]}^{r_0}$ 
newOut[ $r_1$ ]  $\leftarrow z_{i[r_1]}^{r_1}$ 
if  $u \neq NIL$  then //  $\exists$  an unchanged index
  newOut[ $u$ ]  $\leftarrow w$  // copy the unchanged output value
else // all read positions incremented
  newOut[ $r_2$ ]  $\leftarrow z_{i[r_2]}^{r_2}$  // read the third output value
keybit  $\leftarrow$  newOut[0]  $\oplus$  newOut[1]  $\oplus$  newOut[2]
write keybit
control( $i, u, v, newOut$ )
```

```
function control( $i, u, v, out$ )
// out[0..2]=current output values of the LFSR  $L^0, L^1, L^2$ ,  $out[r] \in \{0, 1\}$ 
if  $\exists r \in \{0, 1, 2\} \setminus \{u\} : 6 \cdot i[r] + 3 + r \geq m$  then
  stop
Read  $\leftarrow \{0, 1, 2\} \setminus \{u\}$ 
Let  $r_0, \dots, r_{|Read|-1}$  the elements of Read in ascending order, i.e.  $r_m < r_n$  for  $m < n$ 
 $c[r_0]$   $\leftarrow z_{i[r_0]}^{3+r_0}$ 
 $c[r_1]$   $\leftarrow z_{i[r_1]}^{3+r_1}$ 
if  $u \neq NIL$  then //  $\exists$  an unchanged index
   $c[u]$   $\leftarrow v$  // copy the unchanged control value
else // all read positions incremented
   $c[r_2]$   $\leftarrow z_{i[r_2]}^{3+r_2}$  // read the third control value
controlbit  $\leftarrow$  maj3( $c[0], c[1], c[2]$ )
if  $\exists r \in \{0, 1, 2\} : c[r] \neq controlbit$  then
  // By definition of maj3,  $\exists$  at most one such  $r$ .
  newU  $\leftarrow r$  // set unchanged index
  newV  $\leftarrow controlValue \oplus 1$  // set unchanged control value
  newW  $\leftarrow out[r]$  // set unchanged output value
else // all read positions incremented
  newU  $\leftarrow NIL$ 
  newV  $\leftarrow NIL$ 
  newW  $\leftarrow NIL$ 
for  $l = 0, 1, 2$  do
   $\Delta i[l] \leftarrow \begin{cases} 0 & \text{for } l = newU \\ 1 & \text{for } l \neq newU \end{cases}$ 
output( $i + \Delta i, newU, newV, newW$ )
```

$out[0..2]$ können wir hierbei vernachlässigen, denn die Reihenfolge, in der die Bits des internen Bitstroms eingelesen werden, ist von diesen Werten unabhängig, wie man anhand der Funktionen $output$ und $control$ leicht nachprüft.

Wir betrachten zunächst die Funktion $output$. Falls alle Lesepositionen inkrementiert werden, werden drei neue Bits des internen Bitstroms eingelesen, und wir erhalten den $output - BDD(i, u, v)$ der Form F_o (vgl. Abbildung 7.1). Andernfalls, d.h. falls nur zwei neue Bits eingelesen werden, ergibt sich der $output - BDD(i, u, v)$ der Form F'_o (vgl. Abbildung 7.2). Der Knoten p wird jeweils definiert durch

$$p := control - BDD(i, u, v)$$

Falls ein $r \in \{0, 1, 2\} \setminus \{u\}$ existiert, so dass $6 \cdot i[r] + r \geq m$, ist das Ende von z erreicht und der Algorithmus 6 bricht ab. Da der Steuerungsgraph G_m^C per definitionem auf jedem Pfad von der Wurzel bis zur *-Senke alle m Variablen des internen Bitstroms einlesen muss, besteht der $output - BDD$ in diesem Fall aus einer Liste aller auf dem Pfad von der Wurzel von G_m^C zum aktuellen Blatt noch nicht eingelesenen Variablen des Bitstroms z in der durch z vorgegebenen Reihenfolge mit der *-Senke als letztem Listenelement.

Die Funktion $control$ kann in folgender Weise in einen BDD $control - BDD(i, u, v)$ transformiert werden:

Für $u = NIL$, d.h. falls alle Lesepositionen inkrementiert wurden, werden drei neue Variablen des internen Bitstroms eingelesen, für die sich insgesamt 8 verschiedene mögliche Belegungen ergeben. Der $control - BDD$ hat für $u = NIL$ also die Form F_c (vgl. Abbildung 7.3), wobei die Blattknoten q_0, \dots, q_7 wie folgt bestimmt werden:

Es seien $Read$ und $r_0, \dots, r_{|Read|-1}$ definiert wie in der Funktion $control$. Weiterhin seien $(a_0^j, a_1^j, a_2^j) \in \{0, 1\}^3$ die Belegungen der Variablen $z_{i[r_0]}^{3+r_0}$, $z_{i[r_1]}^{3+r_1}$ und $z_{i[r_2]}^{3+r_2}$ auf dem Pfad von der Wurzel in F_c zu q_j , $j = 0, \dots, 7$. Falls ein $r \in \{0, 1, 2\}$ existiert, so dass $a_r^j \neq maj_3(a_0^j, a_1^j, a_2^j)$, wähle $u(q_j) := r$ und $v(q_j) := maj_3(a_0^j, a_1^j, a_2^j) \oplus 1$, ansonsten setze $u(q_j) := v(q_j) := NIL$. Dann sind die Blattknoten definiert durch

$$q_j := output - BDD(i + \Delta i(u(q_j)), u(q_j), v(q_j))$$

Hierbei definieren wir analog zur Funktion $control$

$$\Delta i[l](b) := \begin{cases} 1 & \text{falls } l = b \\ 0 & \text{falls } l \neq b \end{cases} \quad \text{für } l \in \{0, 1, 2\} \text{ und } b \in \{0, 1\}$$

Im Fall $u \neq NIL$, d.h. falls eine der Lesepositionen nicht inkrementiert wird, hat der $control - BDD(i, u, v)$ die Form F'_c (vgl. Abbildung 7.4). Die Blattknoten $q_{2j, 2j+1}$ von F'_c ergeben sich für $j = 0, \dots, 3$ basierend auf dem bereits bekannten Wert v als

$$q_{2j, 2j+1} := \left\{ \begin{array}{ll} q_{2j} & \text{falls } v = 0 \\ q_{2j+1} & \text{falls } v = 1 \end{array} \right\} = q_{2j+v}$$

Insgesamt kann damit der Steuerungsgraph G_m^C dargestellt werden als

$$output - BDD([0, 0, 0], NIL, NIL)$$

Jeder rekursiv erzeugte Teil-BDD wird nur einmal im Gesamt-BDD repräsentiert und besitzt ohne die rekursiven Fortsetzungen an den Blättern eine konstante Größe. Da die Parameter i, u, v insgesamt $O(m^3)$ verschiedene mögliche Belegungen besitzen, existieren höchstens $O(m^3)$ verschiedene rekursiv erzeugte Teil-BDDs, die ohne die rekursiven Fortsetzungen an den Blättern maximal 7 Knoten enthalten. Wir erhalten also:

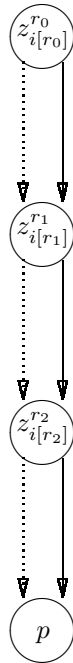


Abbildung 7.1: BDD F_o



Abbildung 7.2: BDD F'_o

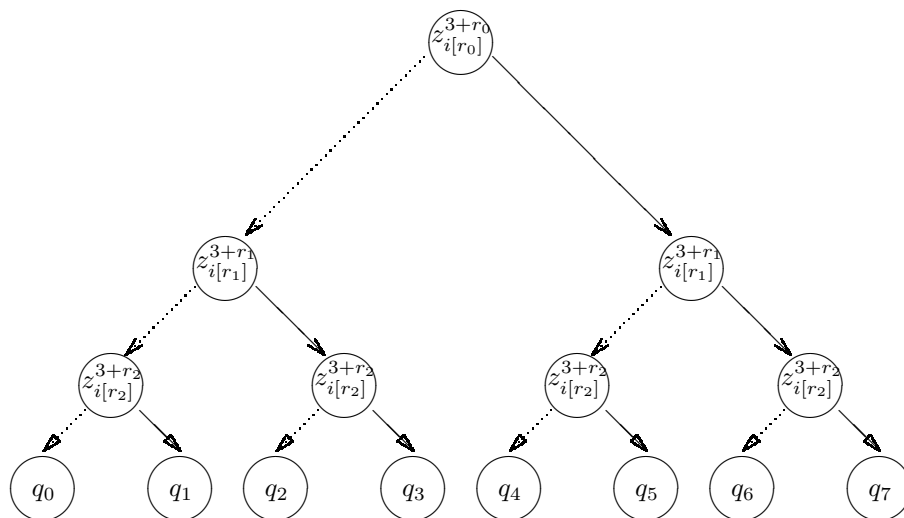
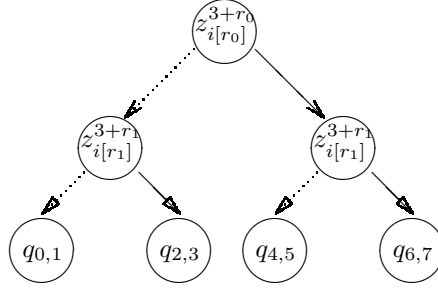


Abbildung 7.3: BDD F_c


 Abbildung 7.4: BDD F'_c

Lemma 53. *Der Steuerungsgraph G_m^C des A5/1 Schlüsselstromgenerators, der für $z \in \{0,1\}^m$ angibt, in welcher Reihenfolge die Bits von z durch die Kompressionsfunktion C eingelesen werden, besitzt eine Größe von $|G_m^C| \in O(m^3)$. \square*

7.3 Konstruktion des G_m^C -FBDDs Q_m

Neben dem Steuerungsgraphen G_m^C benötigen wir als nächstes den G_m^C -FBDD Q_m , der für einen internen Bitstrom z mit $|z| = m$ entscheidet, ob $C(z)$ Präfix von y ist. Ein Algorithmus für dieses Entscheidungsproblem erhalten wir unmittelbar aus Algorithmus 6, indem wir anstatt die berechneten Schlüsselbits in der Funktion *output* auszugeben, überprüfen ob die Schlüsselbits mit den Bits des beobachteten Schlüsselstroms y übereinstimmen (vgl. Algorithmus 7).

Der zugehörige G_m^C -FBDD *test – control – FBDD* kann unmittelbar aus dem BDD *control – BDD* abgeleitet werden. Der *test – output – FBDD* besteht im Unterschied zum *output – BDD* nun analog zum *control – BDD* ebenfalls aus einem Binärbaum mit 8 Blättern für $u = NIL$ bzw. 4 Blättern für $u \neq NIL$. Die Blattknoten p_0, \dots, p_7 ergeben sich als

$$p_n := \begin{cases} 0\text{-Senke} & \text{für } b_0^j \oplus b_1^j \oplus b_2^j \neq y_j \\ \text{control – FBDD}(i, u, v, \text{newOut}, j) & \text{für } b_0^j \oplus b_1^j \oplus b_2^j = y_j \end{cases}$$

d.h. wenn das aus b_0^j, b_1^j und b_2^j berechnete Ausgabebit nicht dem beobachteten Ausgabebit y_{pos} entspricht, wird direkt zur 0-Senke verzweigt, da der aus dem Bitstrom z erzeugte Schlüsselstrom $C(z)$ in diesem Fall nicht Präfix von y sein kann. Zusätzlich besteht der *test – output – FBDD* im Fall $\exists r \in \{0, 1, 2\} \setminus \{u\} : 6 \cdot i[r] + r \geq m$ lediglich aus der 1-Senke.

Insgesamt kann der G_m^C -FBDD Q_m somit dargestellt werden als

$$\text{test – output – FBDD}([0, 0, 0], NIL, NIL, NIL, 0)$$

Aus m internen Bits werden höchstens $\gamma m \in O(m)$ Schlüsselbits erzeugt. Damit existieren höchstens $O(m^4)$ verschiedene Belegungen für die Variablen $i, u, v, w, \text{out}, j$ und damit höchstens $O(m^4)$ verschiedene *test – output – FBDDs* bzw. *test – control – FBDDs*, die jeweils nur einmal im Gesamt-FBDD repräsentiert werden und höchstens 7 Knoten enthalten. Wir erhalten somit:

Lemma 54. *Der G_m^C -FBDD Q_m des A5/1 Schlüsselstromgenerators, der für einen internen Bitstrom $z \in \{0,1\}^m$ entscheidet, ob $C(z)$ Präfix eines Schlüsselstroms $y \in \{0,1\}^*$ ist, besitzt eine Größe von $|Q_m| \in O(m^4)$. \square*

Algorithmus 7 Test- $C(z, y)$

```

// test whether the keystream  $C(z)$  that is computed
// from the given internal bitstream  $z$  is prefix of a given keystream  $y$ 
//  $z$  is interpreted as  $z = z_0^0, \dots, z_0^5, \dots, z_s^0, \dots, z_s^5, \dots, z_{m-1 \text{ div } 6}^{m-1 \text{ mod } 6}$ 
test - output([0, 0, 0], NIL, NIL, NIL, 0)

```

```

function output( $i, u, v, w, pos$ )
//  $i$ =current read position
//  $u$ =unchanged index  $\in \{0, 1, 2, NIL\}$ ,  $v$ =unchanged control value  $\in \{0, 1, NIL\}$ 
//  $w$ =unchanged output value  $\in \{0, 1, NIL\}$ 
//  $pos$ =comparison position  $\in \{0, \dots, \gamma m - 1\}$ 
if  $\exists r \in \{0, 1, 2\} \setminus \{u\} : 6 \cdot i[r] + r \geq m$  then
  stop
  Read  $\leftarrow \{0, 1, 2\} \setminus \{u\}$ 
  Let  $r_0, \dots, r_{|Read|-1}$  the elements of Read in ascending order, i.e.  $r_m < r_n$  for  $m < n$ 
  newOut[ $r_0$ ]  $\leftarrow z_{i[r_0]}^{r_0}$ 
  newOut[ $r_1$ ]  $\leftarrow z_{i[r_1]}^{r_1}$ 
if  $u \neq NIL$  then //  $\exists$  an unchanged index
  newOut[ $u$ ]  $\leftarrow w$  // copy the unchanged output value
else // all read positions incremented
  newOut[ $r_2$ ]  $\leftarrow z_{i[r_2]}^{r_2}$  // read the third output value
  keybit  $\leftarrow$  newOut[0]  $\oplus$  newOut[1]  $\oplus$  newOut[2]
if  $y_{pos} \neq keybit$  then
  stop(false)
else
  control( $i, u, v, newOut, pos$ )

```

```

function control(i,u,v,out,pos)
// out[0..2]=current output values of the LFSR  $L^0, L^1, L^2$ ,  $out[r] \in \{0,1\}$ 
if  $\exists r \in \{0,1,2\} \setminus \{u\} : 6 \cdot i[r] + 3 + r \geq m$  then
    stop
    Read  $\leftarrow \{0,1,2\} \setminus \{u\}$ 
    Let  $r_0, \dots, r_{|Read|-1}$  the elements of Read in ascending order, i.e.  $r_m < r_n$  for  $m < n$ 
     $c[r_0] \leftarrow z_{i[r_0]}^{3+r_0}$ 
     $c[r_1] \leftarrow z_{i[r_1]}^{3+r_1}$ 
if  $u \neq NIL$  then //  $\exists$  an unchanged index
     $c[u] \leftarrow v$  // copy the unchanged control value
else // all read positions incremented
     $c[r_2] \leftarrow z_{i[r_2]}^{3+r_2}$  // read the third control value
    controlbit  $\leftarrow maj_3(c[0], c[1], c[2])$ 
if  $\exists r \in \{0,1,2\} : c[r] \neq controlbit$  then
    // By definition of  $maj_3$ ,  $\exists$  at most one such  $r$ .
    newU  $\leftarrow r$  // set unchanged index
    newV  $\leftarrow controlValue \oplus 1$  // set unchanged control value
    newW  $\leftarrow out[r]$  // set unchanged output value
else // all read positions incremented
    newU  $\leftarrow NIL$ 
    newV  $\leftarrow NIL$ 
    newW  $\leftarrow NIL$ 
for  $l = 0, 1, 2$  do
     $\Delta i[l] \leftarrow \begin{cases} 0 & \text{for } l = newU \\ 1 & \text{for } l \neq newU \end{cases}$ 
    output( $i + \Delta i, newU, newV, newW, pos + 1$ )

```

7.4 Konstruktion der G_m^C -FBDDs R_m und S_m

Der G_m^C -FBDD R_m entscheidet für einen internen Bitstrom $z \in \{0, 1\}^m$, ob $z = L_{\leq m}(i_m(L, z))$, d.h. ob z durch den linearen Bitstromgenerator L erzeugt werden kann (vgl. Definition 42).

Wie im Abschnitt 7.1.2 dargestellt, besteht der lineare Bitstromgenerator L des A5/1-Schlüsselstromgenerators aus den sechs LFSR L^0, \dots, L^5 und erzeugt für $m \geq 1$ den internen Bitstrom

$$z = z_0^0, \dots, z_0^5, \dots, z_s^0, \dots, z_s^5, \dots, z_{m-1}^{m-1 \bmod k}, \dots, z_{m-1}^{m-1 \operatorname{div} k} \in \{0, 1\}^m$$

Es gilt also

$$z_s^r = \bigoplus_{j=0}^{n_r-1} L_{j,s}^r \cdot z_j^r \text{ für } r \in \{0, \dots, 5\} \text{ und } s \geq n_r \quad (7.1)$$

Zusätzlich entspricht für $r \in \{0, 1, 2\}$ der Ausgabebitstrom des LFSR L^{3+r} dem um N^r verschobenen Ausgabebitstrom des LFSR L^r , d.h.

$$z_s^r = z_{s-N^r}^{3+r} \text{ für } r \in \{0, 1, 2\} \text{ und } s \geq N^r \quad (7.2)$$

Der G_m^C -FBDD R_m muss also für einen gegebenen internen Bitstrom $z \in \{0, 1\}^m$ überprüfen, ob z die Linearitätsbedingung (7.1) und die *Shift*-Bedingung (7.2) erfüllt.

Wir wollen zunächst die folgende, für die Konstruktion von R_m grundlegende Beobachtung anstellen:

Beobachtung 55. Für eine beliebige, durch den Steuerungsgraphen G_m^C des A5/1 Schlüsselstromgenerators definierte Einleseordnung π der Variablen z_0, \dots, z_{m-1} mit $z_s^r = z_{6 \cdot s + r}$ gilt

$$\pi(z_{s-N^r}^{3+r}) < \pi(z_s^r) \text{ für } r \in \{0, 1, 2\} \text{ und } s \geq N^r$$

Beweis. Die Konstruktion des A5/1-Schlüsselstromgenerators und des Steuerungsgraphen G_m^C (vgl. Abschnitt 7.2) impliziert für $r \in \{0, 1, 2\}$ und $s \geq 0$

$$\pi(z_s^r) < \pi(z_s^{3+r}) < \pi(z_{s+1}^r)$$

Für $N^r \geq 1$ folgt hieraus unmittelbar

$$\pi(z_{s-N^r}^{3+r}) < \pi(z_{s-N^r+1}^r) \leq \pi(z_s^r) \quad \square$$

Aus jedem Pfad in G_m^C wird also das Kontrollbit $z_{s-N^r}^{3+r}$ vor dem korrespondierenden Ausgabebit z_s^r eingelesen. Auf diese Weise wird, falls alle z_s^{3+r} mit $r \in \{0, 1, 2\}$ und $s \geq n_r$ die Linearitätsbedingung erfüllen, durch die *Shift*-Bedingung für z_s^r , $r \in \{0, 1, 2\}$ und $s \geq N^r$ gleichzeitig sichergestellt, dass die z_s^r mit $r \in \{0, 1, 2\}$ und $s \geq N^r$ die Linearitätsbedingung erfüllen. Lediglich für die ersten N^r linear kombinierten Bits der LFSR L^0 , L^1 und L^2 muss die Linearitätsbedingung zusätzlich geprüft werden, da die ersten N^r Bits dieser LFSR durch die *Shift*-Bedingung nicht erfasst werden.

Um zu entscheiden, ob ein gegebener interner Bitstrom $z \in \{0, 1\}^m$ durch L erzeugt werden kann, genügt es also, folgende Bedingungen zu überprüfen:

$$z_s^r = \bigoplus_{j=0}^{n_r-3-1} L_{j,s}^r \cdot z_j^r \text{ für } r \in \{3, 4, 5\} \text{ und } s \geq n_{r-3} \quad (7.3)$$

$$z_s^r = \bigoplus_{j=0}^{n_r-1} L_{j,s}^r \cdot z_j^r \text{ für } r \in \{0, 1, 2\} \text{ und } n_r \leq s < n_r + N^r \quad (7.4)$$

$$z_s^r = z_{s-N^r}^{3+r} \text{ für } r \in \{0, 1, 2\} \text{ und } s \geq N^r \quad (7.5)$$

Wir erhalten nun basierend auf Algorithmus 5, den wir im Beweis von Lemma 47 entworfen haben, den *read-once* Algorithmus 8 für das Entscheidungsproblem, ob ein gegebener interner Bitstrom $z \in \{0, 1\}^m$ durch den linearen Bitstromgenerator des A5/1 Schlüsselstromgenerators erzeugt worden ist.

Algorithmus 8 *test – linearity – A5(z, G_m^C)*

$b^c = (b_{j_1}^c, \dots, b_{j_{|b^c|}}^c) \leftarrow \vec{0}$ where $j_l \in I^c := \{6s + r \mid r \in \{3, 4, 5\}, s \geq n_{r-3}\}$
 $b^o = (b_{j_1}^o, \dots, b_{j_{|b^o|}}^o) \leftarrow \vec{0}$ where $j_l \in I^o := \{6s + r \mid r \in \{0, 1, 2\}, n_r \leq s < n_r + N^r\}$
 $b^{shift} = (b_{j_1}^{shift}, \dots, b_{j_{|b^{shift}|}}^{shift}) \leftarrow \vec{0}$ where $j_l \in I^{shift} := \{6s + r \mid r \in \{0, 1, 2\}, s \geq N^r\}$
 Let $\pi_{G_m^C}(z)$ be the reading-order of the z_i , $i \in \{0, \dots, m-1\}$, as defined by G_m^C
for $l \leftarrow 0$ to $m-1$ **do**
 $i \leftarrow (\pi_{G_m^C}(z))(l)$
 $r \leftarrow i \bmod 6$
 $s \leftarrow i \operatorname{div} 6$
 if $r \in \{3, 4, 5\}$ AND $s < n_{r-3}$ **then** // update b^c
 for all $j \in I^c$ **do**
 $b_j^c \leftarrow b_j^c \oplus L_{s,s(j)}^{r-3} \cdot z_i$
 if $r \in \{0, 1, 2\}$ AND $s < n_r$ **then** // update b^o
 for all $j \in I^o$ **do**
 $b_j^o \leftarrow b_j^o \oplus L_{s,s(j)}^r \cdot z_i$
 if $r \in \{3, 4, 5\}$ AND $6 \cdot (s + N^r) + (r - 3) \in I^{shift}$ **then** // update b^{shift}
 $b_{6 \cdot (s + N^r) + (r - 3)}^{shift} \leftarrow z_i$
 if $r \in \{3, 4, 5\}$ AND $s \geq n_{r-3}$ **then** // check consistency with b_i^c
 if $b_i^c \neq z_i$ **then**
 stop(0)
 if $r \in \{0, 1, 2\}$ AND $n_r \leq s < n_r + N^r$ **then** // check consistency with b_i^o
 if $b_i^o \neq z_i$ **then**
 stop(0)
 if $r \in \{0, 1, 2\}$ AND $s \geq N^r$ **then** // check consistency with b_i^{shift}
 if $b_i^{shift} \neq z_i$ **then**
 stop(0)
 stop(1)

Die Umwandlung von Algorithmus 8 in den G_m^C FBDD R_m erfolgt analog zur Konstruktion im Beweis von Lemma 47 und soll daher an dieser Stelle nicht im Detail ausgeführt werden.

Die Größe von R_m ist genau wie im allgemeinen Fall abhängig vom zusätzlich benötigten Speicherplatz, d.h. von der Gesamtlänge $|I^c| + |I^o| + |I^{shift}|$ der Vektoren b^c , b^o und b^{shift} . Unter der Annahme, dass m durch 6 teilbar ist, werden genau $\frac{m}{2}$ Bits von z durch die LFSR L^0 , L^1 und L^2 und die andere Hälfte durch L^3 , L^4 und L^5 erzeugt. Damit ist $|I^c| = \frac{m}{2} - |I_m(L)|$, wobei $|I_m(L)| \leq n$, da die Initialzustandsbits der LFSR L^0 , L^1 und L^2 diejenigen der LFSR L^3 , L^4 und L^5 sich gegenseitig determinieren. I^o und I^{shift} enthalten zusammen genau $\frac{m}{2}$ Elemente, so dass wir insgesamt erhalten:

$$|I^c| + |I^o| + |I^{shift}| = \frac{m}{2} - I_m(L) + \frac{m}{2} = m - I_m(L)$$

und damit gilt genau wie im allgemeinen Fall $|R_m| \leq |G_m^C| 2^{m - |I_m(L)|}$. Falls m nicht durch 6 teilbar ist, unterscheidet sich $|R_m|$ von diesem Wert um einen konstanten Faktor. Wir haben damit gezeigt:

Lemma 56. *Der G_m^C -FBDD R_m des A5/1 Schlüsselstromgenerators, der für einen gegebenen internen Bitstrom $z \in \{0, 1\}^m$ entscheidet, ob $z = L_{\leq m}(i_m(L, z))$ gilt, besitzt eine Größe von $|R_m| \leq$*

$$|G_m^C|^{2^m - |I_m(L)|}. \quad \square$$

Der G_m^C -FBDD S_m , der für einen internen Bitstrom $z \in \{0, 1\}^m$ entscheidet, ob $z_{m-1} = L_{m-1}(i_m(L, z))$ gilt, muss für $m-1 \bmod 6 \in \{3, 4, 5\}$ nur einen Vergleichswert b_1^c speichern und für $m-1 \bmod 6 \in \{0, 1, 2\}$ höchstens zwei Vergleichswerte b_1^o und b_1^{shift} verwalten. Es folgt somit unmittelbar:

Korollar 57. *Der G_m^C -FBDD S_m , der für einen internen Bitstrom $z \in \{0, 1\}^m$ entscheidet, ob $z_{m-1} = L_{m-1}(i_m(L, z))$ erfüllt ist, besitzt eine Größe von $|S_m| \leq 2^2 |G_m^C| = 4|G_m^C|$.* \square

7.5 Statistische Eigenschaften des modifizierten A5/1 Schlüsselstromgenerators

Gemäß Definition 8 und Lemma 7 kann der interne Bitstrom eines LFSR-basierten Schlüsselstromgenerators als pseudozufällig vorausgesetzt werden. Wir treffen daher insbesondere für den A5/1 Generator für die weitere Analyse die folgende grundlegende Annahme:

Annahme 5. *Der einzelnen Bits des inneren sind unabhängige Zufallsvariablen, deren Werte unter Gleichverteilung aus $\{0, 1\}$ gezogen werden.*

Aus der Definition von maj_3 lässt sich unmittelbar ableiten:

Beobachtung 58. *Bei der Berechnung eines Schlüsselbits können zwei Fälle auftreten:*

- (i) *Alle drei LFSR sind getaktet worden, d.h. es werden drei Bits des internen Bitstroms zur Berechnung des nächsten Schlüsselbits und drei Bits zur Ermittlung des nächsten Kontrollbits eingelesen.*
- (ii) *Es sind nur zwei der drei LFSR getaktet worden, d.h. es werden jeweils zwei Bits des internen Bitstroms zur Berechnung des nächsten Schlüsselbits bzw. des Kontrollbits eingelesen, und die Werte des nicht getakteten LFSRs werden nochmals verwendet.*

Wir benötigen für die weitere Analyse die folgende Aussage:

Lemma 59. *Es sei $Y'_i \in \{0, 1, -\}$, $i \geq 1$, der Wert der zur Berechnung des i -ten Kontrollbits wiederverwendeten Variablen. Dann gilt $\text{Prob}[Y'_i = 0] = \text{Prob}[Y'_i = 1] = \frac{3}{8}$ und $\text{Prob}[Y'_i = -] = \frac{2}{8}$.*

Beweis. Wir beweisen die behauptete Aussage mittels vollständiger Induktion über i .

Es gilt

$$\begin{aligned} \text{Prob}[Y'_1 = 0] &= \text{Prob}[(z_3, z_4, z_5) \in \{(0, 1, 1), (1, 0, 1), (1, 1, 0)\}] = \frac{3}{8} \\ \text{Prob}[Y'_1 = 1] &= \text{Prob}[(z_3, z_4, z_5) \in \{(1, 0, 0), (0, 1, 0), (0, 0, 1)\}] = \frac{3}{8} \\ \text{Prob}[Y'_1 = -] &= \text{Prob}[(z_3, z_4, z_5) \in \{(0, 0, 0), (1, 1, 1)\}] = \frac{2}{8} \end{aligned}$$

Damit ist die Behauptung wahr für $i = 1$. Wir nehmen nun an, die Behauptung gelte für $i \geq 1$.

Y'_{i+1} ergibt sich aus Y'_i und den neu eingelesenen Kontrollvariablen z_{i_1}, z_{i_2} für $Y'_i \in \{0, 1\}$ bzw. $z_{i_1}, z_{i_2}, z_{i_3}$ für $Y'_i = -$ in folgender Weise:

Y'_i	z_{i_1}	z_{i_2}	z_{i_3}	Y'_{i+1}
0	0	0		-
0	0	1		1
0	1	0		1
0	1	1		0
1	0	0		1
1	0	1		0
1	1	0		0
1	1	1		-
-	0	0	0	-
-	0	0	1	1
-	0	1	0	1
-	0	1	1	0
-	1	0	0	1
-	1	0	1	0
-	1	1	0	0
-	1	1	1	-

 Tabelle 7.1: Berechnung von Y'_{i+1}

Aus der Definition der Taktkontrollfunktion ergibt sich, dass Y'_i dem Wert eines zuvor eingelesenen internen Bits entspricht. Da die Bitwerte des internen Bitstroms als unabhängig vorausgesetzt werden (vgl. Annahme 5), sind Y'_i und die neu eingelesenen Variablen $z_{i_1}, z_{i_2}, z_{i_3}$ unabhängige Zufallsvariablen. Es gilt daher für $c \in \{0, 1\}$ und $c_1, c_2 \in \{0, 1\}$ bzw. für $c = -$ und $c_1, c_2, c_3 \in \{0, 1\}$:

$$\text{Prob}[Y'_i = c, z_{i_1} = c_1, z_{i_2} = c_2] = \text{Prob}[Y'_i = c] \cdot \text{Prob}[z_{i_1} = c_1, z_{i_2} = c_2] = \text{Prob}[Y'_i = c] \cdot \frac{1}{4}$$

bzw. $\text{Prob}[Y'_i = -, z_{i_1} = c_1, z_{i_2} = c_2, z_{i_3} = c_3] = \text{Prob}[Y'_i = -] \cdot \frac{1}{8}$.

Aufgrund der Induktionsannahme gilt $\text{Prob}[Y'_i = 0] = \text{Prob}[Y'_i = 1] = \frac{3}{8}$ und $\text{Prob}[Y'_i = -] = \frac{2}{8}$. Durch Aufsummieren der jeweiligen Zeilenwahrscheinlichkeiten erhält man aus Tabelle 7.1

$$\text{Prob}[Y'_{i+1} = 0] = 3 \cdot \frac{3}{8} \cdot \frac{1}{4} + 3 \cdot \frac{2}{8} \cdot \frac{1}{8} = \frac{3}{8}$$

$$\text{Prob}[Y'_{i+1} = 1] = 3 \cdot \frac{3}{8} \cdot \frac{1}{4} + 3 \cdot \frac{2}{8} \cdot \frac{1}{8} = \frac{3}{8}$$

$$\text{Prob}[Y'_{i+1} = -] = 2 \cdot \frac{3}{8} \cdot \frac{1}{4} + 2 \cdot \frac{2}{8} \cdot \frac{1}{8} = \frac{2}{8}$$

Damit gilt die Behauptung für $i + 1$. □

Definition 60. Für $i \geq 1$ bezeichne die Zufallsvariable Y_i die Anzahl der Bits des internen Bitstroms, die der A5/1 Generator zur Berechnung des Schlüsselbits y_i einliest.

Gemäß Beobachtung 58 gilt $Y_i \in \{4, 6\}$ für alle $i \geq 1$.

Lemma 61. Für $i \geq 1$ ist $\text{Prob}[Y_i = 6] = \frac{1}{4}$ und $\text{Prob}[Y_i = 4] = \frac{3}{4}$.

Beweis. Für alle $i \geq 1$ ist $Y_i = 4$ genau dann, wenn $Y'_i \in \{0, 1\}$ und $Y_i = 6$ genau dann, wenn $Y'_i = -$. Aus Lemma 59 folgt

$$\text{Prob}[Y_1 = 6] = \text{Prob}[Y'_1 = -] = \frac{2}{8} = \frac{1}{4}$$

$$\text{Prob}[Y_1 = 4] = \text{Prob}[Y'_1 \in \{0, 1\}] = \frac{6}{8} = \frac{3}{4} \quad \square$$

Lemma 62. $Y_1, Y_2, \dots, Y_l, l \geq 1$, sind unabhängige Zufallsvariablen.

Beweis. Wir zeigen mittels vollständiger Induktion über l , dass

$$\forall J = \{j_1, \dots, j_{|J|}\} \in \mathcal{P}(\{1, \dots, l\}), \forall c = (c_1, \dots, c_{|J|}) \in \{4, 6\}^{|J|} \text{ gilt}$$

$$Prob[Y_j = c_j, j \in J] = \prod_{k=1}^{|J|} Prob[Y_{j_k} = c_k] \quad (7.6)$$

wobei $\mathcal{P}(\{1, \dots, l\})$ die Potenzmenge, d.h. alle möglichen Teilmengen von $\{1, \dots, l\}$ bezeichnet.

Der Fall $l = 1$ ist trivial.

Wir nehmen nun an, (7.6) sei erfüllt für $l \geq 1$. Es ist

$$\mathcal{P}(\{1, \dots, l, l+1\}) = \mathcal{P}(\{1, \dots, l\}) \cup (\mathcal{P}(\{1, \dots, l\}) \times \{l+1\})$$

Für den Nachweis, dass (7.6) für $l+1$ erfüllt ist, genügt es also zu zeigen, dass für alle $J = \{j_1, \dots, j_{|J|}\} \in \mathcal{P}(\{1, \dots, l\})$ und für alle $c = (c_1, \dots, c_{|J|}, c_{l+1}) \in \{4, 6\}^{|J|+1}$ gilt

$$Prob[Y_{j_1} = c_1, Y_{j_2} = c_2, \dots, Y_{j_{|J|}} = c_{|J|}, Y_{l+1} = c_{|J|+1}] = \prod_{k=1}^{|J|+1} Prob[Y_{j_k} = c_k]$$

Es gilt für alle $J \in \mathcal{P}(\{1, \dots, l\})$ und $c = (c_1, \dots, c_{|J|}, c_{l+1}) \in \{4, 6\}^{|J|+1}$

$$\begin{aligned} & Prob[Y_{j_1} = c_1, \dots, Y_{j_{|J|}} = c_{|J|}, Y_{l+1} = c_{l+1}] \\ &= Prob[Y_{l+1} = c_{l+1} | Y_{j_1} = c_1, \dots, Y_{j_{|J|}} = c_{|J|}] \cdot Prob[Y_{j_1} = c_1, \dots, Y_{j_{|J|}} = c_{|J|}] \\ &= Prob[Y_{l+1} = c_{l+1} | Y_{j_1} = c_1, \dots, Y_{j_{|J|}} = c_{|J|}] \cdot \prod_{k=1}^{|J|} Prob[Y_{j_k} = c_k] \end{aligned}$$

gemäß der Induktionsvoraussetzung. Weiterhin ist

$$Prob[Y_{l+1} = c_{l+1} | Y_{j_1} = c_1, \dots, Y_{j_{|J|}} = c_{|J|}] = Prob[Y_{l+1} = c_{l+1} | Y_l = c_l] = Prob[A_{l+1} | A_l]$$

$$\text{mit } A_k := \begin{cases} (Y'_k = 0 \vee Y'_k = 1) & \text{für } c_k = 4 \\ (Y'_k = -) & \text{für } c_k = 6 \end{cases}$$

Man prüft mit Hilfe von Tabelle 7.1 leicht nach, dass $Prob[A_{l+1} | A_l] = Prob[Y_{l+1} = c_{l+1}]$. Es gilt also

$$Prob[Y_{j_1} = c_1, \dots, Y_{j_{|J|}} = c_{|J|}, Y_{l+1} = c_{l+1}] = \prod_{k=1}^{|J|+1} Prob[Y_{j_k} = c_k]$$

d.h. (7.6) ist erfüllt für $l+1$. □

Lemma 63. Der A5/1 Generator erfüllt die Unabhängigkeitsannahme.

Beweis. Es ist zu zeigen, dass für $m \geq 1$ und einen zufällig gemäß Gleichverteilung gewählten internen Bitstrom z die Wahrscheinlichkeit, dass $C(z)$ Präfix eines Schlüsselstroms y ist, für alle Schlüsselströme y denselben Wert besitzt.

Wir fixieren ein $i \in N$ und einen zufällig gemäß Gleichverteilung gewählten internen Bitstrom $z \in \{0, 1\}^m$ mit $|C(z)| = i$, d.h. die Kompressionsfunktion generiert aus z die Schlüsselbits

$$C(z) = \tilde{y} = (\tilde{y}_0, \dots, \tilde{y}_{i-1}) \in \{0, 1\}^i$$

Damit $C(z)$ Präfix eines beliebig gewählten Schlüsselstroms $y = (y_0, \dots, y_{|y|-1}) \in \{0, 1\}^*$ ist, muss gelten

$$\tilde{y}_j = y_j \text{ für alle } j = 0, \dots, i-1$$

Wir werden in Lemma 64 zeigen, dass für alle $1 \leq j \leq i$ gilt $Prob[\tilde{y}_0 = y_0, \dots, \tilde{y}_{j-1} = y_{j-1}] = \left(\frac{1}{2}\right)^j$.
Damit erhalten wir:

$$Prob_{z, |C(z)|=i}[C(z) \text{ ist Präfix von } y] = Prob[\tilde{y}_0 = y_0, \dots, \tilde{y}_{i-1} = y_{i-1}] = \left(\frac{1}{2}\right)^i$$

und somit

$$\begin{aligned} Prob_z[C(z) \text{ ist Präfix von } y] &= \\ \sum_{i=0}^{\lceil \gamma m \rceil} Prob_z[|C(z)| = i] \cdot Prob_{z, |C(z)|=i}[C(z) = (y_0, \dots, y_{i-1})] &= \\ \sum_{i=0}^{\lceil \gamma m \rceil} Prob_z[|C(z)| = i] \cdot 2^{-i} & \end{aligned}$$

d.h. $Prob_z[C(z) \text{ ist Präfix von } y]$ ist unabhängig von y und damit für alle Schlüsselströme gleich. \square

Lemma 64. Für alle $1 \leq j \leq i$ gilt $Prob[\tilde{y}_0 = y_0, \dots, \tilde{y}_{j-1} = y_{j-1}] = \left(\frac{1}{2}\right)^j$.

Beweis. Für $i = 0$ ist $Prob[\tilde{y}_0 = y_0] = \frac{1}{2}$ wegen

$$\begin{aligned} Prob[\tilde{y}_0 = 0] &= Prob[(z_0, z_1, z_2) \in \{(0, 0, 0), (1, 0, 1), (1, 1, 0), (0, 1, 1)\}] = \frac{4}{8} = \frac{1}{2} \\ Prob[\tilde{y}_0 = 1] &= Prob[(z_0, z_1, z_2) \in \{(0, 0, 1), (0, 1, 0), (1, 0, 0), (1, 1, 1)\}] = \frac{4}{8} = \frac{1}{2} \end{aligned}$$

Es gelte nun $Prob[\tilde{y}_0 = y_0, \dots, \tilde{y}_{j-1} = y_{j-1}] = \left(\frac{1}{2}\right)^j$ für $j \geq 1$. Es ist

$$\begin{aligned} & Prob[\tilde{y}_0 = y_0, \dots, \tilde{y}_j = y_j] \\ &= Prob[\tilde{y}_j = y_j | \tilde{y}_0 = y_0, \dots, \tilde{y}_{j-1} = y_{j-1}] \cdot Prob[\tilde{y}_0 = y_0, \dots, \tilde{y}_{j-1} = y_{j-1}] \\ &= Prob[\tilde{y}_j = y_j | \tilde{y}_0 = y_0, \dots, \tilde{y}_{j-1} = y_{j-1}] \cdot \left(\frac{1}{2}\right)^j \text{ gemäß Induktionsvoraussetzung} \end{aligned}$$

Analog zur Berechnung des Kontrollbits existieren zwei Möglichkeiten für die Berechnung von \tilde{y}_j : Falls alle drei LFSR getaktet wurden, werden drei Bits $z_{i_1}, z_{i_2}, z_{i_3}$ des internen Bitstroms eingelesen. Falls nur zwei Bits des internen Bitstroms neu eingelesen werden, ist das dritte Bit z_{i_0} der Wert einer Variablen, die in einer vorherigen Iteration eingelesen wurde. In beiden Fällen ist $Prob[\tilde{y}_j = 0 | \tilde{y}_0 = y_0, \dots, \tilde{y}_{j-1} = y_{j-1}] = Prob[\tilde{y}_j = 1 | \tilde{y}_0 = y_0, \dots, \tilde{y}_{j-1} = y_{j-1}] = \frac{1}{2}$, wie man anhand von Tabelle 7.2 leicht nachprüft.

Damit gilt

$$Prob[\tilde{y}_0 = y_0, \dots, \tilde{y}_j = y_j] = \left(\frac{1}{2}\right)^{j+1}$$

d.h. die Behauptung gilt für $j+1$. \square

z_{i_0}	z_{i_1}	z_{i_2}	z_{i_3}	\tilde{y}_i
0	0	0		0
0	0	1		1
0	1	0		1
0	1	1		0
1	0	0		1
1	0	1		0
1	1	0		0
1	1	1		1
	0	0	0	0
	0	0	1	1
	0	1	0	1
	0	1	1	0
	1	0	0	1
	1	0	1	0
	1	1	0	0
	1	1	1	1

Tabelle 7.2: Berechnung von \tilde{y}_i

7.6 Bestimmung der Informationsrate α des modifizierten A5/1 Schlüsselstromgenerators

Lemma 65. Für die Informationsrate α des A5/1 Pseudozufallsbitgenerators gilt

$$\alpha = \frac{1}{2} \log u_1 \approx 0,2193$$

wobei u_1 die positive reelle Nullstelle des Polynoms $p(u) = u^3 - 3u^2 + 8$ bezeichnet.

Beweis. Für einen beliebigen Schlüsselstrom y und $m \geq 1$ und einen zufälligen gemäß Gleichverteilung gewählten internen Bitstrom z , $|z| = m$, sei $p(m) := \text{Prob}_z[C(z)$ ist Präfix von $y]$. Weiterhin sei für $m \geq 0$ und $k \leq m$

$$p(m, k) := \text{Prob}_{z \in_V \{0,1\}^m}[|C(z)| = k]$$

Weil A5/1 für die Berechnung eines Schlüsselbits entweder 4 oder 6 Bits des internen Bitstroms einliest, ist $\frac{m}{6} \leq |C(z)| \leq \frac{m}{4}$, d.h. aus z können zwischen $\frac{m}{6}$ und $\frac{m}{4}$ Schlüsselbits erzeugt werden. Mit Hilfe von Gleichung (2.1) können wir $p(m)$ somit darstellen als

$$\begin{aligned} p(m) &= \text{Prob}_z[C(z) \text{ ist Präfix von } y] \\ &= \sum_{i=0}^{\lceil \gamma m \rceil} \text{Prob}_z[|C(z)| = i] \cdot \text{Prob}_{C(z)=i}[C(z) = y_0, \dots, y_{i-1}] \\ &= \sum_{k=\frac{m}{6}}^{\frac{m}{4}} p(m, k) \cdot 2^{-k} \end{aligned}$$

Da A5/1 die Gleichverteilungsannahme erfüllt (Lemma 63), gilt andererseits $p(m) = 2^{-\alpha m}$. Wir ermitteln daher zunächst eine geeignete rekursive Darstellung für $p(m)$ und bestimmen hieraus den Wert der Informationsrate α .

Es sei Z die Zufallsvariable, die die Anzahl der Schlüsselbits angibt, die für die Berechnung der ersten $\frac{m}{6}$ Bits des internen Schlüsselstroms verwendet wurden. Weiterhin sei Z' die Zufallsvariable, die die Anzahl

derjenigen Schlüsselbits angibt, für deren Berechnung 6 Bits des internen Bitstroms verwendet wurden. Damit kann Z mit Hilfe von Z' folgendermaßen ausgedrückt werden:

$$Z = 6 \cdot Z' + 4 \cdot \left(\frac{m}{6} - Z'\right)$$

Für die Anzahl der nach der Berechnung der ersten $\frac{m}{6}$ Schlüsselbits noch nicht eingelesenen $m - Z$ Bits des internen Bitstroms gilt

$$m - Z = m - (6 \cdot Z' + 4 \cdot (\frac{m}{6} - Z')) = \frac{1}{3}m - 2Z'$$

Wir erhalten also die folgende Rekurrenz für $p(m)$:

$$\begin{aligned} p(m) &= \sum_{i=0}^{\frac{m}{6}} \left(2^{-\frac{m}{6}} \text{Prob}[Z' = i] \cdot \sum_{j=(\frac{1}{3}m-2i)/6}^{(\frac{1}{3}m-2i)/6} 2^{-j} p\left(\frac{1}{3}m - 2i, j\right) \right) \\ &= \sum_{i=0}^{\frac{m}{6}} 2^{-\frac{m}{6}} \text{Prob}[Z' = i] \cdot p\left(\frac{1}{3}m - 2i\right) \end{aligned}$$

Wir benötigen als nächstes eine Abschätzung für die Wahrscheinlichkeit $\text{Prob}[Z' = i]$. Es gilt:

Hilfsbehauptung 66. Z' ist $(\frac{m}{6}, \frac{1}{4})$ -binomialverteilt.

Beweis. Es sei $Z'_i(Y_i) := \begin{cases} 0 & \text{für } Y_i = 4 \\ 1 & \text{für } Y_i = 6 \end{cases}$ für $i = 0, \dots, \frac{m}{6} - 1$

Es gilt also $Z' = \sum_{i=0}^{\frac{m}{6}-1} Z'_i$. Wegen Lemma 61 ist Z_i Bernoulli-verteilt mit dem Parameter $\theta = \frac{1}{4}$. Da $Y_0, \dots, Y_{\frac{m}{6}-1}$ unabhängig sind (Lemma 62), ist $Z' (\frac{m}{6}, \frac{1}{4})$ -binomialverteilt. \square

Somit kann $\text{Prob}[Z' = i]$ dargestellt werden als

$$\text{Prob}[Z' = i] = \binom{\frac{m}{6}}{i} \left(\frac{1}{4}\right)^i \left(\frac{3}{4}\right)^{\frac{m}{6}-i}$$

Es gilt also für $p(m)$:

$$p(m) = \sum_{i=0}^{\frac{m}{6}} 2^{-\frac{m}{6}} \binom{\frac{m}{6}}{i} \left(\frac{1}{4}\right)^i \left(\frac{3}{4}\right)^{\frac{m}{6}-i} \cdot p\left(\frac{1}{3}m - 2i\right)$$

Mit Hilfe von $p(m) = 2^{-\alpha m}$ erhalten wir hieraus eine Rekurrenz für α :

$$\begin{aligned} 2^{-\alpha m} &= \sum_{i=0}^{\frac{m}{6}} 2^{-\frac{m}{6}} \binom{\frac{m}{6}}{i} \left(\frac{1}{4}\right)^i \left(\frac{3}{4}\right)^{\frac{m}{6}-i} \cdot 2^{-\alpha(\frac{1}{3}m-2i)} \\ &= 2^{-(\frac{1}{6}+\frac{\alpha}{3})m} \sum_{i=0}^{\frac{m}{6}} \binom{\frac{m}{6}}{i} \left(\frac{1}{4}\right)^i \left(\frac{3}{4}\right)^{\frac{m}{6}-i} \cdot 2^{2\alpha i} \end{aligned}$$

Wegen $\sum_{i=0}^n \binom{n}{i} p^i (1-p)^{n-i} 2^{\beta i} = (1-p+p2^\beta)^n$ (Lemma 67) ist diese Gleichung äquivalent zu

$$\begin{aligned} 2^{-\alpha m} &= 2^{-(\frac{1}{6}+\frac{\alpha}{3})m} \left(\frac{1}{4}\right)^{\frac{m}{6}} (3+2^{2\alpha})^{\frac{m}{6}} \\ \iff 2^{-6\alpha} &= 2^{-(1+2\alpha)} \frac{1}{4} (3+2^{2\alpha}) \\ \iff 2^{1-4\alpha} &= \frac{1}{4} (3+2^{2\alpha}) \\ \iff 2(2^{2\alpha})^{-2} - \frac{1}{4}(2^{2\alpha}) - \frac{3}{4} &= 0 \end{aligned}$$

Durch die Substitution $u := 2^{2\alpha}$, d.h. $\alpha = \frac{1}{2} \log u$, erhält man

$$u^3 + 3u^2 - 8 = 0$$

□

Lemma 67. Für alle $n \in N \cup \{0\}$, $p \in (0, 1)$ und $\beta \in R$ gilt

$$\sum_{i=0}^n \binom{n}{i} p^i (1-p)^{n-i} 2^{\beta i} = (1-p + p2^\beta)^n$$

Beweis. Für alle $n \in N \cup \{0\}$, $p \in (0, 1)$, $\beta \in R$ sei $f(n) := \sum_{i=0}^n \binom{n}{i} p^i (1-p)^{n-i} 2^{\beta i}$. Dann gilt

$$f(n) = (1-p)^n + \sum_{i=1}^n \binom{n}{i} p^i (1-p)^{n-i} 2^{\beta i}$$

Aus

$$\begin{aligned} \binom{n-1}{i} + \binom{n-1}{i-1} &= \frac{(n-1)!}{(n-1-i)!i!} + \frac{(n-1)!}{(n-i)!(i-1)!} \\ &= \frac{(n-1)!(n-i) + (n-1)!i}{(n-i)!i!} = \frac{n!}{(n-i)!i!} = \binom{n}{i} \end{aligned}$$

ergibt sich für $f(n)$

$$\begin{aligned} f(n) &= (1-p)^n + \sum_{i=1}^n \binom{n-1}{i} p^i (1-p)^{n-i} 2^{\beta i} + \sum_{i=1}^n \binom{n-1}{i-1} p^i (1-p)^{n-i} 2^{\beta i} \\ &= \sum_{i=0}^n \binom{n-1}{i} p^i (1-p)^{n-i} 2^{\beta i} + \sum_{i=0}^{n-1} \binom{n-1}{i} p^{i+1} (1-p)^{n-(i+1)} 2^{\beta(i+1)} \end{aligned}$$

Gemäß der Definition des Binomialkoeffizienten ist $\binom{n-1}{n} = 0$. Damit folgt

$$\begin{aligned} f(n) &= (1-p) \underbrace{\sum_{i=0}^{n-1} \binom{n-1}{i} p^i (1-p)^{n-1-i} 2^{\beta i}}_{f(n-1)} + p2^\beta \underbrace{\sum_{i=0}^{n-1} \binom{n-1}{i} p^i (1-p)^{n-1-i} 2^{\beta i}}_{f(n-1)} \\ &= (1-p + p2^\beta) f(n-1) \end{aligned}$$

Wegen $f(0) = 1$ folgt schließlich

$$f(n) = \prod_{i=1}^n (1-p + p2^\beta) = (1-p + p2^\beta)^n$$

□

7.7 Gesamtresultat für den A5/1 Schlüsselstromgenerator

In diesem Kapitel haben wir gezeigt, dass die zu Beginn der Arbeit allgemein für LFSR-basierte Schlüsselstromgeneratoren dargestellten Resultate auf den konkreten Fall des A5/1 Schlüsselstromgenerators übertragbar sind. Anhand der berechneten Informationsrate $\alpha \approx 0,2193$ und der *best case* Kompressionsrate $\gamma = \frac{1}{4}$ können wir nun die Laufzeit und den Speicherplatzbedarf für die Kryptanalyse des A5/1 Generators bestimmen:

Theorem 68. *Aus den ersten $[1, 14n]$ Schlüsselbits eines A5/1 Schlüsselstromgenerators der Schlüssellänge n kann in einer Laufzeit und mit einem Speicherplatzbedarf von $n^{O(1)}2^{0,6403n}$ der zugrunde liegende Initialzustand x berechnet werden. \square*

Für den Vergleich mit den empirischen Resultaten wollen wir schließlich den polynomiellen Faktor $n^{O(1)}$ genauer bestimmen:

Beobachtung 69. *Für den A5/1 Schlüsselstromgenerator und für alle $n' \leq m \leq \lceil \alpha^{-1}n \rceil$ gilt*

$$|P_m| \in O(n^{10}) \cdot 2^{0,6403n}$$

Beweis. Im Beweis von Lemma 49 hatten wir festgestellt, dass

$$|P_m| \leq p(m) \cdot 2^{\frac{1-\alpha}{1+\alpha}n}$$

mit $p(m) = |G_m^C|^2 \cdot |Q_m|$. Für den A5/1 Schlüsselstromgenerator haben wir in diesem Kapitel $|G_m^C| \in O(m^3)$ und $|Q_m| \in O(m^4)$ nachgewiesen (vgl. Lemma 53 bzw. Lemma 54). \square

Diese Schranke soll im Folgenden experimentell überprüft werden.

Kapitel 8

Implementation des FBDD-Pakets

Obwohl zahlreiche teilweise weit verbreitete OBDD-Implementationen existieren (vgl. etwa [Uni04] für eine erste Auswahl), die in verschiedenen akademischen und industriellen Anwendungen Verwendung finden, scheinen für FBDDs keine Implementationen verfügbar zu sein. Lediglich ein technischer Bericht der Universität Trier dokumentiert die Entwicklung eines FBDD-Pakets basierend auf der OBDD-Implementation CMU (vgl. [SM93]), dessen Quellcode allerdings nicht mehr verfügbar ist. Dennoch belegt [SM93], dass die fundamentalen Designprinzipien eines OBDD-Pakets auch zur Implementation von FBDDs ausgenutzt werden können. Da die Implementation von OBDDs bis heute ein aktuelles Forschungsthema ist und seit der Veröffentlichung der ersten OBDD-Pakete Anfang der 90er-Jahre deutliche Fortschritte in der Leistungsfähigkeit der Pakete erzielt werden konnten, wollen wir die in [SM93] dargestellten Erkenntnisse ausnutzen und unsere FBDD-Implementation mit Hilfe eines aktuellen OBDD-Pakets durchführen.

Als Entscheidungsgrundlage für die Auswahl eines OBDD-Pakets, das als Basis unserer Implementation dienen soll, wollen wir zunächst die grundlegenden Designprinzipien moderner OBDD-Implementationen zusammentragen.

8.1 Designkonzepte von OBDD-Implementationen

Trotz der großen Zahl verschiedener Implementationen basieren alle OBDD-Pakete auf wenigen Schlüsselkonzepten, die im Grundsatz bereits Anfang der 90er-Jahre in [BRB90] entwickelt wurden.¹ Diese Konzepte wollen wir ergänzt um aktuellere Forschungsergebnisse nun überblicksartig darstellen.

Für die Betrachtungen in diesen Abschnitt nehmen wir eine fixierte Variablenordnung π an, so dass alle OBDDs als π -OBDDs aufzufassen sind.

8.1.1 *shared* OBDDs und *Manager*

Eine OBDD-Implementation sollte natürlich in der Lage sein, mehrere OBDDs gleichzeitig zu verwalten. Dabei ist es offensichtlich effizient, alle benötigten OBDDs in demselben Graphen zu speichern, damit gemeinsame Untergraphen der OBDDs jeweils nur einmal repräsentiert werden müssen. Einen solchen,

¹Für eine gegenüber der Originalarbeit ausführlichere Darstellung vgl. [MT98].

als *shared* OBDD bezeichneten Graphen benutzen alle modernen OBDD-Pakete.

Um mehrere *shared* OBDDs gleichzeitig verwalten zu können, unterstützen einige OBDD-Pakete sogenannte *Manager*, in denen jeweils ein *shared* OBDD und die zugehörigen Hilfsdatenstrukturen zusammengefasst werden. Dazu zählen die im Folgenden erläuterten Tabellen *unique-table* und *computed-table* aber auch die Variablenordnung π , die für jeden *Manager* individuell festgelegt werden kann.

8.1.2 *unique-table*

Wir haben im Kapitel 4.3 festgestellt, dass jede Boolesche Funktion $f \in B_n$ durch einen reduzierten OBDD in kanonischer Weise dargestellt werden kann. Um den verfügbaren Speicherplatz effizient auszunutzen werden daher die OBDDs ausschließlich in reduzierter Form gespeichert. In Verbindung mit der Forderung, dass gemeinsame Untergraphen nur einmal im *shared* OBDD repräsentiert werden, erreicht man sogar eine eingdeutige Darstellung jeder Funktion $f \in B_n$ in dem Sinne, dass Funktionen $f, g \in B_n$ genau dann äquivalent sind, wenn sie durch denselben OBDD-Knoten, d.h. durch dasselbe Objekt im Speicher, repräsentiert werden.

Damit sichergestellt ist, dass diese strenge kanonische Form auch nach dem Einfügen von neuen Knoten in den *shared* OBDD erhalten bleibt, muss für einen einzufügenden Knoten v mit der Bezeichnung x_i und den Nachfolgerknoten v_1 und v_0 zunächst geprüft werden, ob ein solcher Knoten bereits existiert, d.h. ob die Funktion f_v bereits durch einen OBDD-Knoten repräsentiert ist. Nur wenn noch kein derartiger Knoten vorhanden ist, wird ein neuer Knoten erzeugt, ansonsten wird der bereits gespeicherte Knoten verwendet. Diese Vorgehensweise haben wir im Zusammenhang mit dem Synthesealgorithmus für FBDDs (vgl. Algorithmus 2) bereits kennengelernt.

Zur effizienten Implementation dieser Einfügeoperation verwenden die OBDD Pakete eine mit *unique-table* bezeichnete *Map*, die jedem Tripel $(x_i, v_1, v_0) \in X_n \times V \times V$ den zugehörigen Knoten $v \in V$ zuordnet. Um schnelle Zugriffszeiten zu garantieren, werden die *unique-tables* in vielen Fällen als *Hashtabellen* mit Kollisionslisten implementiert.

8.1.3 *computed-table*

OBDD-Implementationen müssen vor allem die effiziente Manipulation von als OBDDs gegebenen Booleschen Funktionen ermöglichen. Wir hatten im Kapitel 4 für unseren Anwendungskontext bereits die binäre Synthese als wichtige Manipulationsoperation identifiziert.

Die rekursive Struktur der OBDDs und der zugehörigen Operationen sowie die Tatsache, dass ein einzelner Teil-OBDD möglicherweise Bestandteil mehrerer OBDDs ist, führen dazu, dass u.a. im Laufe einer Syntheseoperation Teilergebnisse der Berechnung mehrfach verwendet werden. Um aufwendige Neuberechnungen dieser Ergebnisse zu vermeiden, benutzen OBDD-Implementationen einen als *computed-table* bezeichneten *Cache*, der einer Booleschen Operation $\otimes \in B_n$ und den zugehörigen Operanden $(f_1, \dots, f_n) \in B_{i_1} \times \dots \times B_{i_n}$ das Ergebnis dieser Operation in Form eines OBDD-Knotens v mit $f_v = \otimes(f_1, \dots, f_n)$ zuordnet. Die verschiedenen Operationen \otimes werden dabei meist mit ganzen Zahlen, sogenannten *opcodes* identifiziert.

Die *computed-table* wird in vielen Fällen als *Hashtabelle* ohne Kollisionslisten implementiert, d.h. für jeden Hashwert existiert nur höchstens ein Eintrag in der *Hashtabelle*, so dass existierende Einträge von neuen Einträgen, die denselben *Hashwert* besitzen, überschrieben werden. Eine empirisch motivierte Empfehlung für die Wahl der *Hashfunktion* findet sich etwa in [YBO⁺98].

Einerseits ist es sinnvoll, möglichst viele Zwischenergebnisse zu *cachen* um Rechenzeit zu sparen, andererseits führt das Speichern von nicht oder nur selten wiederverwendeten Ergebnissen zu einer ineffizienten Speicherplatzausnutzung. Ebenso sollten triviale Ergebnisse, d.h. Ergebnisse, die in konstanter Zeit neu berechnet werden können, nicht im *Cache* abgelegt werden. Die Bestimmung eines optimalen *Time-Space Tradeoffs* und die Minimierung der Anzahl von *Cache Misses*, d.h. der erfolglosen Suche nach Einträgen im *Cache*, sind Gegenstand vieler empirischer Untersuchungen. [Lon98] nutzt zum Beispiel aus, dass Elternknoten immer später als ihre Kindknoten erzeugt werden, und gruppiert mit Hilfe dieser Eigenschaft Knoten, die durch kurze Pfade in den OBDDs verbunden sind, in nahe beieinanderliegenden Speicherbereichen. Auf diese Weise können Suchzeiten in der *computed table* und auch in der *unique table* zum Teil deutlich verringert werden.

8.1.4 komplementierte Kanten

Man sieht leicht, dass sich die OBDD-Repräsentation einer Funktion $f \in B_n$ von der ihres Komplements \bar{f} nur dadurch unterscheidet, dass die Werte der beiden Senken vertauscht sind. Die meisten OBDD-Pakete nutzen diesen Erkenntnis aus, indem sie die OBDD-Kanten um ein Komplementbit erweitern und einen Teil-OBDD mit der Wurzel v , der mit einer komplementierten Kante referenziert wird, als Referenz auf die Funktion \bar{f}_v interpretieren. Ein durch eine nicht komplementierte (reguläre) Kante referenzierter Teil-OBDD wird dagegen als Darstellung der Funktion f_v aufgefasst. Die Funktion f und ihr Komplement \bar{f} können somit durch denselben OBDD-Knoten repräsentiert werden. Insbesondere ist nur noch eine Senke erforderlich, da die zweite Senke als Komplement der ersten dargestellt werden kann. So kann zum Beispiel die konstante Nullfunktion durch eine komplementierte Referenz auf die 1-Senke repräsentiert werden.

Wenn wir einen Knoten mit einer eingehenden und zwei ausgehenden Kanten betrachten, erkennen wir, dass es insgesamt 2^8 Möglichkeiten gibt, Komplementbits auf die Kanten zu verteilen. Diese Möglichkeiten lassen sich jedoch in vier Äquivalenzklassen partitionieren, wie in Abbildung 8.1 dargestellt.

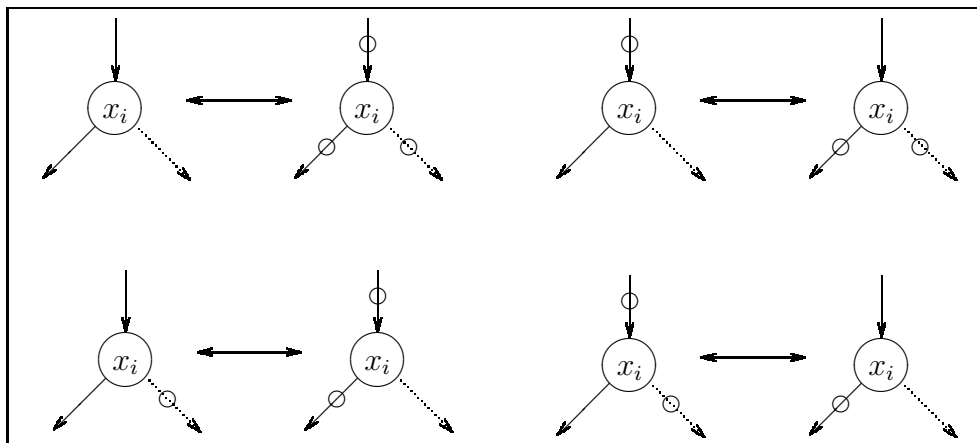


Abbildung 8.1: Äquivalenzklassen bezüglich Komplementbits

Um die Kanonizität der Darstellung nicht zu verlieren, lässt man jeweils nur eine der äquivalenten Darstellungen zu. Viele OBDD Pakete treffen daher die Festlegung, dass die 1-Kante eines Knotens immer regulär sein muss. Auf diese Weise wird für jede der vier Äquivalenzklassen jeweils die linke Darstellung als Repräsentant definiert.

Wenn ein neuer Knoten eingefügt wird, müssen die Komplementbits ggf. durch Äquivalenzumformung in

die geforderte Darstellung gebracht werden.

8.1.5 Implementation von Knoten und Referenzen

Da OBDDs als Spezialfälle allgemeiner Graphen $G = (V, E)$ aufgefasst werden können, werden die Knoten $v \in V$ zunächst als Objekte (z.B. in der Programmiersprache C als `structs`) gespeichert. Eine naheliegende Möglichkeit, die Referenzen auf die Nachfolgerknoten eines Knotens v zu verwalten, ist die Integration von zwei Zeigervariablen, die die Speicheradresse der Nachfolgerknoten beinhalten, in das zu v gehörende Knotenobjekt. Aus dieser zeigerbasierten Methode, die einen Knoten mit seiner Adresse im Speicher identifiziert, ergeben sich jedoch die folgenden Nachteile:

- Bei der Portierung eines zeigerbasierten OBDD-Pakets von einer Maschine mit 32 Bit Registerbreite auf eine 64-Bit Maschine verdoppeln sich auch die Bitlängen der Speicheradressen von 32 auf 64 Bit, d.h. zur Verwaltung der Referenzen wird die doppelte Menge an Speicherplatz benötigt.
- Die Speicheradressen werden im Wesentlichen durch das Speichermanagement des Betriebssystems festgelegt und können durch das OBDD-Paket nicht unmittelbar beeinflusst werden, so dass die Knoten im Allgemeinen bei jedem Programmaufruf verschiedene Speicheradressen erhalten. Falls die *Hashfunktionen* der *unique-table* und der *computed-table* die *Hashwerte* der Knoten anhand ihrer Speicheradressen bestimmen, entstehen dadurch unterschiedliche Zugriffsmuster auf die *Cache*-Einträge, die wiederum die *Cache*-Heuristiken für die Vergrößerung des *Cache* beeinflussen, usw. Insgesamt verhält sich also jede Ausführung desselben deterministischen OBDD-Programms unterschiedlich. Fehler sind somit nicht mehr ohne weiteres reproduzierbar und können schwerer lokalisiert werden.
Abhilfe schaffen hier zum Beispiel die in [Lon98] vorgeschlagenen adressunabhängigen *Hashfunktionen*.
- Aus der Speicheradresse zweier Knoten können im Allgemeinen keinerlei semantische Informationen wie der Abstand der Knoten, Vorgänger-Nachfolger- und Nachbarschaftsbeziehungen etc. gewonnen werden, die zum Beispiel durch den *Cache* ausgenutzt werden könnten.

Verschiedene OBDD-Pakete verwenden aus diesen Gründen nicht die Speicheradresse zur Identifikation der Knoten sondern vergeben für jeden Knoten $v \in V$ einen eindeutigen Index $i(v) \in N$ - der Einfachheit halber den Index innerhalb des Feldes, in dem die Knotenobjekte verwaltet werden. Anstatt über die Speicheradresse werden die Nachfolgerknoten innerhalb eines Knotenobjekts dann über ihren Index referenziert. In Verbindung mit Konsistenzregeln wie

$$i(v) < i(w) \iff v \text{ ist früher als } w \text{ erzeugt worden}$$

lässt sich sicherstellen, dass jeder Knoten in einem deterministischen OBDD-Programm in jeder Ausführung und auf jeder Plattform durch denselben Index identifiziert wird und damit auch die *Hashfunktionen* nicht durch wechselnde Parameter beeinflusst werden. [Jan01] beobachtet, dass sich auf diese Weise die von [Lon98] entwickelte Beschleunigung der *Cache*-Operationen auch mit einem indexbasierten OBDD-Paket realisieren lässt. Ebenso lassen sich komplementierte Kanten in diesen Ansatz etwa durch die Komplementierung des *most significant Bits* der Indexvariablen integrieren.

8.1.6 Speicherverwaltung

In vielen Anwendungen werden OBDDs *bottom-up* konstruiert, d.h. der benötigte OBDD wird schrittweise aus kleineren Teil-OBDDs zusammengesetzt, die lediglich den Charakter von Zwischenergebnissen haben

und für die weitere Berechnung nicht mehr von Bedeutung sind. Solche OBDDs können und sollten im Sinne einer effizienten Speicherverwaltung also aus dem *shared* OBDD gelöscht werden.

Allerdings werden die Knoten in einem *shared* OBDD im Allgemeinen von mehreren anderen Knoten referenziert, so dass für einen OBDD, den der Benutzer nicht mehr benötigt, nicht gewährleistet ist, dass jeder seiner Knoten direkt gelöscht werden kann. Vor dem endgültigen Löschen eines Knotens muss also in der *unique table* nach Knoten gesucht werden, die den zu löschenden Knoten referenzieren. Ausserdem werden einzelne Knoten des zu löschenden OBDDs eventuell aus der *computed table* heraus als Ergebnis von Syntheseoperationen o.ä. referenziert. Solche Einträge der *computed table* müssen ebenfalls gelöscht werden.

Die Suche in der *unique table* und der *computed table* nach den entsprechenden Einträgen ist im Allgemeinen zeitaufwendig und kann nur durch Speichern von zusätzlichen Informationen beschleunigt werden. Wenn der Benutzer einen OBDD freigibt, werden daher die zu löschenden Knoten lediglich markiert und erst dann endgültig gelöscht, wenn sich so viele markierte Knoten angesammelt haben, dass der Zeitaufwand für die Löschoptionen in einem vernünftigen Verhältnis zum freiwerdenden Speicherplatz steht. Für diese als *garbage collection* bezeichnete Strategie existieren im Wesentlichen zwei Vorgehensweisen:

- (i) *Reference counting garbage collection* verwaltet für jeden Knoten $v \in V$ einen Referenzzähler, der mit jeder neu hinzugefügten Referenz auf v inkrementiert und mit jeder gelöschten Referenz auf v dekrementiert wird. Falls ein Referenzzähler für einen Knoten $v \in V$ auf Null dekrementiert wird, werden auch die Referenzzähler aller Knoten des OBDDs mit der Wurzel v dekrementiert. Während der *garbage collection* werden genau diejenigen Knoten aus dem *shared* OBDD gelöscht, deren Referenzzähler gleich Null ist.
- (ii) *Mark and sweep garbage collection* führt nicht laufend Buch über die Anzahl der Referenzen auf jeden einzelnen Knoten. Zu Beginn der *garbage collection* werden stattdessen ausgehend von den Wurzeln der vom Benutzer benötigten OBDDs mittels Breiten- oder Tiefensuche alle referenzierten Knoten markiert (*mark*). Alle nicht markierten Knoten des *shared* OBDD werden nicht mehr referenziert und daher im nächsten Schritt gelöscht (*sweep*).

Gegenüber *reference counting* besitzt *mark and sweep* den Vorteil, dass durch den Wegfall der Referenzzähler deren Speicherplatz sowie die Zeit für die Inkrementierungs- und Dekrementierungsoperationen eingespart werden können, andererseits stehen zwischen den *garbage collections* keine Informationen mehr über die Anzahl der Referenzen auf einen bestimmten Knoten zur Verfügung.

Durch geeignete Regruppierung der Knoten im Speicher kann mit beiden Verfahren während der *garbage collection* die Effizienz späterer Hauptspeicher- und *Cachezugriffe* gesteigert werden (vgl. [Lon98] und [Jan01]).

8.2 Auswahl des OBDD-Basispakets

Wir haben im Kapitel 4 beobachtet, dass der einzige wesentliche Unterschied zwischen OBDDs und FBDDs darin besteht, dass die Einleseordnung der Variablen durch einen Steuerungsgraphen und nicht durch eine Permutation der Indexmenge gegeben ist. Somit sind alle im vorigen Abschnitt vorgestellten Designkonzepte von OBDD-Implementationen prinzipiell auch auf FBDDs anwendbar.

OBDD-Pakete, die auf diesen Designprinzipien beruhen, sind also generell als Basispakete für unsere FBDD-Implementation geeignet, müssen im Detail jedoch einige zusätzliche Voraussetzungen erfüllen, die wir nun formulieren wollen.

8.2.1 Anforderungen an OBDD-Pakete

Das OBDD-Paket sollte die folgenden Eigenschaften besitzen (Musskriterien):

- Der Steuerungsgraph ist in den *shared* OBDD bzw. *shared* FBDD integrierbar. Dazu muss die *unique-table* Tripel der Form $(x_i, v, v) \in X_n \times V \times V$ verarbeiten können, die innerhalb des Steuerungsgraphen wegen der Nichtanwendbarkeit der *elimination rule* auftreten können.
- Die *computed-table* unterstützt Boolesche Funktionen mit mindestens drei Operanden, so dass der Steuerungsgraph als Parameter der Syntheseoperationen verarbeitet werden kann.
- Die Basisalgorithmen des Pakets basieren nicht auf einer globalen Ordnung der Variablen, da diese bei FBDDs definitionsgemäß nicht gegeben ist.
- Der Quellcode ist frei verfügbar und ausreichend dokumentiert.
- Das Paket implementiert effiziente Algorithmen für den *Cache* und das Speichermanagement.
- Das Paket wird laufend weiterentwickelt und an aktuelle Forschungsergebnisse angepasst.

Die nachfolgend genannten Eigenschaften sind wünschenswert, jedoch nicht zwingend erforderlich (Wunschkriterien):

- Die FBDD-Implementation ist ohne Veränderung des Paketquellcodes möglich.
- Es können möglichst viele Knoten verwaltet werden.
- Der Speicherplatzbedarf eines einzelnen Knotens ist möglichst klein.

8.2.2 Frei verfügbare OBDD-Pakete

Motiviert durch die grundlegenden theoretischen Resultate in [Bry86] und erste generische Implementationsbeschreibungen in [BRB90] sind gerade in den Jahren zwischen 1990 und 2000 zahlreiche OBDD-Pakete entstanden. Zum einen existieren verschiedene kommerzielle Pakete wie TiGeR ([CMT93]) oder die Nachfolgeversionen der ursprünglich frei verfügbaren Pakete EHV ([Jan98]) und CMU ([Lon93]).

Aufgrund beschränkter finanzieller Ressourcen kommen für unsere FBDD-Implementation allerdings nur frei verfügbare OBDD-Pakete in Frage. In diesem Segment sind unter anderem die folgenden Pakete vertreten:

- ABCD ([Bie98], experimentell), ETH Zürich
- PBF ([YCBO98], experimentell), Carnegie Mellon University in Pittsburgh
- CAL ([RS97]), University of California in Berkeley
- CUDD ([Som01]), University of Colorado in Boulder
- BuDDy ([LN02]), IT-University of Copenhagen

Auf den Seiten des BDD-Portals der Universität Trier ([Uni04]) bzw. in [Sen96] und [YBO⁺98] befinden sich Zusammenstellungen der wichtigsten Eigenschaften der verschiedenen Pakete.

Die Pakete ABCD und PBF eignen sich wegen ihres experimentellen Status zumindest im ersten Ansatz nicht für eine FBDD-Implementation. Das CAL-Paket scheint nicht mehr weiterentwickelt zu werden, denn der Quellcode der auf der *Homepage* der Autoren erhältlichen Version 2.1 ist datiert mit Oktober 1998.

Die aktuelle Version 2.2 des BuDDy-Pakets ist 2002 veröffentlicht worden, und die zugehörige *Homepage* wurde zuletzt im Januar 2003 geändert. Zudem ist auf der *Homepage* eine Liste von akademischen Projekten angegeben, in denen BuDDy erfolgreich eingesetzt wurde, so dass von einer gewissen Robustheit des Pakets ausgegangen werden kann. BuDDy verwaltet Knoten und Kanten indexbasiert und verwendet *mark and sweep garbage collection*, allerdings belegen die Knoten mit 20 Bytes im Vergleich zu den übrigen Paketen relativ viel Speicherplatz. Trotzdem können bis zu 2^{32} Knoten verwaltet werden.

Das CUDD-Paket ist seit Januar 2004 in der Version 2.4.0 verfügbar, in der gegenüber der Vorgängerversion 2.3.1 einige Fehler entfernt sowie verbesserte *garbage collection* Algorithmen implementiert wurden. CUDD arbeitet zeigerbasiert und mit einer *reference counting garbage collection*. Ein Knoten kann mit 16 Bytes repräsentiert werden, insgesamt verarbeitet CUDD bis zu 2^{28} Knoten. Im Gegensatz zu BuDDy ist die Implementation des Pakets ausführlich in einem *Programmer's Manual* dokumentiert, so dass in kurzer Zeit Einblick in die Details des Pakets gewonnen werden kann. Auch der Quellcode ist in CUDD übersichtlicher und besser dokumentiert als in BuDDy.

Ein älterer Vergleich der Pakete CUDD, CAL und CMU (vgl. [Sen96]) bescheinigt CUDD einen guten Kompromiss zwischen benötigter CPU-Zeit und Speicherplatzverbrauch, und Experimente in [YBO⁺98] belegen eine effizientere *Cache*-Heuristik im Vergleich zu verschiedenen anderen Paketen. Die in [Lon98] und [Jan01] beschriebenen Verbesserungen durch eine altersabhängige Ordnung der Knoten und indexbasiertes Speichermanagement scheinen zumindest älteren Versionen von CUDD überlegen zu sein, allerdings sind die den jeweiligen Arbeiten zugrundeliegenden OBDD-Pakete nicht öffentlich verfügbar.

Insgesamt erscheint CUDD daher als eine geeignete Wahl für die Erstimplementation eines FBDD-Pakets. Da sich CUDD und BuDDy jedoch sowohl bezüglich des Speichermanagements als auch in der Verwaltung der Knoten und Kanten grundlegend unterscheiden, sollte eine Vergleichsimplementation basierend auf BuDDy in einem nächsten Schritt angestrebt werden. Ebenso könnte auch die Eignung der experimentellen Pakete ABCD und PBF für eine FBDD-Implementation genauer überprüft werden.

8.3 Design und Implementation des FBDD-Pakets

Obwohl wir uns im vorherigen Abschnitt zunächst für ein bestimmtes OBDD-Paket als Basis für unsere FBDD-Implementation entschieden haben, legt die gemeinsame Grundstruktur der OBDD-Pakete nahe, unser FBDD-Paket so entwerfen, dass das zugrundeliegende OBDD-Paket ohne Veränderung der FBDD-Implementation ausgetauscht werden kann. Dazu ist es notwendig, von den konkreten Datenstrukturen und Algorithmen einzelner OBDD-Implementationen zu abstrahieren und eine gemeinsame Schnittstelle zu entwerfen, an die eine konkrete OBDD-Implementation durch einen Adapter (*Wrapper*, vgl. [GHJV96]) angepasst wird. Die FBDD-Implementation benutzt dann ausschließlich diese Schnittstelle, um mit dem OBDD-Paket zu kommunizieren.

Diese Vorgehensweise erzeugt gegenüber einer direkten Verwendung eines bestimmten OBDD-Pakets einen gewissen *Overhead*, den wir aber zu Gunsten leichter Wart- und Erweiterbarkeit im ersten Schritt in Kauf nehmen wollen.

Da die große Mehrzahl der OBDD-Pakete in der Programmiersprache *C* implementiert ist und unsere Problemstellung eher prozeduralen als objektorientierten Charakter hat, wollen wir unser FBDD-Paket ebenfalls in *C* implementieren und auch aus Effizienzgründen auf eine objektorientierte Umsetzung in *C++* verzichten.

Wir beschreiben daher direkt anhand der *Header*-Dateien die Definition der benötigten Schnittstellen und die Umsetzung des Adapter-Musters mit den Mitteln der Programmiersprache *C*.

In Analogie zu den OBDD-Paketen, die OBDDs immer als BDDs referenzieren, sollen auch in unserer Darstellung und im Programmquelltext mit der Bezeichnung BDD im Folgenden immer OBDDs gemeint sein. Weiterhin legen wir fest, dass alle OBDDs definiert sind über der Variablenmenge $\{x_0, \dots, x_{n-1}\}$ für ein $n \in \mathbb{N}$.

8.3.1 BDD-Adapter

Der BDD-Adapter versieht die OBDD-Pakete mit einer einheitlichen Schnittstelle und ist verteilt auf die *Header*-Dateien `bddData.h` und `bddAdapter.h`.

`bddData.h`

Die *Header*-Datei `bddData.h` beschreibt die Schnittstelle zu den Datenstrukturen eines OBDD-Pakets.

```
typedef void* BddNodePtr
typedef void* BddManagerPtr
```

definieren generische Zeiger auf einen Knoten des BDDs bzw. auf den BDD-*Manager*. Der *C*-Standard verbietet die Dereferenzierung von `void`-Zeigern, so dass der Benutzer keine Möglichkeit hat, direkt auf die Komponenten der referenzierten Objekte zuzugreifen. Für den Zugriff auf die Komponenten stellt der Adapter die folgenden Funktionen zur Verfügung:

- `BddNodePtr bddRegular(BddNodePtr node)` gibt eine nicht komplementierte (reguläre) Version der Referenz `node` zurück.
- `BddNodePtr bddNot(BddNodePtr node)` negiert die Referenz auf den Knoten `node`, d.h. eine komplementierte Referenz wird zu einer regulären Referenz und eine reguläre zu einer komplementierten.
- `int bddIndex(BddNodePtr node)` gibt den Index der Variablen zurück, mit der `node` bezeichnet ist. `node` muss eine reguläre Referenz sein.
- `int bddIsConstant(BddNodePtr node)` gibt einen Wert ungleich 0 (entspricht *true* in der Programmiersprache *C*) zurück, falls durch `node` eine Senke referenziert wird, sonst den Wert 0.
- `BddNodePtr bddThen(BddNodePtr node)` gibt eine Referenz auf den 1-Nachfolger von `node` zurück. `node` muss eine reguläre Referenz sein.
- `BddNodePtr bddElse(BddNodePtr node)` gibt eine Referenz auf den 0-Nachfolger von `node` zurück. `node` muss eine reguläre Referenz sein.
- `int BddIsComplement(BddNodePtr node)` gibt einen Wert ungleich 0 (*true*) zurück, falls die Referenz `node` komplementiert ist, sonst den Wert 0.

`bddAdapter.h`

Die *Header*-Datei `bddAdapter.h` beschreibt die Auswertungs- und Manipulationsoperationen für BDDs.

```
typedef void* BddLocalCachePtr
```

definiert einen Zeiger auf einen lokalen, von der *computed-table* des *Managers* unabhängigen *Cache*, in dem Ergebnisse von Funktionen mit beliebig langer Parameterliste gespeichert werden können.

Darüber hinaus definiert `bddAdapter.h` die folgenden Funktionen zur Auswertung und Manipulation von BDDs:

- `BddManagerPtr bddInit(unsigned int numVars, unsigned int numSlots, unsigned int cacheSize, unsigned long maxMemory)`
gibt einen Zeiger auf einen neu erzeugten `BddManager` zurück, der die folgenden Parameter besitzt:
 - `numVars`: Initialwert für die Anzahl der verschiedenen Variablen
 - `numSlots`: Initialwert für die maximale Anzahl von Knoten, die mit denselben Variablen bezeichnet sind
 - `cacheSize`: Initialwert für die Anzahl der Einträge in der *computed-table*
 - `maxMemory`: Speicherplatz in Bytes, der vom erzeugten `BddManager` maximal belegt wird
- `BddManagerPtr bddInitDefault(void)` erzeugt einen `BddManager` mit Standardwerten für die Parameter von `bddInit`.
- `void bddQuit(BddManagerPtr manager)` gibt den durch einen `BddManager` belegten Speicherplatz wieder frei.
- `BddNodePtr bddIthVar(BddManagerPtr manager, int i)` gibt eine Referenz auf einen Knoten zurück, der die Projektionsfunktion zur Variablen x_i repräsentiert, d.h. der Knoten ist mit x_i bezeichnet, seine 1-Kante referenziert die 1-Senke, und die 0-Kante ist eine komplementierte Referenz auf die 1-Senke. Falls ein solcher Knoten bereits in der *unique-table* enthalten ist, wird eine Referenz auf den existierenden Knoten zurückgegeben, ansonsten wird ein neuer Knoten erzeugt und der *unique-table* hinzugefügt.
- `BddNodePtr bddUniqueInter(BddManagerPtr manager, int index, BddNodePtr thenNode, BddNodePtr elseNode)`
gibt eine Referenz auf den Knoten mit der Variablen x_{index} als Bezeichnung, dem `thenNode` als 1-Nachfolger und dem `elseNode` als 0-Nachfolger zurück. Falls ein solcher Knoten in der *unique-table* bereits existiert, wird eine Referenz auf den existierenden Knoten zurückgegeben, ansonsten wird ein neuer Knoten erzeugt und der *unique-table* hinzugefügt.
- `BddNodePtr bddOne(BddManagerPtr manager)` gibt eine Referenz auf die 1-Senke zurück.
- `bddCacheInsert2(BddManagerPtr manager, unsigned int opcode, BddNodePtr f, BddNodePtr g, BddNodePtr data)`
fügt `data` als Ergebnis der durch `opcode` $\in \{0, \dots, 7\}$ gegebenen Operation angewendet auf die Argumente `f` und `g` in die *computed-table* ein.
- `BddNodePtr bddCacheLookup2(BddManagerPtr manager, unsigned int opcode, BddNodePtr f, BddNodePtr g)`
gibt das in der *computed-table* gespeicherte Resultat der durch `opcode` $\in \{0, \dots, 7\}$ gegebenen Operation angewendet auf die Argumente `f` und `g` zurück. Gibt `NULL` zurück, falls die *computed-table* keinen solchen Eintrag enthält.

- `void bddCacheInsert3(BddManagerPtr manager, unsigned int opcode, BddNodePtr f, BddNodePtr g, BddNodePtr h, BddNodePtr data)`
analog zu `bddCacheInsert2` mit `h` als zusätzlichem Funktionsargument
- `BddNodePtr bddCacheLookup3(BddManagerPtr manager, unsigned int opcode, BddNodePtr f, BddNodePtr g, BddNodePtr h)`
analog zu `bddCacheLookup2` mit `h` als zusätzlichem Funktionsargument
- `void bddCacheFlush(BddManagerPtr manager)` leert die *computed-table* des übergebenen `BddManagers`.
- `BddLocalCachePtr bddLocalCacheInit(BddManagerPtr manager, unsigned int keySize, unsigned int cacheSize, unsigned int maxCacheSize)`
erzeugt einen lokalen *Cache* für Funktionen mit `keySize` Argumenten, dessen Initialgröße bzw. Maximalgröße `cacheSize` bzw. `maxCacheSize` Einträge beträgt.
- `bddLocalCacheQuit(BddManagerPtr manager, BddLocalCachePtr cache)` gibt den durch `cache` belegten Speicherplatz frei.
- `void bddLocalCacheInsert(BddLocalCachePtr cache, BddNodePtr *key, BddNodePtr data)`
fügt das Ergebnis `data` in den `cache` ein. `key` ist ein Zeiger auf den Anfang der als Feld gegebenen Parameterliste. Die Länge der Parameterliste muss dem Wert `keySize` entsprechen, der bei der Initialisierung von `cache` angegeben wurde.
- `BddNodePtr bddLocalCacheLookup(BddLocalCachePtr cache, BddNodePtr *key)`
gibt das in `cache` gespeicherte Resultat zur in `key` übergebenen Parameterliste zurück bzw. `NULL`, falls für die gegebenen Parameter kein Ergebnis gespeichert ist. Genau wie für `bddLocalCacheInsert` ist `key` ein Zeiger auf den Anfang der als Feld gegebenen Parameterliste.
- `double bddSatCount(BddManagerPtr manager, BddNodePtr f, double nvars)`
gibt die Anzahl der erfüllenden Variablenbelegungen der durch den Knoten `f` repräsentierten Funktion zurück, unter der Annahme, dass `f` von den Variablen $x_0, \dots, x_{nvars-1}$ abhängt.
- `void bddRef(BddManagerPtr manager, BddNodePtr f)` inkrementiert den Referenzzähler von `f`. Falls dadurch der Wertebereich der Variablen, in der der Referenzzähler gespeichert wird, verlassen würde, wird der Referenzzähler nicht verändert. Falls das unterliegende BDD-Paket keine Referenzzähler verwendet, bleibt diese Operation ohne Effekt.
Die Inkrementierung des Referenzzählers ist auch für Zwischenergebnisse notwendig, damit nicht referenzierte, aber vom Benutzer noch benötigte BDDs nicht zwischenzeitlich durch *garbage collections* gelöscht werden.
- `void bddDeref(BddManagerPtr manager, BddNodePtr f)` dekrementiert den Referenzzähler von `f`, falls dieser noch nicht gleich Null ist. Falls das unterliegende BDD-Paket keine Referenzzähler verwendet, bleibt diese Operation ohne Effekt.
- `void bddRecursiveDeref(BddManagerPtr manager, BddNodePtr f)` dekrementiert den Referenzzähler von `f`, falls dieser noch nicht gleich Null ist. Wenn der Zähler durch die Dekrementierung den Wert Null erreicht, wird `bddRecursiveDeref` für beide Nachfolgerknoten von `f` aufgerufen, da

`f` nun von keinem anderen Knoten mehr referenziert wird. Falls das unterliegende BDD-Paket keine Referenzzähler benutzt, bleibt die Funktion ohne Effekt.

Analog zu `bddRef` sollten auf der anderen Seite nicht mehr benötigte BDDs freigegeben werden, um nicht unnötig Speicherplatz zu blockieren.

- `void bddMinterm(BddManager manager, BddNodePtr f, int **resultList)`
schreibt die Minterme der Funktion f , die durch die einzelnen Pfade zur 1-Senke gegeben sind, in den von `resultList` referenzierten Speicherbereich. Dabei ist `resultList[i][j]` der Wert der Variablen x_j in Minterm i . Die Funktion setzt voraus, dass der übergebene Speicherbereich ausreichend groß ist.
- `int bddSize(BddNodePtr f)` gibt die Anzahl der Knoten in `f` zurück
- `double bddCountPathsToNonZero(BddNodePtr f)` gibt die Anzahl der Pfade in `f` zurück, die zu einer Senke ungleich der Null-Senke führen.

Die durch `bddData.h` und `bddAdapter.h` definierte Schnittstelle wurde für das CUDD-Paket in der Datei `bddAdapterCudd.c` implementiert.

Änderungen am Quellcode von CUDD

Nach einer *garbage collection* müssen Einträge in den *local caches*, die Referenzen auf gelöschte Knoten enthalten, aus den *local caches* entfernt werden. Im OBDD-Paket CUDD, das wir für unsere FBDD-Implementation ausgewählt haben, ist hierfür die Funktion `cuddLocalCacheClearDead` in der Datei `cuddLCache.c` zuständig, die nach jeder *garbage collection* durch das Paket aufgerufen wird, in der Originalversion jedoch zu Speicherzugriffsfehlern bei leeren *Cache*-Einträgen führt. Dieser Umstand macht kleinere Änderungen am Quelltext der Funktion erforderlich, die in Anhang A abgedruckt sind.

8.3.2 Schnittstelle des FBDD-Pakets

Die *Header*-Dateien `fbdd.h` und `fbddInt.h` definieren die Schnittstelle des FBDD-Pakets.

`fbdd.h`

In der Datei `fbdd.h` werden die FBDD-Datenstrukturen und -Funktionen definiert, die sich nicht auf die innere Struktur des FBDD-Pakets beziehen.

```
typedef struct {
    BddNodePtr oracle;
    BddManagerPtr bddManagerPtr;
} FbddManager;
```

erweitert den `BddManager` des unterliegenden BDD-Pakets um eine Referenz auf den Steuerungsgraphen, der genau wie die Variablenordnung bei den OBDDs global für alle FBDDs gültig ist, die durch den *Manager* verwaltet werden.

typedef FbddManager *FBDDManagerPtr definiert einen Zeiger auf einen FBDDManager.

Neben diesen Datenstrukturen sind analog zu den BDD-Funktionen in `bddAdapter.h` die folgenden Funktionen in `fbdd.h` definiert:

- FbddManager *fbddInit(unsigned int numVars, unsigned int numSlots, unsigned int cacheSize, unsigned long maxMemory)
- FbddManager *fbddInitDefault()
- void fbddQuit(FbddManager *manager)
- BddNodePtr fbddIthVar(FbddManager *manager, int i)
- BddNodePtr fbddOne(FbddManager *manager)
- double fbddSatCount(FbddManager *manager, BddNodePtr f, double nvars)
- void fbddMinterm(FbddManager *manager, BddNodePtr f, int **resultList)
- int fbddSize(BddNodePtr f)
- double fbddCountPathsToNonZero(BddNodePtr f)
- void fbddRef(FbddManager *manager, BddNodePtr f)
- void fbddDeref(FbddManager *manager, BddNodePtr f)
- void fbddRecursiveDeref(FbddManager *manager, BddNodePtr f)

Zusätzlich definiert `fbdd.h` die folgenden Funktionen:

- BddNodePtr fbddEval(FbddManager *manager, BddNodePtr f, char *input)
wertet den gegebenen FBDD f für die Variablenbelegung `input` aus. `input` ist gegeben als `char`-Feld. Dabei entspricht das höchstwertige Bit (Bit 7) von `input[0]` der Belegung b_0 der Variablen x_0 , das nächstniedrigere Bit (Bit 6) von `input[0]` der Belegung b_1 der Variablen x_1 , usw. `input` muss für alle Variablen x_i , die in f eingelesen werden, eine Belegung b_i enthalten.
- BddNodePtr fbddAnd(FbddManager *manager, BddNodePtr oracle, BddNodePtr f, BddNodePtr g)
gibt den FBDD zurück, der die Funktion $f \cdot g$ repräsentiert.

fbddInt.h

Die *Header*-Datei `fbddInt.h` definiert diejenigen Datenstrukturen und Funktionen, die sich auf die *unique-table*, die *computed-table* und lokale *Caches*, d.h. auf die innere Struktur des FBDD-Pakets beziehen.

```
typedef struct FbddLocalCache{
    void* bddLocalCache;
#ifdef FBDD_CACHE_STATS
    unsigned long hits;
```

```

    unsigned long misses;
#endif
} FbddLocalCache;

```

definiert einen lokalen FBDD-*Cache*. Bei Bedarf kann durch Definition des *C*-Präprozessorsymbols `FBDD_CACHE_STATS` die Zählung von *cache hits* und *cache misses*, d.h. die Zählung von erfolgreichen bzw. erfolglosen Suchoperationen aktiviert werden.

Folgende, zu `bddAdapter.h` analoge Funktionsdefinitionen sind in `fbddInt.h` enthalten:

- `BddNodePtr fbddUniqueInter(FbddManager *manager, int index, BddNodePtr thenNode, BddNodePtr elseNode)`
- `void fbddCacheFlush(FbddManager *manager)`
- `void fbddCacheInsert1(FbddManager *manager, unsigned int opcode, BddNodePtr oracle, BddNodePtr f, BddNodePtr data)`
- `BddNodePtr fbddCacheLookup1(FbddManager *manager, unsigned int opcode, BddNodePtr oracle, BddNodePtr f)`
- `void fbddCacheInsert2(FbddManager *manager, unsigned int opcode, BddNodePtr oracle, BddNodePtr f, BddNodePtr g, BddNodePtr data)`
- `BddNodePtr fbddCacheLookup2(FbddManager *manager, unsigned int opcode, BddNodePtr oracle, BddNodePtr f, BddNodePtr g)`
- `FbddLocalCache *fbddLocalCacheInit(FbddManager *manager, unsigned int keySize, unsigned int cacheSize, unsigned int maxCacheSize)`
- `void fbddLocalCacheQuit(FbddManager *manager, FbddLocalCache *cache)`
- `void fbddLocalCacheInsert(FbddLocalCache *cache, BddNodePtr *key, BddNodePtr data)`
- `BddNodePtr fbddLocalCacheLookup(FbddLocalCache *cache, BddNodePtr *key);`

`void fbddLocalCacheStats(FbddLocalCache *cache)` gibt die Anzahl der *cache hits* und *cache misses* für den gegebenen *cache* aus, falls das Präprozessorsymbol `FBDD_CACHE_STATS` definiert ist.

Eine generische, nur über den `bddAdapter` mit dem unterliegenden OBDD-Paket kommunizierende FBDD-Implementation befindet sich in `fbddGeneric.c` bzw. `fbddIntGeneric.c`. Der schon angesprochene *overhead* bei der Kommunikation mit dem OBDD-Paket über einen Adapter könnte - allerdings auf Kosten der Portabilität - in einem nächsten Schritt durch eine stärker mit einem bestimmten OBDD-Paket verflochtene Implementation der FBDD-Schnittstellen verringert werden.

Kapitel 9

Implementation der FBDDs für die Kryptanalyse

Nach der Darstellung der Implementation des FBDD-Pakets wollen wir nun auf die wichtigsten Designentscheidungen bei der Implementation der eigentlichen Kryptanalyse eingehen.

9.1 Berechnung des Steuerungsgraphen G_m^C und des G_m^C -FBDDs Q_m

Der Steuerungsgraph G_m^C kann basierend auf den Überlegungen in Kapitel 7.2 effizient *bottom-up* in einer Laufzeit und mit einem Speicherplatzbedarf von $O(|G_m^C|)$ berechnet werden (vgl. Algorithmus 9 für eine Darstellung in Pseudocode). In der Implementation benutzen wir die 1-Senke als *-Senke und codieren die Parameter der rekursiven Unterfunktionen in einen einzigen `unsigned int`. Da die Zwischenergebnisse der *bottom-up* Konstruktion von G_m^C nicht für einen Steuerungsgraphen $G_{m'}^C$ mit $m' > m$ wiederverwendet werden können, benutzen wir anstatt der globalen *computed table* einen lokalen *Cache*, der für jede Steuerungsgraphenberechnung neu initialisiert wird.

Alternativ kann man den Steuerungsgraphen G_{m+1}^C auch aus dem Steuerungsgraphen G_m^C durch Fortsetzung an den Blättern berechnen, die zur *-Senke führen. Dies ist allerdings nur auf den ersten Blick sinnvoll, denn wir benötigen für dieses Vorgehen den nicht minimierten Steuerungsgraphen G_m^C und zudem in jedem Blatt die Information, wie der Steuerungsgraph an dieser Stelle fortgesetzt werden muss. Man kann diese Information zwar mit Hilfe von *Algebraic Decision Diagrams* (ADDs) verwalten, die im Unterschied zu FBDDs neben der 0-Senke und der 1-Senke auch Senken mit beliebigen anderen ganzen Zahlen als Bezeichnern zulassen, und diese Senken zum Speichern der Informationen benutzen, aber nach der Berechnung von G_{m+1}^C ist eine Minimierung in einer Laufzeit von $O(|G_{m+1}^C|)$ erforderlich. Dieser Ansatz ist somit zumindest asymptotisch genauso schnell wie die *bottom-up* Berechnung, verbraucht aber, weil nicht minimierte Zwischenergebnisse materialisiert werden müssen, im schlechtesten Fall exponentiell mehr Speicherplatz. Die Implementation der ADD-basierten Konstruktion verbrauchte darüber hinaus gegenüber der *bottom-up* Konstruktion bis zu 50% mehr Zeit und wurde daher nicht in die endgültige Version des Kryptanalysepakets übernommen.

Die Überlegungen bezüglich des Steuerungsgraphen G_m^C lassen sich mit Hilfe der Ausführungen in Kapitel 7.3 unmittelbar auf den G_m^C -FBDD Q_m übertragen. Wir verzichten daher auf eine detaillierte Pseudoco-

dedarstellung des Konstruktionsalgorithmus $keystreamFbdd(m)$.

9.2 Berechnung des G_m^C -FBDDs S_m

Der G_m^C -FBDD R_m , der für einen gegebenen internen Bitstrom z entscheidet, ob z durch den linearen Bitstromgenerator des A5/1-Algorithmus erzeugt worden ist, hat nur theoretische Bedeutung, so dass wir uns auf die Implementation des G_m^C -FBDDs S_m beschränken können.

Die Struktur von S_m hatten wir in Kapitel 7.4 bereits hergeleitet. Der Berechnungsalgorithmus 10 mit einer Laufzeit und einem Speicherplatzbedarf von $O(|S_m|)$ folgt unmittelbar aus diesen Überlegungen, allerdings benutzen wir zur Berechnung des Kontrollwerts b_{m-1} die LFSR-Rekurrenzdarstellung aus Beobachtung 2 anstelle der Darstellung in Beobachtung 3, die die Verwaltung der Koeffizienten $L_{j,i}$ erfordern würde.

Ähnlich wie bei der Konstruktion von G_m^C codieren wir aus Speicherplatzgründen alle Parameter in einen einzigen `unsigned int`. Auch für S_m gilt, dass die Berechnungsergebnisse für die Konstruktion von S_{m+1} nicht wiederverwendet werden können, so dass wir auch hier einen lokalen *Cache* anstatt der globalen *computed table* verwenden.

9.3 Implementation des Kryptanalysealgorithmus

Nachdem wir nun in der Lage sind, den Steuerungsgraphen und die benötigten FBDDs effizient zu berechnen, kann der generische Kryptanalysealgorithmus aus Kapitel 5 (vgl. Algorithmus 4) für den A5/1-Schlüsselstromgenerator konkretisiert werden.

Die zusätzliche *Shift*-Bedingung für den A5/1-Schlüsselstromgenerator macht eine Verknüpfung von S_m schon für $m < n'$ erforderlich. Da S_m so implementiert ist, dass direkt die 1-Senke zurückgegeben wird, falls für m keine Konsistenzprüfung durchgeführt werden muss, starten wir der Einfachheit halber die *for*-Schleife bei $m = 1$.

Wir hatten im Kapitel 5 beobachtet, dass auf bestimmten Pfaden in G_m^C unter Umständen nicht alle Bits des internen Bitstroms $z \in \{0, 1\}^m$ eingelesen werden. Dies gilt insbesondere für den irregulär getakteten A5/1-Schlüsselstromgenerator, denn es ist möglich (allerdings nicht sehr wahrscheinlich), dass immer nur die ersten beiden LFSR getaktet werden und damit nie ein anderes als das erste Ausgabebit z_0^2 des dritten LFSR in die Berechnung der Schlüsselbits geht. Allerdings kann es sein, dass der Übergang zu einem internen Bitstrom der Länge $m + i$, $i \in N$, dafür sorgt, dass das dritte LFSR schließlich getaktet und damit das interne Bit z_1^2 eingelesen wird. Somit kommt z_1^2 auf dem entsprechenden Pfad in G_m^C nicht vor, wird in der Fortsetzung des Pfades in G_{m+1}^C jedoch abgefragt. Wenn wir nun zum Beispiel den FBDD S_8 , der entscheidet, ob das interne Bit z_1^2 durch den linearen Bitstromgenerator des A5/1-Schlüsselstromgenerators erzeugt worden ist, auf Basis des Steuerungsgraphen G_m^C konstruieren, wird das Bit auf dem oben beschriebenen Pfad in G_m^C nicht eingelesen und damit die Konsistenzbedingung für dieses Bit nicht geprüft. Um die Konsistenzbedingung auch auf diesem Pfad zu testen, müssen wir also den Steuerungsgraphen G_{m+i}^C für die Berechnung von S_8 verwenden.

Für unseren Algorithmus bedeutet dieser Umstand, dass wir alle FBDDs S_m aus dem größten betrachteten Steuerungsgraphen, d.h. aus $G_{\lceil \alpha^{-1}n \rceil}^C$, konstruieren und damit auch die Syntheseoperationen basierend auf diesem Steuerungsgraphen durchführen müssen. Die in Kapitel 5 angestellten Überlegungen zur asymptotischen Laufzeit unserer Kryptanalyse bleiben aufgrund der FBDD-Annahme jedoch gültig, so

dass wir aus Gründen der Übersichtlichkeit auf eine Integration dieser Anpassung des Algorithmus in die theoretische Darstellung in Kapitel 5 verzichtet haben.

Weiterhin stellt die Implementation des FBDD-Pakets sicher, dass die Minimierungsoperation bereits in die Syntheseoperation integriert ist, so dass wir auf eine explizite Minimierung des Resultats verzichten können. Insgesamt erhalten wir damit den in Algorithmus 11 dargestellten, an den A5/1 Schlüsselstrom-generator angepassten Kryptanalysealgorithmus.

Um die globale *computed table*, die in der CUDD-Implementation nur drei Operanden zulässt, verwenden zu können, ist die tenäre Synthese im ersten Ansatz als verkettete binäre Synthese implementiert worden.

Erste Tests haben ergeben, dass es sinnvoll ist, nicht in jeder Iteration den FBDD Q_m zum Zwischenergebnis P_m zu multiplizieren. Die P_m werden dadurch zwar größer, aber es lassen sich durch Einsparen der zeitintensiven Syntheseoperation Geschwindigkeitsvorteile realisieren, die die Gesamtlaufzeit des Algorithmus gegenüber der herkömmlichen Variante verkürzen.

Die Konstruktion der FBDDs ist in der Datei `attackA5-1fbdds.c` implementiert, während sich die Umsetzung des Kryptanalysealgorithmus in der Datei `genericAttack.c` befindet. Für die Verarbeitung von linearen Bitstromgeneratoren stehen weiterhin die Datei `lfsrGenerator.c` sowie die auf [BGW99] basierende Datei `A5-1Simulator.c` speziell für den A5/1 Generator zur Verfügung.

Mit Hilfe des Programms `attackA5-1` kann ein einzelner Angriff auf den A5/1 Generator simuliert werden. Zudem besteht die Möglichkeit, mit `attackEval` Angriffsserien auf zufällig zusammengestellte A5/1 Generatoren durchzuführen. Dieses Programm wurde insbesondere für die Datengewinnung im Rahmen der empirischen Analyse verwendet. Die Verwendung von `attackA5-1` und `attackEval` ist auf der dieser Arbeit beiliegenden CD-ROM beschrieben.

Algorithmus 9 oracle(m)

```

// compute the oracle graph  $G_m^C$  for an internal bitstream of length  $m$ 
return outputRecur([0, 0, 0], NIL, NIL)

```

```

function outputRecur( $i, u, v$ )
//  $i[0..2]$  =current read positions,  $u$  =unchanged index  $\in \{0, 1, 2, NIL\}$ 
//  $v$  =unchanged control value  $\in \{0, 1, NIL\}$ 
if  $\exists r \in \{0, 1, 2\} \setminus \{u\} : 6 \cdot i[r] + r \geq m$  then
    return * - sink
result  $\leftarrow$  localCache.lookup( $i, u, v$ )
if result  $\neq$  NIL then // result is already known
    return result
Read  $\leftarrow \{0, 1, 2\} \setminus \{u\}$ 
Let  $r_0, \dots, r_{|Read|-1}$  the elements of Read in ascending order, i.e.  $r_m < r_n$  for  $m < n$ 
 $r \leftarrow$  controlRecur( $i, u, v$ )
if  $u = NIL$  then
    maxLevel  $\leftarrow$  2
else
    maxLevel  $\leftarrow$  1
for level = maxLevel downto 0 do
     $s =$  unique - table.get( $z_{i[r_{level}]}^{level}, r, r$ )
     $r \leftarrow s$ 
localCache.insert( $i, u, v, r$ )
return  $r$ 

```

```

function controlRecur( $i, u, v$ )
if  $\exists r \in \{0, 1, 2\} \setminus \{u\} : 6 \cdot i[r] + 3 + r \geq m$  then
    return * - sink
     $Read \leftarrow \{0, 1, 2\} \setminus \{u\}$ 
    Let  $r_0, \dots, r_{|Read|-1}$  the elements of  $Read$  in ascending order, i.e.  $r_m < r_n$  for  $m < n$ 
if  $u \neq NIL$  then
     $c[u] \leftarrow v$ 
for  $p = 0$  to 7 do // compute  $t_0, \dots, t_7$ 
    Let  $(p_2, p_1, p_0)$  the binary representation of  $p$ , i.e.  $p = \sum_{j=0}^2 p_j 2^j$ 
     $c[r_0] \leftarrow p_2$ 
     $c[r_1] \leftarrow p_1$ 
if  $u = NIL$  then
     $c[r_2] \leftarrow p_0$ 
else if  $p_0 \neq v$  then
    continue //  $t[p]$  is not needed
     $controlValue \leftarrow maj_3(c[0], c[1], c[2])$ 
if  $\exists r \in \{0, 1, 2\} : c[r] \neq controlValue$  then
    // By definition of  $maj_3$ ,  $\exists$  at most one such  $r$ 
     $newU \leftarrow r$ 
     $newV \leftarrow controlValue \oplus 1$  // set unchanged control value
else
     $newU \leftarrow NIL$ 
     $newV \leftarrow NIL$ 
for  $l = 0, 1, 2$  do
     $\Delta i[l] \leftarrow \begin{cases} 0 & : l = newU \\ 1 & : l \neq newU \end{cases}$ 
     $t[p] \leftarrow outputRecur(i + \Delta i, newU, newV)$ 
for  $p = 0$  to 7 step 2 do // compute  $t_{01}, \dots, t_{67}$ 
if  $u = NIL$  then
     $t[p] \leftarrow unique - table.get(z_{i[r_2]}^{3+r_2}, t[p+1], t[p])$ 
else
     $t[p] \leftarrow t[p+v]$ 
for  $p = 0$  to 7 step 4 do // compute  $t_{03}, \dots, t_{47}$ 
     $t[p] \leftarrow unique - table.get(z_{i[r_1]}^{3+r_1}, t[p+2], t[p])$ 
 $t[0] \leftarrow unique - table.get(z_{i[r_0]}^{3+k_0}, t[4], t[0])$ 
return  $t[0]$ 

```

Algorithmus 10 $linearityFbdd(u, m)$

```

// compute the linearity-check FBDD  $S_m$ 
//  $u$  = root of the  $G_m^C$ -subgraph
//  $m$  = position in the internal bitstream to check
 $r_m = (m - 1) \bmod 6$ 
 $s_m = (m - 1) \text{ div } 6$ 
 $checkShift \leftarrow r_m \leq 2$  AND  $s_m \geq N^{r_m}$ 
 $checkLinearity \leftarrow (r_m > 2$  AND  $s_m \geq n^{r_m})$  OR  $(r_m \leq 2$  AND  $s_m \in \{n^{r_m}, \dots, n^{r_m} + N^{r_m} - 1\})$ 
if ( $!checkShift$  AND  $!checkLinearity$ ) then
    return 1 - sink
return  $linearityFbddRecur(u, 0, 0)$ 

```

```

function linearityFbddRecur( $u, b, c$ )
  //  $b$  = current value of  $b_{m-1}$ 
  //  $c$  = comparison value  $z_{s_m - N^{r_m}}^{3+r_m}$  for  $r_m \in \{0, 1, 2\}$ 
  if  $u = * - \text{sink}$  then
    return  $1 - \text{sink}$ 
  if  $u.\text{label} = m - 1$  then
    if  $\text{checkShift}$  AND  $\text{checkLinearity}$  AND  $b \neq c$  then
      return  $0 - \text{sink}$ 
    if ( $\text{checkShift}$  AND  $c = 0$ ) OR ( $\text{checkLinearity}$  AND  $b = 0$ ) then
      return  $\text{unique} - \text{table.get}(u.\text{label}, 0 - \text{sink}, 1 - \text{sink})$ 
    else
      return  $\text{unique} - \text{table.get}(u.\text{label}, 1 - \text{sink}, 0 - \text{sink})$ 
   $\text{result} \leftarrow \text{localCache.lookup}(u, b, c)$ 
  if  $\text{result} \neq \text{NIL}$  then
    return  $\text{result}$ 
   $r \leftarrow u.\text{label.index} \text{ div } 6$ 
   $s \leftarrow u.\text{label.index} \text{ mod } 6$ 
  for  $i \in \{0, 1\}$  do
     $\text{newC}[i] \leftarrow c$ 
     $\text{newB}[i] \leftarrow b$ 
  if  $\text{checkShift}$  AND  $r = r_m + 3$  AND  $s = s_m - N^r$  then
     $\text{newC}[0] \leftarrow 0$ 
     $\text{newC}[1] \leftarrow 1$ 
  if  $\text{checkLinearity}$  AND  $r = r_m$  AND  $s_m - s \leq n^{r_m}$  then
     $\text{newB}[0] \leftarrow b \oplus c_{s_m - s - 1}^r \cdot 0$ 
     $\text{newB}[1] \leftarrow b \oplus c_{s_m - s - 1}^r \cdot 1$ 
   $\text{thenBranch} \leftarrow \text{linearity} - \text{fbdd}(u_1, m, \text{newB}[1], \text{newC}[1])$ 
   $\text{elseBranch} \leftarrow \text{linearity} - \text{fbdd}(u_0, m, \text{newB}[0], \text{newC}[0])$ 
   $\text{result} \leftarrow \text{unique} - \text{table.get}(u.\text{label}, \text{thenBranch}, \text{elseBranch})$ 
   $\text{cache.put}(u, b, c, \text{result})$ 
  return  $\text{result}$ 

```

Algorithmus 11 FBDD-COMPUTE-A5/1-x

```

 $G_{\lceil \alpha^{-1} n \rceil}^C \leftarrow \text{oracle}(\lceil \alpha^{-1} n \rceil)$ 
 $P \leftarrow 1 - \text{sink}$ 
for  $m \leftarrow 1$  to  $\lceil \alpha^{-1} n \rceil$  do
   $P \leftarrow \text{SYNTH}(G_{\lceil \alpha^{-1} n \rceil}^C, P, \text{keystreamFbdd}(m), \text{linearityFbdd}(G_{\lceil \alpha^{-1} n \rceil}^C.\text{root}, m), \wedge)$ 
return  $i_m(L, z^*)$  for a  $z^* \in \text{One}(P)$ 

```

Kapitel 10

Empirische Resultate

Mit der erstellten Software sind wir nun in der Lage, auf kleineren Instanzen des A5/1 Schlüsselstromgenerators Angriffe durchzuführen und den dargestellten theoretischen Resultaten das Verhalten des Algorithmus in der Praxis gegenüberzustellen.

10.1 Entwicklung der maximalen Größe von P_m in Abhängigkeit der Schlüssellänge

Im Kapitel 5 haben wir die maximale Größe der Zwischenresultate P_m , $m \in \{n', \dots, \lceil \alpha^{-1}n \rceil\}$, als zentralen Einflussfaktor der Laufzeit unseres Kryptanalysealgorithmus identifiziert. Wir wollen daher für den konkreten Fall des A5/1 Schlüsselstromgenerators nun feststellen, wie stark die maximale Größe von P_m von der in Kapitel 7 berechneten theoretischen Schranke

$$|P_m| \in O(n^{10}) \cdot 2^{0,6403n}$$

in der Praxis abweicht.

10.1.1 Datengenerierung

Wir betrachten für unsere Untersuchung A5/1 Schlüsselstromgeneratoren mit Schlüssellängen zwischen 13 und 28 Bit, deren LFSR zwischen 3 und 14 Bit lang sind. Genau wie beim realen A5/1 Generator sind die charakteristischen Polynome der LFSR stets primitiv, wobei die primitiven Polynome der unter [Cha] verfügbaren Liste entnommen wurden.

In Anlehnung an [Sch02] unterscheiden wir zwischen Generatoren, deren LFSR spärlich bzw. reichlich besetzte charakteristische Polynome besitzen. Dabei fassen wir ein charakteristisches Polynom als spärlich besetzt auf, wenn weniger als die Hälfte der Koeffizienten gleich 1 ist, andernfalls sei das Polynom reichlich besetzt. Für kleine n existieren nur wenige primitive Polynome vom Grad n , so dass wir in diesen Fällen alle primitiven Polynome vom Grad n sowohl als spärlich als auch als reichlich besetzt betrachten wollen.

Für jede Schlüssellänge $n \in \{13, \dots, 28\}$ erzeugen wir zufällig 4 Generatoren für $n \leq 16$ und 10 Generatoren für $n > 16$. Wir wählen dazu die Längen n_0, n_1 der ersten beiden LFSR zufällig aus $[\frac{n}{3} - 3, \frac{n}{3} + 3]$

und die Länge n_2 des dritten LFSR gemäß der verbleibenden Schlüssellänge $n - n_1 - n_2$, um für jede Schlüssellänge unterschiedliche lineare Bitstromgeneratoren in der Analyse zu berücksichtigen.

Auf jeden dieser Generatoren werden $p(n)$ Angriffe durchgeführt mit $p(n) = 30$ für $n \leq 22$, $p(n) = 10$ für $n \in \{23, \dots, 26\}$ und $p(n) = 5$ für $n \in \{27, 28\}$. Zu Beginn jedes Angriffs erzeugen wir zufällig einen Initialzustand $x \in \{0, 1\}^n$ mit der Eigenschaft, dass die Initialzustände der drei LFSR jeweils von Null verschieden sind. Aus diesem Initialzustand lassen wir den Generator genügend viele, d.h. mindestens $\lceil \gamma \alpha^{-1} n \rceil$ viele Schlüsselbits produzieren. Mit diesen Schlüsselbits als beobachtetem Schlüsselstrom im Sinne des in Kapitel 2 dargestellten Angriffsmodells starten wir den Kryptanalysealgorithmus und messen die maximale Größe von P_m , die während der Berechnung als Zwischenergebnis auftritt.

Der Algorithmus läuft zunächst nur bis zur theoretischen Eindeutigkeitsgrenze, d.h. über $\lceil \alpha^{-1} n \rceil$ Iterationen. Nach wie vielen Iterationen nur noch Initialzustände übrigbleiben, die mit dem beobachteten Schlüsselstrom vollständig konsistent sind, wäre ein möglicher Gegenstand weiterer Analysen.

10.1.2 Ergebnisse

Wir gehen zunächst zu einer logarithmierten Darstellung der interessierenden Größen über, um das exponentielle Wachstum leichter einschätzen zu können. Für den Logarithmus $s(n)$ der theoretischen Schranke aus Beobachtung 69 ergibt sich

$$\begin{aligned} s(n) &= \log(c \cdot n^{10} \cdot 2^{0,6403n}) \\ &= \log(c) + 10 \cdot \log(n) + 0,6403n \end{aligned}$$

Die Diagramme 10.1 und 10.2, in die $s(n)$ (unter Vernachlässigung der Konstanten c) als theoretischer Wert eingetragen ist, zeigen die Resultate dieses Experiments für spärlich bzw. reichlich besetzte charakteristische Polynome.

Die empirischen Ergebnisse folgen der Tendenz der theoretischen Schranke, denn die Steigungen liegen mit etwa 0,66 bzw. 0,74 im Bereich der Steigung des linearen Anteils der logarithmierten theoretischen Schranke. Mit jedem zusätzlichen Schlüsselbit wächst $|P_m|$ somit um einen Faktor von $2^{0,66} = 1,58$ bzw. $2^{0,74} = 1,67$. Weiterhin liegen die Kurven für die maximalen und die minimalen Werte ähnlich wie im von [Sch02] untersuchten OBDD-Fall relativ nahe beieinander.

Auffällig ist allerdings in beiden Fällen der große Abstand der theoretischen Schranke von den empirischen Ergebnissen. Dies könnte auf die sehr pessimistische Annahme von $O(n^{10})$ für den polynomiellen Faktor in der theoretischen Schranke zurückzuführen sein. Insbesondere weil der G_m^C -FBDD Q_m weitgehend dem Steuerungsgraphen entspricht, ist es unwahrscheinlich, dass die *worst-case* Größe des Syntheseresultats tatsächlich angenommen wird. Auf diese Fragestellung könnte im theoretischen Umfeld, aber auch in weiteren empirischen Untersuchungen, etwa durch Betrachtung der Größen von P_m in Abhängigkeit der Syntheseparameter, näher eingegangen werden. Auch empirische Ergebnisse für größere Schlüssellängen wären an dieser Stelle hilfreich.

10.2 $|P_m|$ in Abhängigkeit der Iterationen

Wir wollen zusätzlich überprüfen, ob die Größen von P_m in den einzelnen Iterationen den in der Theorie beobachteten Verlauf besitzen, d.h. ob $|P_m|$ zunächst klein ist, bis zu seinem Maximalwert ansteigt und

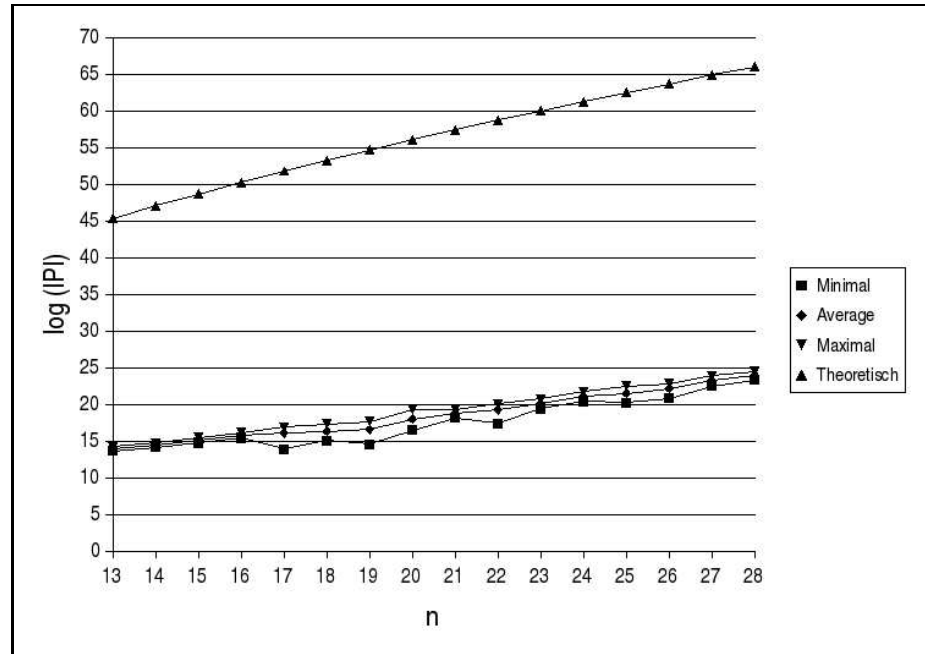


Abbildung 10.1: Maximale Größe des FBDDs P_m in Abhängigkeit der Schlüssellänge n für spärlich besetzte charakteristische Polynome

anschließend wieder kleiner wird, und wie stark sich das Verhalten der einzelnen Generatoren unterscheidet.

10.2.1 Datengenerierung

Wir betrachten 10 verschiedene, zufällig gewählte Generatoren der Schlüssellänge 20 und führen für jeden dieser Generatoren 10 Angriffe mit jeweils zufällig gewählten, für alle LFSR von Null verschiedenen Initialzuständen durch. Auch hier unterscheiden wir wiederum zwischen spärlich und reichlich besetzten charakteristischen Polynomen.

Nun lassen wir den Algorithmus über 120 Iterationen und damit deutlich über die Eindeutigkeitsgrenze

$$m^* = \lceil \alpha^{-1} n \rceil = \lceil (0, 2193)^{-1} \cdot 20 \rceil = 92$$

hinaus laufen.

10.2.2 Ergebnisse

Anhand der Diagramme 10.3 und 10.4 erkennt man, dass der in der Theorie abgeleitete Verlauf auch in der Praxis eintritt, und dass beim Erreichen der Eindeutigkeitsgrenze $m^* = 92$ die Größe von P_m von ihrem Maximalwert schon wieder sehr weit entfernt ist. Auffällig ist, dass $|P_m|$ für reichlich besetzte Polynome schneller zu sinken scheint als für spärlich besetzte. Obwohl die Kurve der durchschnittlichen Werte in beiden Fällen einen ähnlichen Verlauf besitzt, ist analog zu den Beobachtungen in [Sch02] auch

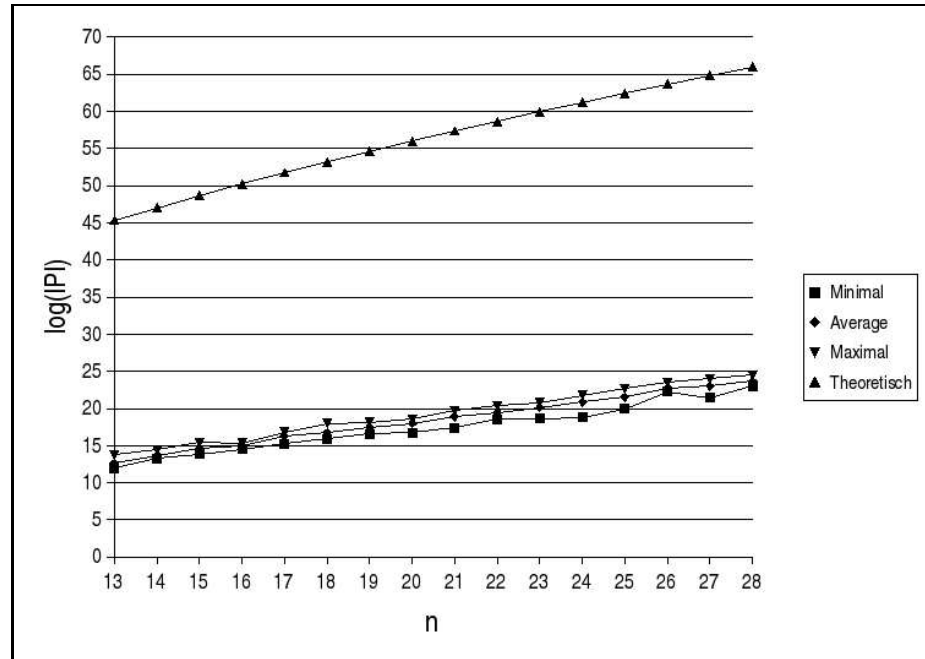


Abbildung 10.2: maximale Größe des FBDDs P_m in Abhängigkeit der Schlüssellänge n für reichlich besetzte charakteristische Polynome

für den A5/1 Schlüsselstromgenerator der Abstand zwischen der Maximal- und der Minimalkurve für spärliche besetzte Polynome deutlich größer als für reichlich besetzte.

10.3 Verwendete Hardware und Systemumgebung

Abschliessend soll kurz auf die Systemressourcen eingegangen werden, die für die Implementation und die Experimente zur Verfügung standen.

Alle Experimente wurden auf einem PC mit den folgenden Kernkomponenten durchgeführt:

- Intel Pentium 4 Prozessor mit 3,4 GHz Taktfrequenz und 512 kB Cache
- Intel D865PERLX Motherboard mit FSB 800
- 1 GB Hauptspeicher (2 · 512 MB PC-400 DDR-SDRAM Module)

Übersetzt wurde die erstellte Software unter Linux mit dem *gcc-Compiler* in der Version 3.3.3.

Wie durch die theoretische Laufzeit- und insbesondere Speicherplatzschränke schon angedeutet, erweist sich in der Praxis eindeutig der zur Verfügung stehende Arbeitsspeicher als wichtigster Engpassfaktor. Angriffe auf Generatoren mit Schlüssellängen bis zu 28 benötigten auf dem verwendeten System höchstens 12 – 15 Minuten, ein effizienter Angriff auf Generatoren der Länge 29 scheiterte allerdings am zu kleinen Arbeitsspeicher. Dies legt die Vermutung nahe, dass mit größerem Arbeitsspeicher auch Generatoren mit Längen größer als 30 mit der entwickelten Implementation effizient angegriffen werden könnten.

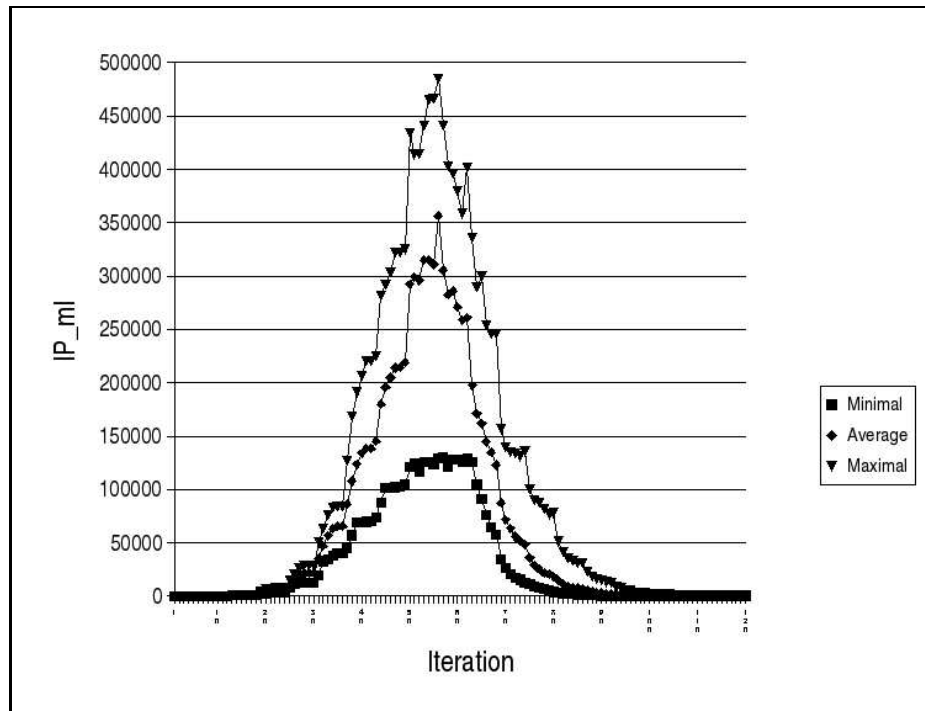


Abbildung 10.3: Größe des FBDDs P_m in Abhängigkeit der Iteration m für spärlich besetzte charakteristische Polynome

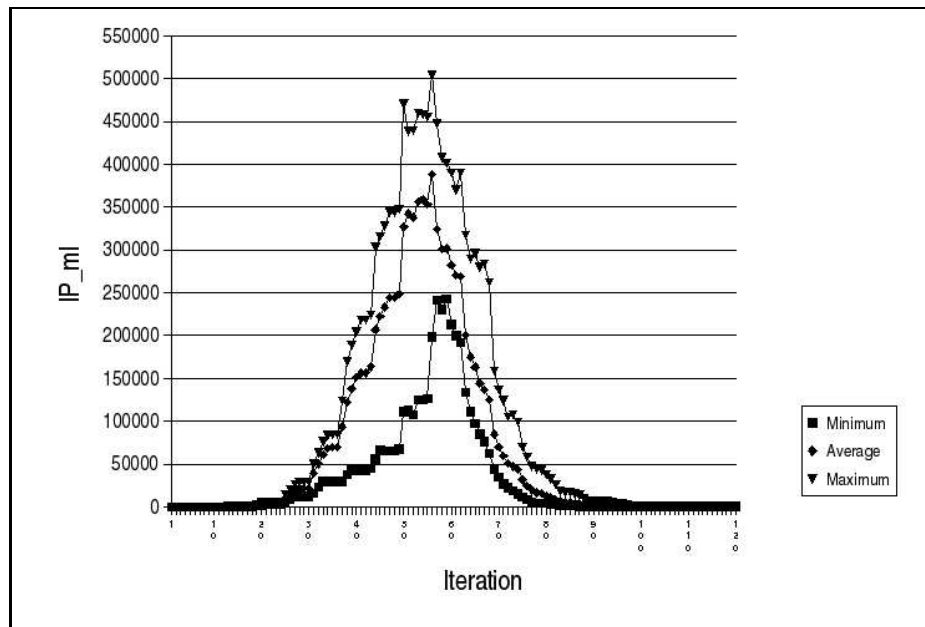


Abbildung 10.4: Größe des FBDDs P_m in Abhängigkeit der Iteration m für reichlich besetzte charakteristische Polynome

Kapitel 11

Zusammenfassung und Ausblick

Nach der theoretischen Betrachtung LFSR-basierter Schlüsselstromgeneratoren und der FBDD-basierten Kryptanalyse dieser Generatoren ist im zweiten Teil am Beispiel des A5/1 Schlüsselstromgenerators ein System entwickelt worden, das die dargestellte Kryptanalyse auf hinreichend kleinen Instanzen sehr schnell durchführt und damit die theoretische obere Schranke deutlich unterbietet. Dies liegt vermutlich zu einem wesentlichen Teil an der Überschätzung des polynomiellen Faktors in der theoretischen Laufzeitschranke, aber dennoch wirken sich Details wie komplementierte Kanten in der Implementation positiv aus. Um in der vorliegenden Anwendung den von komplementierten Kanten ausgehenden Effekt quantifizieren zu können, wäre eine Alternativimplementation des Angriffs ohne komplementierte Kanten von Vorteil. Dies ist mit dem CUDD-Paket eventuell durch Übergang zu *Algebraic Decision Diagrams* möglich.

Im Rahmen der Implementation ist mangels anderer verfügbarer Pakete als Nebenprodukt ein generisches FBDD-Paket entstanden, das mit herkömmlichen und in großer Zahl verfügbaren OBDD-Paketen kombiniert werden kann. Auf diese Weise konnten die effizienten Algorithmen des OBDD-Pakets CUDD für die Implementation der Kryptanalyse ausgenutzt werden, ohne sämtliche Basisdatenstrukturen neu implementieren zu müssen. Allerdings sind auch die in OBDD-Paketen enthaltenen Algorithmen, sobald die Probleminstanzen eine gewisse Größe erreichen, limitiert durch die Berechnungskraft der Maschinen, auf denen sie ausgeführt werden. Im vorliegenden Fall konnte nicht ausreichender Arbeitsspeicher algorithmisch nicht effizient ausgeglichen werden, so dass die Angriffe bei weniger als der halben Schlüssellänge des originalen A5/1 Schlüsselstromgenerators abgebrochen werden mussten. Die zu Beginn der Arbeit aufgeworfene Frage, ob der betrachtete Angriff aus praktischer Sicht eine ernstzunehmende Gefahr darstelle, ist damit in eindeutiger Weise beantwortet.

Andererseits empfiehlt die hohe Speicherplatzintensität den implementierten Kryptanalysealgorithmus für das *Benchmarking* verschiedener OBDD-Pakete, die durch die generische Schnittstelle leicht an das System anzubinden wären. Ebenso könnten auch andere Chiffren mit geringen Aufwand durch das implementierte System analysiert werden.

Literaturverzeichnis

- [BGW99] BRICENO, M., I. GOLDBERG und D. WAGNER: *A pedagogical implementation of A5/1*, Mai 1999. <http://jya.com/a51-pi.htm>.
- [Bie98] BIERE, A.: *ABCD: An experimental BDD library*. Eidgenössische Technische Hochschule Zürich, Schweiz, 1998.
<http://i10www.ira.uka.de/biere/abcd/index.html>.
- [BRB90] BRACE, K., R. RUDELL und R. BRYANT: *Efficient Implementation of a BDD Package*. In: *Conference proceedings on 27th ACM/IEEE design automation conference, DAC '90*, Seiten 40–45. ACM Press, New York, NY, USA, 1990.
- [Bri99] BRICENO, M.: *A5/1 FAQ*, Mai 1999. <http://jya.com/a51-pi.htm>.
- [Bry86] BRYANT, R.: *Graph-Based Algorithms for Boolean Function Manipulation*. IEEE Transactions on Computers, C-35(8):677–691, August 1986.
- [BSW01] BIRYUKOV, A., A. SHAMIR und D. WAGNER: *Real Time Cryptanalysis of A5/1 on a PC*. In: *Proceedings of Fast Software Encryption: 7th International Workshop, FSE 2000*, Band 1978 der Reihe *Lecture Notes in Computer Science*, Seiten 1–18. Springer Verlag, Heidelberg, 2001.
- [Cha] CHABAUD, F.: *Primitive Polynomials over GF(2)*.
URL: <http://fchabaud.free.fr/English/default.php>.
- [CLRS01] CORMEN, T., C. LEISERSON, R. RIVEST und C. STEIN: *Introduction to Algorithms*. The MIT Press, Cambridge, MA, Zweite Auflage, 2001.
- [CMT93] COUDERT, O., J. MADRE und H. TOUATI: *TiGeR Version 1.0 User Guide*. Digital Paris Research Lab, December 1993.
- [GHJV96] GAMMA, E., R. HELM, R. JOHNSON und J. VLISSIDES: *Entwurfsmuster: Elemente wiederverwendbarer objektorientierter Software*. Addison-Wesley-Longman, Bonn, 1. Auflage, 1996.
- [Gol82] GOLOMB, S. W.: *Shift Register Sequences*. Aegean Park Press, Laguna Hills, California, USA, 1982.
- [Jan98] JANSSEN, G.: *The Eindhoven BDD Package*. University of Eindhoven, 1998.
<ftp://ftp.ics.ele.tue.nl/pub/users/geert/bdd.tar.gz>.
- [Jan01] JANSSEN, G.: *Design of a Pointerless BDD Package*. In: *IEEE 10th International Workshop on Logic & Synthesis, IWLS 2001*, 2001.
- [Kra02] KRAUSE, M.: *BDD-Based Cryptanalysis of Keystream Generators*. In: *EUROCRYPT 2002*, Band 2332 der Reihe *Lecture Notes in Computer Science*, Seiten 222–237. Springer Verlag, Heidelberg, 2002.

- [LN02] LIND-NIELSEN, J.: *BuDDy - A Decision Diagram Package*. IT University of Copenhagen, Denmark, 2002. <http://www.itu.dk/research/buddy/index.html>.
- [Lon93] LONG, D.: *The BDD Library*. Carnegie Mellon University, Pittsburgh, PA, USA, 1993. <http://www-2.cs.cmu.edu/afs/cs/project/modck/pub/www/bdd.html>.
- [Lon98] LONG, D. E.: *The Design of a Cache-Friendly BDD Library*. In: *Proceedings of the 1998 IEEE/ACM international conference on Computer-aided design, ICCAD 1998*, Seiten 639–645. ACM Press, New York, NY, USA, 1998.
- [Mas76] MASEK, W.: *A fast algorithm for the string editing problem and decision graph complexity*. Diplomarbeit, MIT, Cambridge, MA, USA, 1976.
- [MT98] MEINEL, C. und T. THEOBALD: *Algorithmen und Datenstrukturen im VLSI-Design: OBDD - Grundlagen und Anwendungen*. Springer-Verlag, Berlin, Heidelberg, New York, 1998.
- [RS97] RANJAN, R. und J. SANGHAVI: *CAL-2.0: Breadth-first manipulation based BDD-library*. University of California, Berkeley, CA, USA, Juni 1997. <http://www-cad.eecs.berkeley.edu/Respep/Research/bdd/calbdd/>.
- [Rue86] RUEPPEL, R. A.: *Analysis and Design of Stream Ciphers*. Springer-Verlag, Berlin, Heidelberg, 1986.
- [Sch02] SCHLEER, F.: *Einsatz von OBDDs zur Kryptanalyse von Flusschiffren*. Diplomarbeit, Universität Mannheim, November 2002.
- [Sen96] SENTOVICH, E. M.: *A Brief Study of BDD Package Performance*. In: *Formal Methods in Computer-Aided Design: First International Conference, FMCAD '96*, Band 1166 der Reihe *Lecture Notes in Computer Science*, Seiten 389–403. Springer Verlag, Heidelberg, 1996.
- [Sha49] SHANNON, C. E.: *Communication Theory of Secrecy Systems*. Bell System Technical Journal, 28-4:656–715, 1949.
- [SM93] SLOBODOVÁ, A. und C. MEINEL: *Efficient Manipulation with FBDDs by Means of a Modified OBDD-package*. Technischer Bericht 93-09, Universität Trier, April 1993.
- [Som01] SOMENZI, F.: *CUDD: CU decision diagram package*. University of Colorado, Boulder, CO, USA, März 2001. <http://vlsi.colorado.edu/~fabio/>.
- [Ste03] STEGEMANN, D.: *Free Binary Decision Diagrams als Datenstruktur für Boolesche Funktionen*. Seminararbeit, Universität Mannheim, Oktober 2003.
- [SW95] SIELING, D. und I. WEGENER: *Graph driven BDDs - a new data structure for Boolean functions*. Theoretical Computer Science, 141:283–310, 1995.
- [Uni04] UNIVERSITY OF TRIER, DIVISION FOR COMPLEXITY AND VLSI DESIGN: *BDD-Portal*, 2004. <http://www.bdd-portal.org>.
- [Weg00] WEGENER, I.: *Branching Programs and Binary Decision Diagrams: Theory and Applications*. SIAM Monographs on Discrete Mathematics and Applications, 2000.
- [YBO⁺98] YANG, B., R. BRYANT, D. O'HALLARON, A. BIÈRE, O. COUDERT, G. JANSSEN, R. RANJAN und F. SOMENZI: *A Performance Study of BDD-Based Model Checking*. In: *Formal Methods in Computer-Aided Design: Second International Conference, FMCAD '98*, Band 1522 der Reihe *Lecture Notes in Computer Science*, Seiten 255–289. Springer Verlag, Heidelberg, 1998.
- [YCBO98] YANG, B., Y.-A. CHEN, R. BRYANT und D. O'HALLARON: *Space- and Time-Efficient BDD Construction via Working Set Control*. In: *Proceedings of the Asian-Pacific Design Automation Conference ASPDAC 98*, Seiten 423–432, Februar 1998.

Anhang A

Änderungen am Quellcode von CUDD

Die folgende Funktion in der Quellcodedatei `cuddLCache.c` ist wegen Speicherzugriffsfehlern in der Originalversion in folgender Weise überarbeitet worden:

```
void cuddLocalCacheClearDead(DdManager * manager){
    DdLocalCache *cache = manager->localCaches;
    unsigned int keysize;
    unsigned int itemsize;
    unsigned int slots;
    DdLocalCacheItem *item;
    DdNodePtr *key;
    unsigned int i, j;

    while (cache != NULL) {
        keysize = cache->keysize;
        itemsize = cache->itemsize;
        slots = cache->slots;
        item = cache->item;
        for (i = 0; i < slots; i++) {
            if (item->value!=NULL){
                if (Cudd_Regular(item->value)->ref == 0) {
                    item->value = NULL;
                } else {
                    key = item->key;
                    for (j = 0; j < keysize; j++) {
                        if (Cudd_Regular(key[j])->ref == 0) {
                            item->value = NULL;
                            break;
                        } /* if */
                    } /* for j */
                } /* else */
            } /* if */
            item = (DdLocalCacheItem *) ((char *) item + itemsize);
        }
    }
}
```

```
    } /* for i */  
    cache = cache->next;  
  } /* while */  
} /* end of cuddLocalCacheClearDead */
```

Anhang B

Ehrenwörtliche Erklärung

Ich versichere, dass ich die beiliegende Diplomarbeit ohne die Hilfe Dritter und ohne Benutzung anderer als der angegebenen Quellen und Hilfsmittel angefertigt und die den benutzten Quellen wörtlich oder inhaltlich entnommenen Stellen als solche kenntlich gemacht habe. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Ich bin mir bewusst, dass eine falsche Erklärung rechtliche Folgen haben wird.

Mannheim, den 30.06.2004