

# **Pseudozufallszahlengeneratoren in LiSA**

Diplomarbeit von

**Bärbel Müller**

nach einem **Thema von Prof. Dr. Johannes Buchmann**

Universität des Saarlandes  
Fachbereich 14 Informatik

10. Juli 1996

Hiermit versichere ich an Eides Statt, daß ich bei der Anfertigung dieser Arbeit keine anderen als die angegebenen Quellen benutzt habe.

Saarbrücken, den 10. Juli 1996

Für das interessante Thema und die Betreuung möchte ich mich an dieser Stelle bei Herrn Prof. Dr. Johannes Buchmann bedanken. Sehr dankbar bin ich auch Dr. Ingrid Biehl, Dr. Bernd Meyer, Dipl.-Inf. Diethelm Schlegel und Dr. Christoph Thiel für die hilfreichen Diskussionen und das Korrekturlesen der Arbeit. Mein Freund Ralf Roth und meine Familie, insbesondere meine Schwester Andrea Johanna Magarete Müller haben mich stets tatkräftig unterstützt. Meine lieben Freundinnen und Freunde Josef Gebel, Marie-Hélène Mathieu, Bernhard Kipper, Petra Naumann-Kipper, Christian Thiel und Susanne Wetzel hatten immer Zeit für Diskussionen und motivierende Gespräche.

# Inhaltsverzeichnis

<b>1</b>	<b>Einleitung</b>	<b>5</b>
<b>2</b>	<b>Allgemeine Begriffe und Bezeichnungen</b>	<b>7</b>
<b>3</b>	<b>Pseudozufallszahlengeneratoren</b>	<b>9</b>
3.1	Definitionen und Begriffe . . . . .	9
3.2	Überlegungen zur Existenz von Pseudozufallszahlengeneratoren . . . . .	14
3.3	Verschiedene Pseudozufallszahlengeneratoren in der Theorie . . . . .	20
3.3.1	Der Generator nach Blum, Blum und Shub (Quadratgenerator) . . . . .	20
3.3.2	Der RSA-Bit-Generator . . . . .	22
3.3.3	Der modifizierte Rabin-Bit-Generator . . . . .	22
3.3.4	Der diskrete Exponentialgenerator . . . . .	23
3.3.5	Der Generator auf elliptischen Kurven . . . . .	23
<b>4</b>	<b>Die Klasse RandomBase in LiSA</b>	<b>24</b>
4.1	Vorbemerkungen . . . . .	24
4.2	Aufbau von LiSA . . . . .	26
4.3	Beschreibung der Klasse RandomBase . . . . .	28
<b>5</b>	<b>Der Seedgenerator</b>	<b>30</b>
5.1	Grundsätzliche Vorbemerkungen . . . . .	30
5.2	Die Standardseedgenerator in LiSA . . . . .	31
5.3	Statistische Tests . . . . .	32

<b>6</b>	<b>Implementierte Pseudozufallszahlengeneratoren</b>	<b>37</b>
6.1	Der Blum-Blum-Shub <i>BBS</i> Generator . . . . .	38
6.2	Der RSA-Bit-Generator . . . . .	38
6.3	Der modifizierte Rabin-Bit-Generator . . . . .	38
6.4	Der diskrete Exponentialgenerator I . . . . .	39
6.5	Der diskrete Exponentialgenerator II . . . . .	39
6.6	Der Generator auf elliptischen Kurven I . . . . .	40
<b>7</b>	<b>Zusammenfassung – Ergebnisse</b>	<b>41</b>
7.1	Laufzeitergebnisse des Generators nach Blum, Blum und Shub . . . . .	42
7.2	Laufzeitergebnisse des RSA-Bit-Generators . . . . .	43
7.3	Laufzeitergebnisse des modifizierten Rabin-Bit-Generators . . . . .	44
7.4	Laufzeitergebnisse des diskreten Exponentialgenerators I . . . . .	45
7.5	Laufzeitergebnisse des diskreten Exponentialgenerator II . . . . .	46
7.6	Laufzeitergebnisse des Generators auf elliptischen Kurven . . . . .	47
<b>A</b>	<b>Implementierungen</b>	<b>48</b>
A.1	Der Generator nach Blum-Blum-Shub . . . . .	48
A.2	Der RSA-Bit-Generator . . . . .	50
A.3	Der modifizierte Rabin-Bit-Generator . . . . .	52
A.4	Der diskrete Exponentialgenerator I . . . . .	54
A.5	Der diskrete Exponentialgenerator II . . . . .	59
A.6	Der Generator auf elliptischen Kurven . . . . .	63
A.7	Der Seedgenerator . . . . .	68

# Kapitel 1

## Einleitung

In den letzten Jahren haben—im Rahmen einer zunehmenden Vernetzung von Rechnern in lokalen als auch in weltweiten Netzwerken—Sicherheitsaspekte an Bedeutung gewonnen. Hierbei spielen Schutzmechanismen für unbefugten Zugriff auf private Daten, sowie Authentifizierungsverfahren und Identifizierung von Kommunikationspartnern eine wichtige Rolle. Es existieren eine Reihe verschiedener Implementierungen von kryptographischen Verfahren in diversen Softwarepaketen, um Daten zu schützen bzw. Kommunikation sicher zu machen (siehe auch [Kan94], [Sch96a]). Diese Implementierungen sind jedoch nicht so allgemein gehalten, daß sie ohne großen Aufwand in bestehende Programme integriert werden können.

Daher begann die Arbeitsgruppe von Herrn Professor Johannes Buchmann an der Universität des Saarlandes mit der Entwicklung einer neuen, objektorientierten Klassenbibliothek für kryptographische Verfahren — **LiSA** (**L**ibrary for **S**ecure **A**pplications) — die vor allem mit dem Ziel einer leichten Portierbarkeit und guten Handhabung entworfen wurde (siehe [BKM<sup>+</sup>96]). Als Programmiersprache für **LiSA** wurde C++ (siehe zum Beispiel in [Jv91]) ausgewählt. Ferner sollte die bisher am Lehrstuhl von Herrn Professor Buchmann entstandene Software, darunter die C++-Klassenbibliothek **LiDIA** (siehe [BBP95]) für algorithmische Zahlentheorie, soweit wie möglich genutzt werden.

Inzwischen sind in **LiSA** eine Reihe konkreter (Pseudo-)Zufallszahlengeneratoren (dies geschah im Rahmen dieser Diplomarbeit), kryptographischer Hashfunktionen, Verschlüsselungs- und Signaturverfahren implementiert worden (siehe [Sch96a]). Darüber hinaus bietet **LiSA** nun eine integrierte Schlüsselgenerierung, -verwaltung und -verteilung (siehe [Ken96]). Insbesondere sind die in **LiSA** realisierten Funktionen mit geringem Aufwand in bestehende Programme integrierbar.

Ziel der vorliegenden Arbeit ist die Beschreibung verschiedener Pseudozufallszahlengeneratoren (siehe Definition 3.6) und ihre Implementierung in **LiSA**. Bei der Realisierung dieser Arbeit ergab sich ferner die Notwendigkeit, Möglichkeiten der Erzeugung „echter“ Zufallszahlen in **LiSA** zu untersuchen, und die schließlich benutzte „Zufallsquelle“

(siehe dazu [Sch96b]) hinsichtlich ihrer Verwertbarkeit zu testen. Dazu wurden verschiedene *statistische Tests* (siehe zum Beispiel [Knu81], [Mau90]) implementiert und auf die „Zufallsquelle“ angewendet.

Die Arbeit gliedert sich wie folgt:

- In Kapitel 2 werden allgemeine Begriffe und Definitionen, die für das allgemeine Verständnis der Arbeit notwendig sind, eingeführt.
- In Kapitel 3 definieren wir zunächst den Begriff des *Pseudozufallszahlengenerators*. Wir fassen die für die Arbeit wichtigen Resultate der theoretischen Kryptographie im Zusammenhang mit der Existenz von Pseudozufallszahlengeneratoren im allgemeinen zusammen und geben danach im einzelnen die notwendigen Annahmen und Voraussetzungen für die Korrektheit der von uns in **LiSA** implementierten Pseudozufallszahlengeneratoren an. Schließlich wird das Kapitel durch die Beschreibung der verwendeten Pseudozufallszahlengeneratoren abgeschlossen.
- In Kapitel 4 erklären wir den Aufbau der C++-Klassenbibliothek **LiSA** und im speziellen ihrer objektorientierten Klasse (siehe [Jv91]) **RandomBase**, die uns von **LiSA** als grundlegende Klasse bei der Implementierung der Pseudozufallszahlengeneratoren vorgegeben wurde.
- In Kapitel 5 beschäftigen wir uns mit der Frage, wie man auf einem Rechnersystem eine „Quelle für echte Zufallszahlen“ installieren kann. Nach der Abwägung verschiedener Möglichkeiten begründen wir unsere Entscheidung für das Verfahren von [LMS93], das leicht abgewandelt in [Sch96b] implementiert wurde. Ferner geben wir mehrere statistische Tests zur weiteren praktischen Untersuchung dieser „Zufallsquelle“ an, sowie die bei diesen Tests erzielten Ergebnisse.
- In Kapitel 6 gehen wir auf die weiteren Besonderheiten bei unserer Implementierung der Pseudozufallszahlengeneratoren ein. Wir beschreiben Strategien, Verfahren und „Tricks“, die über die abstrakte Beschreibung der Pseudozufallszahlengeneratoren hinausgehen.
- In Kapitel 7 werden die Laufzeitergebnisse der Pseudozufallszahlengeneratoren angegeben.

# Kapitel 2

## Allgemeine Begriffe und Bezeichnungen

Im weiteren Verlauf dieser Arbeit steht  $\Sigma$  immer für ein Alphabet, das neben anderen die Zeichen  $0$ ,  $1$ ,  $\$$  und  $\#$ , aber nicht die Zeichen  $L$  und  $R$  enthält. Wir schreiben  $\Sigma_0$  für die Menge  $\Sigma - \{\#\}$  und  $\Sigma_\$$  für die Menge  $\Sigma - \{\$, \#\}$ .

Für grundlegende Begriffe und Definitionen aus dem Bereich der Mathematik verweisen wir zum Beispiel auf die Lehrbücher [Fis95], [Mey76] oder [RSV94]. Aus Gründen der Übersichtlichkeit geben wir jedoch an dieser Stelle einige häufig von uns benutzte Definitionen und Bezeichnungen an:

- $\mathbb{N}$  bezeichnet die Menge der **natürlichen Zahlen**  $\{1, 2, 3, \dots\}$ .
- $\mathbb{N}_0$  bezeichnet die Menge  $\mathbb{N} \cup \{0\}$ .
- $\mathbb{Z}$  bezeichnet die Menge der **ganzen Zahlen**  $\{\dots, -3, -2, -1, 0, 1, 2, 3, \dots\}$ .
- Eine **Gruppe** ist ein Paar  $(G, \circ)$ , wobei  $G$  eine Menge und ‘ $\circ$ ’ eine Verknüpfung auf dieser Menge ist. Dabei ist die Menge unter dieser Verknüpfung abgeschlossen, die Verknüpfung ist assoziativ, d.h.  $(a \circ b) \circ c = a \circ (b \circ c)$  für alle  $a, b \in G$ , und die Menge  $G$  enthält ein **neutrales Element**  $e$ , so daß für alle  $g \in G$  gilt:  $e \circ g = g \circ e = g$ . Ferner gibt es zu jedem Element  $g \in G$  ein **inverses Element**  $g^{-1} \in G$  mit  $g \circ g^{-1} = g^{-1} \circ g = e$ .
- Eine Gruppe  $(G, \circ)$  heißt **kommutativ** oder **abelsch**, falls für alle  $a, b \in G$  gilt:  $a \circ b = b \circ a$ .
- Sei  $G$  eine Gruppe, sei  $g \in G$  und sei  $\ell$  die minimale positive, ganze Zahl mit  $g^\ell = 1$ . Dann heißt  $\ell$  die **Ordnung** von  $g$ .

- Ein **Körper** ist eine Menge  $K$  mit zwei Verknüpfungen ‘+’ und ‘·’, für die  $(K, +)$  und  $(K - \{e\}, \cdot)$  jeweils abelsche Gruppen sind, wobei  $e$  das neutrale Element der additiven Gruppe  $(K, +)$  ist. Für die Gruppen muß ferner die Distributivität der beiden Verknüpfungen gelten, d.h für alle  $a, b, c \in K$  muß gelten:  $a \cdot (b+c) = a \cdot b + a \cdot c$  und  $(a + b) \cdot c = a \cdot c + b \cdot c$ .
- Ein **endlicher Körper** ist ein Körper mit nur endlich vielen Elementen.
- Die **Charakteristik** eines Körpers ist die kleinste positive Zahl  $p$  mit  $p \cdot 1 = 0$  bzw. Null, falls keine solche Zahl existiert. Die Charakteristik eines Körpers ist entweder Null oder eine Primzahl.
- Der **Primkörper der Charakteristik  $p$**  ist der Durchschnitt aller Körper der Charakteristik  $p$ . Falls  $p$  eine Primzahl ist, ist der Primkörper der Charakteristik  $p$  isomorph zum Körper  $\mathbb{F}_p := \mathbb{Z}/p\mathbb{Z}$ . Mit  $\mathbb{F}_p^*$  bezeichnen wir die multiplikative Gruppe von  $\mathbb{F}_p$ .
- Ein Körper  $K$  heißt **algebraisch abgeschlossen**, wenn jedes Polynom mit Koeffizienten aus  $K$  über  $K$  ganz in Linearfaktoren zerfällt.
- Zu jedem Körper  $K$  existiert ein Oberkörper  $\overline{K}$ , der algebraisch abgeschlossen ist. Alle algebraisch abgeschlossenen Oberkörper von  $K$  sind zueinander isomorph. Daher spricht man von dem **algebraischen Abschluß** von  $K$ .
- Ein **Erzeuger**  $g$  eines endlichen Körpers  $K$  ist ein Element  $g \in K$  mit  $K = \{g^1, g^2, g^3, \dots\}$ .
- Die xor-Funktion ist eine Abbildung  $\text{xor}: \{0, 1\}^2 \rightarrow \{0, 1\}$  mit

$$\text{xor}(b_1, b_2) = \begin{cases} 0 & \text{falls } b_1 = b_2 \\ 1 & \text{sonst} . \end{cases}$$

Statt  $\text{xor}(b_1, b_2)$  schreiben wir auch  $b_1 \oplus b_2$ .

# Kapitel 3

## Pseudozufallszahlengeneratoren

### 3.1 Definitionen und Begriffe

In kryptographischen Anwendungen muß jeder Teilnehmer über eine „geheime“, nur ihm bekannte Information verfügen. Ist dies nicht der Fall, so kann ein potentieller Gegner alle Berechnungen des Teilnehmers nachvollziehen oder sogar vorhersagen, und ist damit in der Lage, alle Bemühungen, die der Teilnehmer zu seiner Sicherheit anstellen könnte, zu umgehen oder zu manipulieren. Diese „geheime“ Information darf andererseits nicht vom Teilnehmer durch ein „einfaches“ Verfahren gewonnen worden sein, da der Gegner mit Hilfe des gleichen Verfahrens auf die gleiche Information schließen könnte. Dies ist möglich, da man in jedem Sicherheitsmodell sinnvollerweise annehmen muß, daß ein Gegner zumindest über die gleichen Fähigkeiten wie jeder andere Teilnehmer an einem kryptographischen Protokoll verfügt.

Um dieses Problem zu umgehen, wird die spezielle Geheiminformation aus einer Menge von vielen verschiedenen, möglichen Informationen „gleichverteilt und zufällig“ gewählt. Damit erreicht man, daß niemand Rückschlüsse auf die Form und den Inhalt der speziell ausgewählten Information ziehen kann. (Man könnte die Information nur erraten, wofür die Wahrscheinlichkeit jedoch mit wachsender Länge der Information exponentiell kleiner wird). Beispiele für solche Geheiminformationen sind z.B. Schlüssel von Verschlüsselungs-, Signatur- und Authentisierungsverfahren.

Leider ist es aber bisher nicht möglich, Quellen anzugeben, die beweisbar zufällig und gleichverteilt Ausgaben aus einer vorgegebenen Menge liefern. Man hat bisher nur Vermutungen oder bestenfalls Hinweise, daß bestimmte physikalische Quellen diese gewünschte Eigenschaft (annähernd) besitzen (siehe [Agn88]). Doch diese vermuteten Quellen erweisen sich in der Praxis als sehr langsam und sind darüber hinaus nur sehr aufwendig zu realisieren, z.B. durch spezielle Zusatzhardware (siehe [AT&T86]). Man braucht also Verfahren, die die gewonnene zufällige Ausgabe möglichst effizient benutzen.

In unserem Berechnungs- und kryptographischen Sicherheitsmodell werden wir davon ausgehen, daß die Geheiminformationen bzw. die Ausgaben aus einer vermuteten Zufallsquelle Bitstrings, d.h. Strings über dem Alphabet mit den Zeichen 0 und 1, sind. Wir interpretieren diese Strings oft als die entsprechende, in Binärdarstellung angegebene, ganze Zahl. Aus diesen zufälligen Strings bzw. Zahlen berechnen bestimmte deterministische Funktionen, die sogenannten *Pseudozufallszahlengeneratoren*, längere Strings, die zwar nun nicht mehr zufällig sind, aber in einer kryptographischen Anwendung von ebenso langen, wirklich zufälligen Strings nicht unterschieden werden können. Die von Pseudozufallszahlengeneratoren ausgegebenen Strings nennt man daher auch *Pseudozufallszahlen*.

Pseudozufallszahlengeneratoren sind definitionsgemäß deterministische Algorithmen, sie erzeugen bei gleichen Eingaben jeweils gleiche Ausgaben. Da diese Ausgaben im Fall, daß die Eingabe des Pseudozufallszahlengenerators wirklich zufällig ist, für andere Algorithmen wie zufällige Strings wirken, läßt sich mit Hilfe jedes Pseudozufallszahlengenerators ein Verschlüsselungsverfahren konstruieren, daß analog zum klassischen *One-time-Pad* arbeitet: In diesem wird eine Nachricht in Form eines Bitstrings dadurch verschlüsselt, daß sie bitweise mit einem zufälligen Bitstring (Schlüssel) gleicher Länge durch die Operation xor (siehe Kapitel 2) verknüpft wird. Zum Entschlüsseln des verschlüsselten Textes wird die gleiche Vorgehensweise mit dem gleichen Schlüssel angewandt. Zwar ist das One-time-Pad ein informationstheoretisch sicheres Verschlüsselungsverfahren (siehe [Sha49]), doch muß vor jedem Austausch verschlüsselter Texte ein gleichlanger, zufälliger und geheimer Schlüssel über einen sicheren, nicht-abhörbaren Kanal ausgetauscht werden. Neben dem hohen Aufwand, der zur Erzeugung dieses langen Schlüssels notwendig ist, besteht also wieder das Problem der sicheren Datenübertragung, dieses Mal die Übertragung der Schlüssel. Diese Probleme lassen sich durch die Verwendung eines Pseudozufallszahlengenerators reduzieren. Sender und Empfänger benutzen beide denselben Pseudozufallszahlengenerator. Der Sender startet den Pseudozufallszahlengenerator mit einer kleineren, echten Zufallszahl als Eingabe und verknüpft die Nachricht mit dem ausgegebenen, längeren Pseudozufallsstring. Dann muß er nur die kleine Eingabe des Generators über einen sicheren, nicht-abhörbaren Kanal übertragen.

Nach dieser informalen Beschreibung von Pseudozufallszahlengeneratoren führen wir zunächst einige Begriffe ein, die für das Verständnis und weitere Definitionen notwendig sind. Wir gehen davon aus, daß der Leser mit den Grundlagen der Theorie deterministischer und nichtdeterministischer Turingmaschinen vertraut ist. Die von uns in diesem Zusammenhang verwendeten Begriffe und Schreibweisen lehnen sich im wesentlichen an die Darstellungen von Lewis und Papadimitriou (siehe [LP81]) an.

Wir verstehen unter einer **Turingmaschine**  $M$  (mit  $k$  Bändern,  $k \in \mathbb{N}$ ) ein Fünftupel  $M = (K, \Sigma, \Delta, b, h)$ , das folgenden Bedingungen genügt:

- $K$  ist eine endliche Menge von **Zuständen**.

- $\Sigma$  ist ein **endliches Alphabet** mit  $\# \in \Sigma$  und  $L, R \notin \Sigma$ .
- $\Delta \subseteq ((K - \{h\}) \times \Sigma^k) \times (K \times (\Sigma \cup \{L, R\})^k)$  ist eine **Menge von Übergängen**, die die folgende Bedingung erfüllt: für jeden Zustand  $q \in K - \{h\}$  und für jeden String  $a \in \Sigma^k$  enthält die Menge

$$\Delta \cap (\{q\} \times \{a\} \times (K \times (\Sigma \cup \{L, R\})^k)) \quad (3.1)$$

wenigstens einen Übergang.  $\Delta$  heißt auch **Übergangsrelation**.

- $b \in K$  ist der **Startzustand**.
- $h \in K$  ist der **Haltezustand**.

Falls  $\Delta$  eine Funktion von  $((K - \{h\}) \times \Sigma^k)$  nach  $(K \times (\Sigma \cup \{L, R\})^k)$  ist, nennen wir  $M$  eine **deterministische**, ansonsten eine **nichtdeterministische** Turingmaschine.

Um probabilistische Verfahren beschreiben zu können, wurde die *probabilistische Turingmaschine* eingeführt. Diese ist eine nichtdeterministische Turingmaschine, deren Übergangsrelation mit einer Wahrscheinlichkeitsfunktion verbunden ist, d.h. man kann die Wahrscheinlichkeit angeben, mit der ein bestimmter Übergang erfolgt. Dabei beträgt die Summe der Wahrscheinlichkeiten, von einer Konfiguration in eine andere überzugehen, immer 1.

**3.1. Definition (s. [BB91])** *Eine probabilistische Turingmaschine (PTM) ist ein Paar  $Z = (M, p)$ , wobei*

- $M = (K, \Sigma, \Delta, s)$  eine  $k$ -Band nichtdeterministische Turingmaschine ist und
- $p: \Delta \rightarrow [0, 1]$  eine Funktion, die die Wahrscheinlichkeit jedes Übergangs aus  $\Delta$  bestimmt, d.h. für jedes  $q \in K - \{h\}$  und  $\mathbf{a} \in \Sigma^k$  gilt

$$\sum_{d \in \Delta(q, \mathbf{a})} p(d) = 1,$$

mit  $\Delta(q, \mathbf{a}) = \Delta \cap (\{q\} \times \{\mathbf{a}\} \times (K \times (\Sigma \cup \{L, R\})^k))$ .

Ist  $Z$  eine probabilistische Turingmaschine, so bezeichnen wir mit  $\Pr\{Z(x) = y\}$  für  $x, y \in \Sigma_0^*$  die **Wahrscheinlichkeit**, daß die Turingmaschine  $Z$  bei Eingabe  $x$  den String  $y$  ausgibt.

Die Länge einer Berechnung von  $Z$  bei Eingabe  $x \in \Sigma_0^*$  heißt *Laufzeit* dieser Berechnung. Gibt es eine Funktion  $T: \mathbb{N} \rightarrow \mathbb{N}$ , so daß die Laufzeit aller Berechnungen für alle Eingaben  $x \in \Sigma_0^*$  höchstens  $T(|x|)$  ist, so nennen wir  $Z$  eine *probabilistische Turingmaschine mit durch  $T$  beschränkter Laufzeit*. Hat eine Turingmaschine polynomiell beschränkte (erwartete) Laufzeit, so sprechen wir einfach von einer polynomiellen Turingmaschine.

**3.2. Definition** Eine Wahrscheinlichkeitsverteilung  $V$  ist eine Funktion  $V : \Sigma^* \rightarrow [0, 1]$ , für die gilt:

$$\sum_{x \in \Sigma^*} V(x) = 1.$$

**3.3. Definition** Sei  $I$  eine unendliche Menge. Eine Familie  $E = \{E_x\}_{x \in I}$ , wobei  $E_x$  für alle  $x \in I$  eine Wahrscheinlichkeitsverteilung ist, heißt **Ensemble**.

Offensichtlich ist für jedes  $x \in \Sigma^*$  die Funktion  $\Pr\{M(x) = \cdot\} : \Sigma^* \rightarrow [0, 1]$  eine Wahrscheinlichkeitsverteilung. Somit läßt sich das Ausgabeverhalten einer probabilistischen Turingmaschine auch als Ensemble auffassen.

Ein Kriterium dafür, daß ein Algorithmus Pseudozufallszahlengenerator ist, ist daß die Ausgaben von Pseudozufallszahlengeneratoren bei zufälliger Eingabe von anderen Algorithmen (Gegnern) wie „echte Zufallszahlen behandelt werden“, also in anderen Worten nicht von „echten Zufallszahlen unterschieden werden“ können. Im folgenden werden wir den Begriff des „Unterscheidens“ präzisieren. Da wir Ausgabewahrscheinlichkeiten asymptotisch in Abhängigkeit von der Länge der Eingabe der probabilistischen Turingmaschinen betrachten wollen, führen wir zunächst den nachstehenden Begriff ein:

**3.4. Definition** Eine Funktion  $f : \mathbb{N} \rightarrow \mathbb{R}_0^+$  heißt **vernachlässigbar**, wenn es für alle Polynome  $p \in \mathbb{N}[x]$  eine Konstante  $N \in \mathbb{N}$  gibt, so daß für alle  $n \in \mathbb{N}$  mit  $n \geq N$  gilt:

$$f(n) < \frac{1}{p(n)} \quad .$$

Entsprechend nennen wir eine Funktion  $f : \Sigma^* \rightarrow \mathbb{R}_0^+$  **vernachlässigbar**, wenn es für alle Polynome  $p \in \mathbb{N}[y]$  eine Konstante  $N \in \mathbb{N}$  gibt, so daß für alle Strings  $x \in \Sigma^*$  mit  $|x| \geq N$  gilt:

$$f(x) < \frac{1}{p(|x|)} \quad .$$

Siehe [Mey92] und [Thi93]

**3.5. Definition (siehe z.B. [Mey92] und [Thi93])** Seien  $V = \{V_n\}_{n \in \mathbb{N}}$  und  $W = \{W_n\}_{n \in \mathbb{N}}$  Ensembles. Wir nennen  $V$  und  $W$  (**algorithmisch**) **ununterscheidbar**, wenn für alle polynomiellen, probabilistischen Turingmaschinen  $M$  die Funktion

$$f(n) = \left| \sum_{y \in \Sigma^*} \Pr(M(y) = 1) (V_n(y) - W_n(y)) \right| \quad (3.2)$$

vernachlässigbar ist. Gibt es eine polynomielle, probabilistische Turingmaschine, für die  $f(n)$  nicht vernachlässigbar ist, so sagt man, daß diese die beiden Ensembles **unterscheidet** und nennt die Maschine einen **Unterscheider**.

Zur Vereinfachung treffen wir folgende Vereinbarung: Ist  $G$  eine deterministische Turingmaschine und  $V$  eine Verteilung, dann bezeichnen wir mit  $G(V)$  diejenige Verteilung, die sich aus den Ausgaben von  $G$  ergibt, wenn die Eingaben von  $G$  gemäß der Verteilung  $V$  gewählt werden, d.h.

$$G(V)(z) = \sum_{x \in \Sigma^* \text{ mit } G(x)=z} V(x)$$

für alle  $z \in \Sigma^*$ . Damit können wir statt (3.2) auch

$$f(n) = |\Pr(M(V_n) = 1) - \Pr(M(W_n) = 1)|$$

schreiben.

Anschaulich kann man sich den Begriff der Ununterscheidbarkeit (oder entsprechend der Unterscheidbarkeit) folgendermaßen (wenn auch etwas vereinfacht) klarmachen: Zwei Ensembles sind ununterscheidbar, wenn keine Maschine  $M$  mit nicht-vernachlässigbarer Trefferrate unterscheiden kann, von welchem Ensemble sie ihre Eingabe erhält. Dabei bedeutet „unterscheiden“, daß sie sich auf eines der Ensembles festlegt, und immer, wenn sie glaubt, daß sie Eingaben gemäß dieses Ensembles enthält, das Zeichen 1 ausgibt. Glaubte die Maschine, daß die Eingabe dem anderen Ensemble zuzurechnen ist, kann sie einen beliebigen String ausgeben.

Damit liegt es nahe, von einem Pseudozufallszahlengenerator zu fordern, daß das Ensemble, das seinem Ausgabeverhalten bei zufälligen Eingaben entspricht, von einer Gleichverteilung ununterscheidbar ist.

Im folgenden bezeichnen wir für  $n \in \mathbb{N}$  mit  $U_n$  die **Gleichverteilung** auf Strings der Länge  $n$ , die definiert ist durch  $U_n : \Sigma^n \rightarrow [0, 1]$  mit  $U_n(x) = \frac{1}{2^n}$  falls  $x \in \Sigma^n$  und  $|x| = n$  und  $U_n(x) = 0$  sonst. Ist ferner  $G$  eine deterministische Turingmaschine, dann bezeichnen wir gemäß obiger Vereinbarung mit  $G(U_n)$  diejenige Verteilung, die sich aus den Ausgaben von  $G$  ergibt, wenn die Eingaben von  $G$  gemäß der Verteilung  $U_n$  gewählt werden.

**3.6. Definition** *Eine deterministische Turingmaschine  $G$  mit polynomieller Laufzeit heißt **Pseudozufallszahlengenerator**, wenn folgendes gilt:*

- *Es gibt eine Funktion  $\ell : \mathbb{N} \rightarrow \mathbb{N}$  mit  $\ell(n) > n$  für alle  $n \in \mathbb{N}$ , so daß für alle  $x \in \{0, 1\}^*$  gilt:  $G(x) \in \{0, 1\}^{\ell(|x|)}$ .*
- *Die Ensembles  $\{G(U_n)\}_{n \in \mathbb{N}}$  und  $\{U_{\ell(n)}\}_{n \in \mathbb{N}}$  sind **ununterscheidbar**, wobei  $U_n$  bzw.  $U_{\ell(n)}$  die Gleichverteilung auf Strings der Länge  $n$  bzw.  $\ell(n)$  bezeichnen.*

Pseudozufallszahlengeneratoren erzeugen aus einem kurzem Zufallsstring, dem sogenannten **Seed**, einen polynomiell langen String, der für alle polynomiellen Turingmaschinen

zufällig aussieht. Dabei muß man jedoch beachten, daß die Pseudozufallsstrings keinesfalls wirklich zufällige Strings aus der Menge aller Strings der Ausgabelänge sind. Es gilt nämlich:

$$|G(\{0, 1\}^n)| \leq 2^n \ll |\{0, 1\}^{\ell(n)}| = 2^{\ell(n)}.$$

## 3.2 Überlegungen zur Existenz von Pseudozufallszahlengeneratoren

Leider konnte bisher die Existenz von Pseudozufallszahlengeneratoren noch nicht ohne weitere Annahmen bewiesen werden. Es ist nur bekannt, daß es Pseudozufallszahlengeneratoren genau dann gibt, wenn sogenannte *One-way Funktionen* existieren.

**3.7. Definition** Eine Funktion  $f : \Sigma^* \rightarrow \Sigma^*$  heißt **One-way Funktion**, wenn gilt:

- Es existiert eine deterministische, polynomielle Turingmaschine  $M$ , so daß für alle  $x \in \Sigma^*$  gilt

$$M(x) = f(x).$$

- Für alle polynomiellen, probabilistischen Turingmaschinen  $N$  ist die Funktion

$$g(n) = \frac{1}{2^n} \sum_{z \in \Sigma^n} \Pr\{N(f(z), 1^n) \in f^{-1}(f(z))\}$$

vernachlässigbar.

Zur Vereinfachung schreiben wir auch  $g(n) := \Pr\{N(f(U_n), 1^n) \in f^{-1}(f(U_n))\}$ .

Anschaulich bedeutet dies, daß eine solche One-way Funktion „einfach“ (d.h. in Polynomzeit) zu berechnen, aber „schwierig“ (d.h. nicht in probabilistischer Polynomzeit) zu invertieren ist.

Zur Zeit weiß man von keiner Funktion mit Sicherheit, ob sie eine One-way Funktion ist. Man *vermutet* nur, daß dies bei einigen Funktionen der Fall sein könnte. So vermutet man, daß es ungleich schwieriger ist, aus einer berechneten Potenz modulo einer großen Primzahl den Exponenten (das diskrete Logarithmenproblem) zu konstruieren, als umgekehrt die Potenz zu berechnen. In diesem Sinne wäre dann die Exponentiation modulo größerer Primzahlen eine One-way Funktion.

Der Zusammenhang zwischen der Existenz von Pseudozufallszahlengeneratoren und der Existenz von One-way Funktionen wird durch folgenden, in [HILL90] und [Lev87] bewiesenen Satz hergestellt:

**3.8. Satz** *Es gibt One-way Funktionen genau dann, wenn es Pseudozufallszahlengeneratoren gibt.*

Es gibt daher auch keinen einzigen Algorithmus, von dem man zur Zeit beweisen kann, daß er ein Pseudozufallszahlengenerator ist. Allerdings gibt es Algorithmen, von denen man—ähnlich wie im Fall der One-way Funktionen—annimmt, daß sie Pseudozufallszahlengeneratoren sind. Da die Resultate in [HILL90] und [Lev87] in Bezug auf die Existenz von Pseudozufallszahlengeneratoren konstruktiv sind, finden sich natürlich die Annahmen, die im Zusammenhang von One-way-Funktionen genannt werden, auch in der Beschreibung und den „Korrektheitsbeweisen“ der möglichen Pseudozufallszahlengeneratoren wieder. Insbesondere verwenden wir bei den von uns implementierten Pseudozufallszahlengeneratoren folgende Probleme und (bisher unbewiesene) Annahmen:

Wir benötigen zunächst folgende Definitionen:

Sei  $n$  eine natürliche Zahl. Eine ganze Zahl  $y$ , für die es ein  $x \in \mathbb{Z}/n\mathbb{Z}$  mit  $y \equiv x^2 \pmod{n}$  gibt, heißt **quadratischer Rest** in  $\mathbb{Z}/n\mathbb{Z}$ . Jedes solche  $x$  heißt **Quadratwurzel** von  $y$  modulo  $n$ . Ist  $n$  das Produkt zweier verschiedener Primzahlen  $p_1$  und  $p_2$  mit  $p_1, p_2 > 2$ , dann hat jeder quadratische Rest  $y$  modulo  $n$ , für den  $\text{ggT}(y, n) = 1$  ist, genau vier Quadratwurzeln. Gilt außerdem  $p_1 \equiv p_2 \equiv 3 \pmod{4}$ , dann ist  $-1$  ein quadratischer Nichtrest modulo  $p_1$  und modulo  $p_2$ , und aus diesem Grund folgt, daß genau eine dieser vier Quadratwurzeln ein quadratischer Rest modulo  $n$  sein muß.

Das Legendre Symbol gibt für eine Zahl  $y$  und eine Primzahl  $p$  an, ob diese ein quadratischer Rest modulo  $p$ , ein quadratischer Nichtrest oder eine Zahl ist, die teilbar ist durch  $p$ .

**3.9. Definition** *Sei  $y$  eine ganze Zahl und sei  $p > 2$  eine Primzahl. Man definiert das Legendre Symbol  $\left(\frac{y}{p}\right)$  als 0, 1 oder  $-1$ , wie folgt:*

$$\left(\frac{y}{p}\right) = \begin{cases} 0, & \text{wenn } p|y. \\ 1, & \text{wenn } y \text{ ein quadratischer Rest mod } p \text{ ist.} \\ -1, & \text{wenn } y \text{ ein quadratischer Nichtrest mod } p \text{ ist.} \end{cases}$$

Die Berechnung des Legendre Symbols (siehe [Kob94]) kann in Polynomzeit durchgeführt werden.

Da  $N$  ein Produkt aus zwei Primzahlen  $p_1$  und  $p_2$  ist, und das Legendre Symbol aber nicht multiplikativ ist, führen wir ein weiteres Symbol ein, das Jacobi Symbol.

**3.10. Definition** Sei  $a$  eine ganze Zahl und sei  $n$  eine positive, ungerade Zahl. Sei weiterhin  $n = p_1^{\alpha_1} \cdots p_r^{\alpha_r}$  mit  $\alpha_i \in \mathbb{N}$  und  $1 \leq i \leq r$  eine Primzahlenfaktorisierung von  $n$ . Dann definieren wir das **Jacobi Symbol**  $\left(\frac{a}{n}\right)$  als ein Produkt von Legendre Symbolen der Primfaktoren von  $n$ :

$$\left(\frac{a}{n}\right) = \left(\frac{a}{p_1}\right)^{\alpha_1} \cdots \left(\frac{a}{p_r}\right)^{\alpha_r}.$$

Man beachte; Wenn das Jacobi Symbol  $\left(\frac{a}{n}\right) = 1$  ist, bedeutet dies nicht notwendigerweise, daß  $a$  ein Quadrat modulo  $n$  ist. Wenn das Jacobi Symbol  $-1$  ergibt, dann ist eine ungerade Anzahl von Faktoren ein quadratischer Nichtrest und darüber hinaus die Zahl selbst ein quadratischer Nichtrest.

**3.11. Definition** Sei  $N$  eine positive ganze Zahl. Die **Eulersche Phi-Funktion**  $\phi(N)$  ist definiert durch die Anzahl der nicht negativen ganzen Zahlen  $b$ , die kleiner sind als  $N$  und die teilerfremd sind zu  $N$ :

$$\phi(N) \stackrel{\text{def}}{=} |\{0 \leq b < N \mid \text{ggT}(b, N) = 1\}|,$$

d.h. die Eulersche Phi-Funktion zählt die Anzahl der multiplikativen inversen Elemente  $(\text{mod } N)$ .

- *Die Quadratische Restannahme:*

Es gibt keinen probabilistischen Polynomzeitalgorithmus, der bei Eingabe einer natürlichen Zahl  $n$ , die das Produkt zweier verschiedener Primzahlen  $p_1$  und  $p_2$  ist, und eines Elements  $x \in \mathbb{Z}/n\mathbb{Z}$  mit  $\left(\frac{x}{n}\right) = 1$  mit Wahrscheinlichkeit  $1/2 + 1/q(\log n)$  bestimmt, ob  $x$  ein quadratischer Rest modulo  $n$  ist, wobei  $q \in \mathbb{Z}[X]$  ein beliebiges Polynom mit ganzzahligen Koeffizienten sein kann.

- *Die Faktorisierungsannahme:*

Es gibt keinen probabilistischen Polynomzeitalgorithmus, der bei Eingabe einer natürlichen Zahl  $n$ , die das Produkt zweier verschiedener Primzahlen  $p_1$  und  $p_2$  ist,  $p_1$  und  $p_2$  mit Wahrscheinlichkeit  $1/q(\log n)$  ausgibt, wobei  $q \in \mathbb{Z}[X]$  ein beliebiges Polynom mit ganzzahligen Koeffizienten sein kann.

- *Die Annahme über diskrete Logarithmen in  $(\mathbb{Z}/p\mathbb{Z})^*$ :*

Es gibt keinen probabilistischen Polynomzeitalgorithmus, der bei Eingabe einer Primzahl  $p$ , eines Generators  $g$  der multiplikativen Gruppe  $(\mathbb{Z}/p\mathbb{Z})^*$  und eines Elements  $a \in (\mathbb{Z}/p\mathbb{Z})^*$  den diskreten Logarithmus modulo  $p$  von  $a$  bzgl.  $g$ , d.h. die kleinste positive ganze Zahl  $x$  mit  $a \equiv g^x \pmod{p}$ , mit Wahrscheinlichkeit  $1/q(\log p)$  ausgibt, wobei  $q \in \mathbb{Z}[X]$  ein beliebiges Polynom mit ganzzahligen Koeffizienten sein kann.

- *Die RSA-Annahme*

Es gibt keinen probabilistischen Polynomzeitalgorithmus, der bei Eingabe einer natürlichen Zahl  $n$ , die das Produkt zweier verschiedener Primzahlen  $p_1$  und  $p_2$  ist, einer ganzen Zahl  $e$  mit  $\text{ggT}(\phi(n), e) = 1$  und eines Elementes  $x \in \mathbb{Z}/n\mathbb{Z}$ , wobei  $x \equiv m^e \pmod{n}$  mit  $m \in \mathbb{Z}/n\mathbb{Z}$  ist, das Element  $m$  mit Wahrscheinlichkeit  $1/q(\log n)$  ausgibt, wobei  $q \in \mathbb{Z}[X]$  ein beliebiges Polynom mit ganzzahligen Koeffizienten sein kann.

- *Die Annahme bzgl. diskreter Logarithmen über elliptischen Kurven:*

Zuerst werden wir die Definition einer elliptischen Kurve vorstellen. Anschließend werden wir dann einige grundlegende Sätze aus der Theorie der elliptischen Kurven und schließlich das entsprechende diskrete Logarithmenproblem vorstellen.

Sei dazu im folgenden  $K$  ein Körper der Charakteristik  $p > 3$ , und sei  $\overline{K}$  sein algebraischer Abschluß.

**3.12. Definition** Seien  $a, b \in K$  mit der Eigenschaft  $4a^3 + 27b^2 \neq 0$ .

Unter einer **elliptischen Kurve über dem Körper  $K$**  versteht man die Gesamtheit aller Lösungen aus  $\overline{K} \times \overline{K}$  einer Gleichung der Form

$$E : \quad y^2 = x^3 + ax + b \quad (3.3)$$

einschließlich einer formalen Lösung  $\mathcal{O} := (\infty, \infty)$ , der sogenannten Lösung im Unendlichen.

Die Menge aller Punkte auf  $E$ , d.h. die Menge

$$\{ (x, y) \in \overline{K}^2 \mid y^2 = x^3 + ax + b \} \cup \{ \mathcal{O} \},$$

bezeichnet man als  $E(\overline{K})$ .

Die Menge aller  $K$ -rationalen Punkte ist definiert als

$$E(K) = \{ (x, y) \in K^2 \mid y^2 = x^3 + ax + b \} \cup \{ \mathcal{O} \}. \quad (3.4)$$

**3.13. Bemerkung** Die Kurvengleichung (3.3) wird in der Literatur als **kurze Weierstraß-Normalform** bezeichnet. Die kurze Weierstraß-Normalform kann aus einer allgemeinen Kurvengleichung, der sogenannten **Weierstraß-Gleichung**, durch isomorphe Umformungen hergeleitet werden. Die allgemeine Weierstraß-Gleichung beschreibt eine elliptische Kurve in jedem beliebigen Körper, wohingegen die kurze Weierstraß-Normalform nur in Körpern der Charakteristik ungleich 2 und 3 eine elliptische Kurve korrekt beschreibt. Da wir elliptische Kurven nur über Primkörpern großer Charakteristik betrachten, wählen wir statt der allgemeinen Kurvengleichung die einfachere kurze Weierstraß-Normalform. Mehr Informationen über die allgemeine Weierstraß-Gleichungen findet man in [Sil86].

**3.14. Bemerkung** Man kann zeigen, daß die Bedingung  $4a^3 + 27b^2 \neq 0$  aus Definition 3.12 äquivalent dazu ist, daß die rechte Seite  $x^3 + ax + b$  der Kurvengleichung keine mehrfache Nullstelle in  $K$  besitzt. Eine Kurve, die diese Bedingung erfüllt, bezeichnet man als **nicht-singulär**. Bei elliptischen Kurven setzt man damit immer voraus, daß sie nicht-singulär sind.

Es stellte sich schon früh heraus, daß die Gesamtheit aller  $K$ -rationalen Punkte einer elliptischen Kurve nicht nur eine einfache Menge bildet, sondern zusätzlich noch eine Struktur besitzt. Diese Struktur läßt sich durch einfache Formeln beschreiben, so daß es leicht möglich ist, in dieser Punktmenge praktisch zu rechnen. Dies wird im folgenden Satz erläutert.

**3.15. Satz** Sei  $K$  ein Körper und  $a, b \in K$ . Dann bildet die Menge der  $K$ -rationalen Punkte einer elliptischen Kurve  $E : y^2 = x^3 + ax + b$  eine abelsche (i.a. additiv geschriebene) Gruppe.

Die Addition wird dabei durch folgendes Gesetz gegeben:

- Das neutrale Element der Gruppe  $E(K)$  ist der Punkt  $\mathcal{O}$ .
- Zu einem Punkt  $\mathcal{O} \neq P = (x, y) \in E(K)$  definieren wir als negativen Punkt

$$-P := (x, -y) \in E(K).$$

- Für Punkte  $P_1$  und  $P_2$  mit  $P_1 = -P_2$  wird die Summe als  $P_1 + P_2 := \mathcal{O}$  definiert.
- Für Punkte  $P_1 = (x_1, y_1)$  und  $P_2 = (x_2, y_2)$  mit  $P_1 \neq -P_2$  wird die Summe gegeben als  $P_1 + P_2 = (x_3, y_3)$ , wobei

$$\begin{aligned} x_3 &= -x_1 - x_2 + \lambda^2 \\ y_3 &= -y_1 + \lambda \cdot (x_1 - x_3) \end{aligned}$$

mit

$$\begin{aligned} \lambda &= \frac{y_1 - y_2}{x_1 - x_2} && \text{falls } P_1 \neq \pm P_2. \\ \lambda &= \frac{3x_1^2 + a}{2y_1} && \text{falls } P_1 = P_2. \end{aligned}$$

**Beweis:** Der Beweis benutzt die geometrische Bedeutung der Addition zweier Punkte, die in der folgenden Bemerkung 3.16 beschrieben wird. Eine genaue Ausführung des Beweises findet man in dem Buch von Silverman [Sil86]. ■

**3.16. Bemerkung** Geometrisch besitzt die Addition zweier Punkte einer elliptischen Kurve folgende Bedeutung:

Seien  $P_1$  und  $P_2 \neq \pm P_1$  zwei verschiedene Punkte einer elliptischen Kurve, die beide ungleich dem Nullpunkt  $\mathcal{O}$  sind. Um die Summe der beiden Punkte zu bestimmen, legen wir zuerst eine Gerade durch die zwei Punkte  $P_1$  und  $P_2$ . Diese Gerade schneidet die elliptische Kurve mit Sicherheit in einem dritten Punkt  $Q$ . Dann erhalten wir die Summe der beiden Punkte  $P_1$  und  $P_2$  durch Spiegelung des Punktes  $Q$  an der x-Achse, d.h.  $P_1 + P_2 = -Q$ .

Zur Berechnung von  $2 \cdot P_1$  verwenden wir anstatt der Geraden die Tangente an die elliptische Kurve im Punkt  $P_1$ . Diese Tangente schneidet die Kurve ebenfalls in einem weiteren Punkt  $Q'$  und  $2 \cdot P_1$  erhalten wir wiederum durch Spiegelung des Punktes  $Q'$  an der x-Achse, d.h.  $2 \cdot P_1 = -Q'$ .

Sei ab jetzt der Körper  $K$  der Primkörper der Charakteristik  $p$  für eine feste Primzahl  $p > 3$ , d.h.  $K = \mathbb{Z}/p\mathbb{Z}$ . In Analogie zur Literatur werden wir diesen Körper im folgenden auch mit  $\mathbb{F}_p$  bezeichnen.

Das diskrete Logarithmenproblem über elliptischen Kurven über endlichen Primkörpern läßt sich nun analog zum diskreten Logarithmenproblem in  $(\mathbb{Z}/p\mathbb{Z})^*$  beschreiben, wobei allerdings die Gruppe  $(\mathbb{Z}/p\mathbb{Z})^*$  durch die additive Punktgruppe einer elliptischen Kurve ersetzt wird. Unsere Annahme lautet: *Es gibt keinen probabilistischen Polynomzeitalgorithmus, der bei Eingabe einer Primzahl  $p$ , einer elliptischen Kurve über  $\mathbb{Z}/p\mathbb{Z}$  und zweier Punkte  $P, Q$  auf der elliptischen Kurve eine ganze Zahl  $k$  mit  $P = k \cdot Q$  mit Wahrscheinlichkeit  $1/q(\log p)$  ausgibt, wobei  $q \in \mathbb{Z}[X]$  ein beliebiges Polynom mit ganzzahligen Koeffizienten sein kann.* Wir werden eine stärkere Annahme benutzen, die sich auf spezielle Kurven, nämlich supersinguläre elliptische Kurven der Form  $y^2 = x^3 + b$ , bezieht. Diese Kurven haben einerseits den Vorteil, daß man direkt die Ordnung ihrer Punktgruppe über  $\mathbb{F}_p$  kennt. Diese Ordnung ist nämlich gerade  $p + 1$ . Der Nachteil dieser Kurven ist, daß das dazu gehörende diskrete Logarithmenproblem nur so schwer ist, wie das diskrete Logarithmenproblem in einem endlichen Körper  $\mathbb{F}_q$ , wobei  $q = k * p$  für eine Konstante  $k \in \mathbb{N}_{>1}$  ist (siehe [MVO91]). Ein ähnlicher Zusammenhang für allgemeine elliptische Kurven ist nicht bekannt.

## 3.3 Verschiedene Pseudozufallszahlengeneratoren in der Theorie

### 3.3.1 Der Generator nach Blum, Blum und Shub (Quadrat-generator)

Der Generator wurde von Blum, Blum und Shub 1986 entwickelt, siehe [BBS86], und kann folgendermaßen beschrieben werden: Sei  $k$  eine positive ganze Zahl. Sei  $N$  eine ganze Zahl, die sich als das Produkt zweier zufällig gewählter  $k$ -Bit-Primzahlen  $p$  und  $q$  mit  $p \equiv q \equiv 3 \pmod{4}$  darstellen läßt. (Die Annahme, daß die Primzahlen  $p \equiv q \equiv 3 \pmod{4}$  sein müssen, garantiert, daß  $-1$  ein quadratischer Nichtrest  $\pmod{p}$  und  $\pmod{q}$  ist (siehe [Kob94][II. 2, Cor. 2])).

Bei Eingabe des zufälligen Startwertes  $x_0 \in \mathbb{Z}/N\mathbb{Z}$  berechnet der Generator eine Folge der Form

$$x_{n+1} \equiv (x_n)^2 \pmod{N}.$$

Sei schließlich  $z_{n+1}$  gegeben durch

$$z_{n+1} \equiv x_{n+1} \pmod{2}.$$

Dann stellt die Folge  $\{z_n : 1 \leq n \leq k^2\}$  die Pseudozufallsbits dar, die generiert werden aus der zufälligen Seed  $x_0$ .

Wie oben beschrieben, kann man nicht beweisen, daß dieser Algorithmus tatsächlich ein Pseudozufallszahlengenerator ist. Es gilt aber:

**3.17. Satz** *Der Generator nach Blum, Blum und Shub ist ein Pseudozufallszahlengenerator, falls die Quadratische Restannahme zutrifft.*

**Beweis:** Einen vollständigen Beweis dieses Satzes findet man zum Beispiel in [BBS86] oder [Sti95, S. 375–378]. Hier geben wir eine kurze Zusammenfassung des Beweises an, die den Zusammenhang zwischen der Quadratischen Restannahme und den Eigenschaften des Pseudozufallszahlengenerators verdeutlicht: Wir benutzen die Schreibweisen, die wir bei der Beschreibung des genannten Generators verwendet haben. Wir nehmen nun im Gegensatz zur Aussage des Satzes an, daß die Ausgabe des Generators von Blum, Blum und Shub von der Gleichverteilung der Strings der Ausgabelänge unterschieden werden kann.

1. Dann kann man aus einer probabilistischen, polynomiellen Turingmaschine, die diesen Unterschied feststellt, eine probabilistische, polynomielle Turingmaschine  $M$  konstruieren, die bei Eingabe von  $(z_2, \dots, z_{k^2})$ , also der Ausgabe des Generators ohne das  $z_0$ te und  $z_1$ te Bit, das Bit  $z_1$  (und  $z_0$ ) mit Wahrscheinlichkeit  $\frac{1}{2} + \frac{1}{(\log N)^c}$  ausgibt, wobei  $c$  eine positive Konstante ist. Dies ergibt sich als direkte Folge aus der Anwendung eines allgemeineren Satzes von Yao (siehe [Yao82]), der ein entsprechendes Resultat für ein beliebiges, von dem Ensemble  $U = \{U_n\}_{n \in \mathbb{N}}$  (Gleichverteilung) unterscheidbares Ensemble  $V = \{V_n\}_{n \in \mathbb{N}}$  angibt.
2. Aus der Turingmaschine  $M$  läßt sich nun wiederum eine weitere probabilistische, polynomielle Turingmaschine  $M'$  konstruieren, die bei Eingabe von  $N, x \in \mathbb{Z}/N\mathbb{Z}$  mit  $\left(\frac{x}{N}\right) = 1$  mit Wahrscheinlichkeit  $\frac{1}{2} + \frac{1}{(\log N)^{c'}}$ , wobei  $c' > 0$  konstant ist, angeben kann, ob  $x$  ein Quadratischer Rest modulo  $N$  sein kann. Dies wäre dann ein Widerspruch zur Quadratischen Restannahme.

Wir bezeichnen mit  $\text{QR}(N)$  die Menge der quadratischen Reste modulo  $N$ , und mit  $\widetilde{\text{QR}}(N)$  die Menge aller quadratischen Nichtreste, für die jedoch  $\left(\frac{x}{N}\right) = 1$  gilt.

Wir konstruieren aus der Maschine  $M$  die Maschine  $M'$  folgendermaßen: Bei Eingabe von  $x$  berechnet  $M'$  zunächst  $s_0 = x^2 \bmod N$  und benutzt dann den Generator von Blum, Blum und Shub mit Eingabe  $s_0$ , um die Folge  $z_1, z_2, \dots, z_{k^2}$  zu berechnen. Nun arbeitet  $M'$  wie  $M$  bei Eingabe von  $z_1, z_2, \dots, z_{k^2}$ . Wir bezeichnen ihre erhaltene Ausgabe mit  $z_0$ . Gilt nun  $(s_0 \bmod 2) = (z_0 \bmod 2)$ , dann gibt  $M'$  aus, daß  $x$  ein quadratischer Rest modulo  $N$  ist, andernfalls gibt sie aus, daß  $x \in \widetilde{\text{QR}}(N)$  gilt. Die Korrektheit von  $M'$  läßt sich folgendermaßen zeigen: Da  $N$  das Produkt zweier Primzahlen  $p$  und  $q$  mit  $p \equiv q \equiv 3 \pmod{4}$  ist, gilt  $\left(\frac{-1}{n}\right) = 1$ . Daher ist  $-1 \in \widetilde{\text{QR}}(N)$ . Wenn daher  $\left(\frac{x}{n}\right) = 1$  gilt, dann ist  $x$  die Quadratwurzel von  $s_0 = x^2$ , falls  $x \in \text{QR}(N)$ ; ist aber  $x \in \widetilde{\text{QR}}(N)$ , dann ist  $-x$  die Quadratwurzel. Wegen

$$(-x \bmod N) \bmod 2 \neq (x \bmod N) \bmod 2$$

folgt, daß  $M'$  genau dann die korrekte Antwort ausgibt, falls  $M$  korrekt  $z_0$  ausgegeben hat. Offensichtlich ist  $M'$  eine polynomielle Turingmaschine. ■

Bei der Beschreibung der folgenden Generatoren gelten natürlich entsprechende Resultate unter Berücksichtigung der jeweiligen Annahmen. Für die Beweise verweisen wir auf die entsprechenden Originalarbeiten.

### 3.3.2 Der RSA-Bit-Generator

Der RSA-Bit-Generator wurde 1988 von Alexi, Chor, Goldreich und Schnorr entwickelt, siehe [ACGS88]:

Gegeben seien zwei zufällige Zahlen  $k \geq 2$  und  $m \geq 1$ , man wähle zwei zufällige Primzahlen  $p, q > 2$ , die gleichverteilt aus dem Bereich  $[2^k, 2^{k+1})$  stammen, sei ferner  $N = pq$ . Wähle weiter einen zufälligen Exponenten  $e$  mit  $\text{ggT}(e, \phi(N)) = 1$ .

Setze

$$x_{n+1} \equiv (x_n)^e \pmod{N}$$

und sei  $z_{n+1} \in \{0, 1\}$  gegeben durch

$$z_{n+1} \equiv x_{n+1} \pmod{2}.$$

Dann stellt die Folge  $\{z_n : 1 \leq n \leq k^m + m\}$  die Pseudozufallsbits dar, die generiert werden aus der zufälligen Seed  $x_0$  mit der Länge  $\geq 2k$  Bits.

**3.18. Satz** *Der RSA-Bit-Generator ist ein Pseudozufallszahlengenerator, falls die RSA-Annahme zutrifft.*

### 3.3.3 Der modifizierte Rabin-Bit-Generator

Ein weiterer Pseudozufallszahlengeneratoren ist der modifizierte Rabin-Bit-Generator, der ebenfalls 1988 von Alexi, Chor, Goldreich und Schnorr beschrieben wurde, siehe auch [ACGS88].

Gegeben sei eine zufällige, ganze Zahl  $k \geq 2$ , wähle weiter zwei zufällige Primzahlen  $p$  und  $q$  gleichverteilt aus dem Bereich  $[2^k, 2^{k+1})$  mit  $p \equiv q \equiv 3 \pmod{4}$ . Sei  $N = pq$ . Wir setzen dann

$$x_{n+1} = \begin{cases} (x_n)^2 \pmod{N}, & \text{wenn } (x_n)^2 \pmod{N} \in [0, N/2), \\ N - (x_n)^2 \pmod{N}, & \text{sonst,} \end{cases}$$

so daß  $0 \leq x_{n+1} < N/2$  ist und das  $z_{n+1}$ -Bit gegeben ist durch

$$z_{n+1} \equiv x_{n+1} \pmod{2}.$$

Dann stellt die Folge  $\{z_n : 1 \leq n \leq k^m + m\}$  die Pseudozufallsbits dar, die generiert werden mit der zufälligen Seed  $x_0$ , die wenigstens  $2k$ -Bits lang ist.

**3.19. Satz** *Der modifizierte Rabin-Bit-Generator ist ein Pseudozufallszahlengenerator, falls die Quadratische Restannahme und die RSA-Annahme zutrifft.*

### 3.3.4 Der diskrete Exponentialgenerator

Gegeben seien zwei zufällige, ganze Zahlen  $k \geq 2$  und  $m \geq 1$ , man wählt zufällig eine Primzahl  $q > 2$  gleichverteilt über alle Primzahlen aus dem Bereich  $[2^k, 2^{k+1}]$ , versehen mit einer kompletten Faktorisierung von  $q - 1$  und einem Erzeuger  $g$ . Siehe Definitionen in Kapitel 2. Setze

$$x_{n+1} \equiv g^{x_n} \pmod{q}$$

und sei das  $z_{n+1}$ -Bit das höchstwertige Bit (= most significant bit)

$$z_{n+1} \equiv \left\lfloor \frac{x_{n+1}}{2^k} \right\rfloor.$$

Dann stellt die Folge  $\{z_n : 1 \leq n \leq k^m + m\}$  die Pseudozufallsbits dar, die von der zufälligen Seed  $x_0$  generiert werden. Dieser Generator wurde von Blum und Micali 1984 und von Long und Wigderson 1988 entwickelt, siehe [BM84] und [LW88].

**3.20. Satz** *Der diskrete Exponentialgenerator ist ein Pseudozufallszahlengenerator, falls die Annahme über diskrete Logarithmen zutrifft.*

### 3.3.5 Der Generator auf elliptischen Kurven

Dieser Pseudozufallszahlengenerator wurde 1986 von Kaliski entwickelt, siehe auch [Kal86].

Man wähle eine zufällige Primzahl  $p$  mit einer kompletten Faktorisierung von  $p - 1$ . Ferner wähle man eine zufällige elliptische Kurve über  $\mathbb{F}_p$  und einen Punkt  $P = (x_0, y_0)$  auf dieser Kurve mit möglichst hoher Ordnung. Wir setzen

$$(x_{n+1}, y_{n+1}) = y_n \cdot (x_n, y_n)$$

und

$$z_{n+1} = \begin{cases} 1, & \text{falls } y_{n+1} > p/2 \\ 0, & \text{sonst} \end{cases}$$

**3.21. Satz** *Der Generator auf elliptischen Kurven ist ein Pseudozufallszahlengenerator, falls die Annahme bzgl. diskreter Logarithmen über elliptischen Kurven zutrifft.*

Entsprechend gilt

**3.22. Satz** *Der Generator auf supersingulären elliptischen Kurven ist ein Pseudozufallszahlengenerator, falls die Annahme bzgl. diskreter Logarithmen über supersingulären elliptischen Kurven zutrifft.*

# Kapitel 4

## Die Klasse RandomBase in LiSA

### 4.1 Vorbemerkungen

Der Datenaustausch innerhalb großer, für viele Benutzer zugängliche Rechnernetze ist neben seinen Vorteilen auch mit neuen Sicherheitsrisiken verbunden. So stellt sich die Frage, ob und wie man Benutzer des Netzes eindeutig identifizieren kann, wie diese Benutzer Daten austauschen können, ohne den störenden Einfluß Dritter in Kauf nehmen zu müssen, und dabei sicher sein können, daß geheime Daten nicht abgehört werden. Besonders schwerwiegend sind solche Sicherheitsrisiken, wenn es sich um weltweit verteilte Netze wie z.B. das Internet handelt und wenn die Benutzer wichtige, z.B. wirtschaftlich und juristisch verbindliche Daten bearbeiten wollen.

Kryptographische Verfahren wie Public-Key- und Signaturverfahren geben eine Möglichkeit, diese Schwierigkeiten zu lösen. Darüberhinaus existieren bereits Programme, die Nachrichten eines Benutzers identifizieren, signieren und verschlüsseln können. Es gibt auch eine Reihe mehr oder weniger umfangreicher Software-Bibliotheken für kryptographische Algorithmen (siehe zum Beispiel [Sch96a] oder [AT&T86]), die aber oft nicht mehr als eine einfache Sammlung von Implementierungen bekannter Verfahren darstellen. Ein Benutzer wird über das einfache Aufrufen der Hauptverfahren hinaus nicht unterstützt. Er kann diese Bibliotheken nur mit großem Aufwand an seine eigenen Anwendungen anpassen bzw. die Bibliotheken um weitere Verfahren ergänzen. Zudem ist die "kryptographische Unbedenklichkeit" solcher Bibliotheken für den Benutzer nur schwer überprüfbar. Um diesen Problemen zu begegnen wurde die Bibliothek CryptoManager++ (siehe [Kan94]) entwickelt. Eine starke Modularisierung sowie erste objektorientierte Ansätze führen zu größerer Verständlichkeit und Verwertbarkeit der Bibliothek. Leider wurde die Kombination einzelner bereits implementierter Verfahren und die Erweiterung der Bibliothek um neue Verfahren durch den Entwickler kaum unterstützt, durch starke Abhängigkeiten in den festgelegten Klassen sogar erschwert. Ferner wird der Benutzer bei der Bewältigung vieler, im Umfeld der Aufgabenstellung vorkommen-

der Verwaltungsaufgaben in keiner Weise unterstützt. Die Realisierung dieser Aufgaben ist mit einem im Verhältnis zum Gesamtaufwand des jeweiligen Entwicklungsprojektes großen Programmieraufwand verbunden. Allerdings wiederholen sich diese Verwaltungsaufgaben in verschiedenen Projekten oder unterscheiden sich in mehreren Projekten nur in wenigen Punkten. Dazu gehören folgende Aufgaben:

- Erzeugen, Speichern, Verwalten und Verteilen von Schlüsseln,
- Aufteilen von Daten in Blöcke zum Verschlüsseln und Rekonstruieren der Daten aus den Blöcken beim Entschlüsseln,
- Signieren von Daten und Überprüfen von signierten Daten,
- Austausch von temporären Schlüsseln,
- Verschlüsseln von Datenströmen mit kryptographisch sicheren Pseudozufallszahlen.

Eines der Hauptziele der C++-Klassenbibliothek **LiSA** ist es, diese Aufgaben aufzufangen und ein Bindestück zwischen den bestehenden kryptographischen Verfahren und den konkreten Problemstellungen in den Anwendungsprogrammen zu schaffen. Mit **LiSA** können bestehende Anwendungen mit geringem Aufwand um kryptographische Funktionalität erweitert werden.

In **LiSA** wird ein objektorientierter Ansatz verfolgt. Funktionale Eigenschaften werden in einer Klasse zusammengefaßt. (So sind z.B. die generelle Vorgehensweise des Ver- und Entschlüsseln und das Erzeugen von Schlüsseln allen Verschlüsselungsverfahren gemeinsam.) Die Implementierung der Schnittstellen wurde so gestaltet, daß eine effiziente Nutzung der Verfahren in den Anwendungsprogrammen möglich ist. Ferner sind die Schnittstellen so entworfen, daß sie auch unabhängig von dem verwendeten Verfahren sind. Das heißt, daß beim Anlegen eines Objektes einer Klasse auch das Verschlüsselungsverfahren festgelegt wird.

Für die Anwendung ist es unwichtig, welches Verfahren tatsächlich verwendet wird, wie man zu diesem Verfahren Schlüssel erzeugt, usw. Aus der Sicht des Anwenders gibt es nur noch abstrakte Objekte, die Daten ver- und entschlüsseln, und die zugehörigen Schlüssel, ohne daß er sich über den Aufbau dieser Objekte oder ihre Verwendung kümmern muß.

Weitere Funktionalitäten, die von **LiSA** abgedeckt werden, sind: Secret- und Public-Key Verschlüsselungsverfahren, Erzeugung von echten Zufallszahlen und von Pseudozufallszahlen, Hashfunktionen, Signaturen, Streamciphers und Schlüsselverwaltung.

**LiSA** erlaubt es auch diese Objekte in verschiedenen Modi (zum Beispiel Electronic-Code-Book oder Cipher-Block-Chaining (siehe [Sch96a])) zu betreiben und Objekte baustengleich zu kombinieren: Zum Beispiel existieren generische Verfahren, um aus Verschlüsselungsalgorithmen Pseudozufallszahlengeneratoren zu bilden und umgekehrt. Verschlüsselungsverfahren können hybrid zusammengesetzt werden usw.

Ein weiterer Aspekt, der gerade bei der Verwendung von Public-Key Verschlüsselungsverfahren eine große Rolle spielt, ist die Schlüsselverwaltung. Um den Anwender möglichst von dieser Aufgabe zu befreien, hat **LiSA** eine integrierte Schlüsseldatenbank und Schlüsselverwaltung, die unabhängig vom verwendeten Verschlüsselungsverfahren ist. Ein auf einem verteilten Client-Server-Konzept beruhender Schlüsselservers erlaubt es, automatisch Public-Keys unbekannter Kommunikationsteilnehmer über das Internet zu erfragen, auf ihre Gültigkeit zu überprüfen und bei Bedarf in die persönliche Datenbank zu übernehmen, (siehe [Ken96]). Wiederum ist der Anwender vor Problemen, die zum Beispiel aus dem Aufbau der verschiedenen Schlüssel entstehen, geschützt. Aus seiner Sicht gibt es nur abstrakte Objekte vom Typ Schlüssel.

## 4.2 Aufbau von LiSA

Bei der Beschreibung von **LiSA** benutzen wir die im Zusammenhang von C++ und objektorientierter Programmierung in [Jv91] gebräuchliche Begriffe. Die zur Zeit aktuelle Version der C++-Klassenbibliothek **LiSA** unterstützt den Benutzer durch Bereitstellung von verschiedenartigen kryptographischen Verfahren, die entsprechend ihrer Funktionalität in verschiedene Gruppen eingeordnet wurden. Zu jeder dieser Gruppen existiert eine sogenannte *Basisklasse*. Diese stellt eine abstrakte Beschreibung der gemeinsamen Eigenschaften und Funktionen aller Verfahren dieser Gruppe dar. In einer Basisklasse werden diese Eigenschaften und Funktionen nur durch virtuelle Deklarationen (siehe [Jv91]) beschrieben. Bisher kennt **LiSA** Basisklassen für

- Publik-Key- und Secret-Key-Verschlüsselungsverfahren, sowie Signaturverfahren,
- Hashfunktionen,
- Zufallszahlengeneratoren, und
- Schlüsselverwaltungen.

Ein einzelnes Verfahren, das zu einer Gruppe gehört, wird in einer Klasse implementiert, die von der entsprechenden Basisklasse abgeleitet wird. Dabei werden die dort virtuell deklarierten Funktionen gemäß dem aktuellen Verfahren (natürlich unter Beibehaltung des in der Basisklasse gewählten Funktionsnamens) ausprogrammiert.

Aber auch die verschiedenen Basisklassen weisen Gemeinsamkeiten auf, die sich in ihrem gleichen strukturellen Aufbau widerspiegeln. Jede Basisklasse verfügt über Funktionen, die folgende Aufgaben abdecken:

- *Objekt-Instanziierung*: Die Definition jeder Basisklasse enthält Funktionen zur Erzeugung eines Objektes mit wählbarem Typ und SuperTyp.

- *Objektunabhängige Abfragen:* Jede Basisklasse kennt alle Namen und Fähigkeiten ihrer speziellen (kryptographischen) Verfahren sowie Informationen über den/die Programmierer und das Datum der Implementierung.
- *Objektabhängige Abfragen:* Funktionen dieses Aufgabenbereichs liefern Informationen über den aktuellen Zustand bereits erzeugter Objekte.
- *Schlüsselgenerierung:* Schlüssel spielen nur in der Basisklasse für Verschlüsselungs- und Signaturalgorithmen und in der Basisklasse für Pseudozufallszahlengeneratoren eine Rolle, da Hashalgorithmen keine Schlüssel benötigen. Bevor ein Objekt der genannten Klassen verwendet werden kann, muß es mit einem Schlüssel (bei Pseudozufallszahlengeneratoren sprechen wir von einer *Seed*) initialisiert werden. Dabei kann ein Sicherheitsparameter übergeben werden, der jedoch Einfluß auf die Laufzeit hat. Die Implementierung dieses Sicherheitsparameters kann sich bei jedem Verfahren unterscheiden und ist daher auch dort ausprogrammiert.
- *die eigentliche Aufgabe der Klasse:* Dabei werden allgemeine Funktionen definiert, die das Verhalten (z.B. die Form der Ausgabe) der Klasse beschreiben. Natürlich können hier keine Funktionen aufgenommen werden, die in den einzelnen Verfahren einer Klasse jeweils eigene und verschiedene Programmteile erfordern.

Damit die genannten Funktionen jederzeit die Eigenschaften der gerade benutzten Verfahren abfragen können, wird jedes Verfahren einer Klasse durch zwei Parameter gekennzeichnet. Diese sind

1. *der SuperTyp*, der die Zugehörigkeit eines Verfahrens/Algorithmus zu einer Basisklasse angibt oder seine Verwendung beim Aufruf eines Verfahrens aus einer anderen Basisklasse beschreibt. So kann der SuperTyp auch angeben, ob ein Pseudozufallszahlengenerator dazu benutzt wird, ein Verschlüsselungsverfahren zu konstruieren, das analog zum One-Time-Pad arbeitet (siehe Abschnitt 3.1), oder ob umgekehrt aus einem Verschlüsselungsalgorithmus ein Pseudozufallszahlengenerator konstruiert wurde. Schließlich läßt sich aus einem solchen Pseudozufallszahlengenerator wieder ein Verschlüsselungsverfahren konstruieren. Um die Laufzeit möglicher so konstruierter Verfahren in einem realistischen Rahmen zu lassen, werden weitere Rekursionen nicht unterstützt, weshalb auch keine weiteren SuperTypen für diese Fälle vorgesehen sind.
2. *der Typ*, der eine fortlaufende Numerierung aller Verfahren einer Basisklasse darstellt.

Auf Grund ihres Abstraktionsgrades hängen die Basisklassen in gleichem Maße wie die aus der theoretischen Kryptographie stammenden Konzepte voneinander ab: z.B. benutzen Verschlüsselungs- und Signaturverfahren Pseudozufallszahlengeneratoren zur Er-

zeugung geheimer Schlüssel. Verschlüsselungsverfahren im Stil des klassischen One-Time-Pads werden aus Pseudozufallszahlengeneratoren konstruiert. Umgekehrt kann man Pseudozufallszahlengeneratoren aus bestimmten Verschlüsselungsalgorithmen zusammensetzen. Im folgenden Abschnitt werden wir uns auf die Beschreibung der Basisklasse `RandomBase` für Pseudozufallszahlengeneratoren (siehe auch Anhang) beschränken.

### 4.3 Beschreibung der Klasse `RandomBase`

Die Klasse `RandomBase` ist die Basisklasse für Pseudozufallszahlengeneratoren. In ihr werden allgemeine Eigenschaften und Funktionen beschrieben, die dann an die Klassen der einzelnen Pseudozufallszahlengeneratoren weitervererbt oder dort erst dem jeweiligen Pseudozufallszahlengenerator entsprechend ausgearbeitet und ausprogrammiert werden. Wir stellen in der Realisierung der Klasse `RandomBase` mehrere Möglichkeiten zur Objektinstanziierung zur Verfügung. Zunächst besteht die Möglichkeit, ein Objekt alleine durch Angabe des SuperTypes eines `Seedgenerators` (siehe Kapitel 5), die per Referenz übergeben wird, zu erzeugen. Eine weitere Möglichkeit, ein Objekt der Klasse `RandomBase` zu erzeugen, besteht in der direkten Initialisierung mit Angabe des SuperTypes und des Types. Dies geschieht in der Funktion `New`.

Jeder Basisklasse werden objektunabhängige Informationen zugeordnet, die sich über entsprechende Methoden abfragen lassen. Auch die Basisklasse `RandomBase` kann über die Funktion `Info` Informationen zu den einzelnen Typen liefern, d.h. Name des Programmiers, Name des `Random_type` oder eine Kurzbeschreibung. (Wir haben darauf verzichtet, diese Funktion jeweils in den einzelnen Pseudozufallszahlengeneratoren auszuführen, da wir sonst zur Abfrage der Informationen zunächst immer den entsprechenden Generator erzeugen müßten. Unsere Realisierung erlaubt es aber, Informationen abzufragen, ohne diesen Umweg gehen zu müssen.) Eine weitere objektunabhängige Abfrage der Basisklasse ist die Funktion `ShortInfo`, dabei wird nur der Name des `Random_type` wiedergegeben. Auch hier wird bei der Abfrage dieser Information der Generator nicht erzeugt. Die Property-Funktion der Basisklasse `RandomBase` gibt Auskunft, ob ein `Random_type` zur Verschlüsselung geeignet ist. Auch diese Anfrage haben wir global und nicht in jedem einzelnen Pseudozufallszahlengenerator implementiert. Da wir auch Pseudozufallszahlengeneratoren aus Verschlüsselungsverfahren konstruieren (siehe [Sch96b]), rufen wir bei Eingabe des SuperTypes `CryptAsRandom_type` die Property-Funktion der Basisklasse `CryptBase` auf.

Die `MakeNewSeed`-Funktion wird für die Initialisierung der jeweiligen Pseudozufallszahlengeneratoren benötigt. Daher wird die Funktion bei der Ausarbeitung der Pseudozufallszahlengeneratoren überladen und dort jeweils spezifiziert. Die Funktion `SetRandomSeed` liefert an die `Randombase` ein `ReadyFlag` zurück. Dies gibt den Zustand des Generators an. Die Generierung der Seed könnte fehlerhaft ausgeführt worden sein, z.B. könnte

sie die falsche Länge haben. Die Funktion bewirkt weiter, daß eine „zufällige“ Seed an die Pseudozufallszahlengeneratoren weitergegeben wird und diese damit initialisiert werden.

Mit Hilfe der Funktion `Fill` füllt ein Objekt der Klasse `RandomBase` einen Speicherbereich vorgegebener Größe auf mit Ausgaben, die ihm die Funktion `GetUChar()` liefert. Ferner gibt sie einen Pointer auf diesen Speicherbereich zurück. Die Funktion `GetUChar()` wird in der Basisklasse `RandomBase` (vgl. Header-File `randombase.h`) zunächst nur durch ihren Namen eingeführt und erst bei den einzelnen Pseudozufallszahlengeneratoren explizit ausprogrammiert. Diese Funktion stellt den „eigentlichen Pseudozufallszahlengenerator“ dar, d.h. in ihr werden jeweils acht Pseudozufallsbits gemäß der Spezifikation des entsprechenden Pseudozufallszahlengenerators erzeugt.

In dem Header-File `randombase.h` der Klasse `randombase.c` erfolgt die eigentliche Definition der Basisklasse `RandomBase`. Jeder Pseudozufallszahlengenerator benötigt als Eingabe Werte, die ihm ein Seedgenerator liefert. Dieser Seedgenerator sollte eine echte Zufallsquelle sein (siehe Kapitel 5). Wir erlauben es jedoch auch, Ausgaben eines weiteren Pseudozufallszahlengenerators als Eingabe für den aktuell betrachteten zu verwenden. Allerdings muß auch dieser Eingaben von einem Seedgenerator oder einem Pseudozufallszahlengenerator erhalten. Schließlich muß das letzte Glied dieser Kette nun wirklich Eingaben aus dem Seedgenerator, d.h. einem „echten Zufallszahlengenerator“ verwenden.

# Kapitel 5

## Der Seedgenerator

Ein Pseudozufallszahlengenerator erhält einen „zufälligen“ String als Eingabe und gibt einen längeren String aus, der für polynomielle Turingmaschinen immer noch „zufällig wirkt“ (siehe Kapitel 3). Im folgenden Kapitel beschäftigen wir uns mit der Frage, wie man diese „zufällige“ Eingabe des Pseudozufallszahlengenerators erhalten kann.

### 5.1 Grundsätzliche Vorbemerkungen

Zufälligkeit ist nur eine Eigenschaft eines abstrakten Modells. Ob ein solches Modell tatsächlich eine genaue Beschreibung der Realität darstellt, ist eine philosophische Frage, die im Zusammenhang mit der Frage gesehen werden muß, ob alle Gesetze, denen das Universum gehorcht, deterministisch sind oder nicht. Es erscheint unmöglich, diese Frage zu jedermanns Zufriedenheit zu beantworten. Allerdings existieren in der Natur Prozesse, wie z.B. der Zerfall radioaktiver Teilchen oder das „Rauschen“ in Transistoren oder die Form von Luftturbulenzen, deren exakter Verlauf (bisher) nicht vorhersagbar ist. Solche Prozesse erlauben die technische Realisierung einer Quelle „echter Zufälligkeit“, auch *echter Zufallsgenerator* oder *Seedgenerator* genannt, dessen Ausgaben (nach heutigem Wissen) „zufällig“ sind, bei dessen Verwendung man jedoch das Risiko eingeht, daß seine Ausgaben in Zukunft doch präzise deterministisch berechnet werden können.

Die technische Realisierung eines Zufallsgenerators (Seedgenerator) ist sehr aufwendig und schwierig und erfolgt meist unter Verwendung spezieller Hardware (siehe z.B. [Agn88, AT&T86]). Als wichtige Punkte im Anforderungskatalog an **LiSA** werden jedoch ein möglichst hoher Portabilitätsgrad und eine möglichst einfache Installation genannt. Ein für **LiSA** geeigneter Seedgenerator sollte also nur unter Verwendung von Standardkomponenten gängiger Rechnersysteme realisiert werden.

In [DIF94] stellen D. Davis, R. Ithaka und P. Fenstermacher einen Zufallsgenerator vor, der auf den Auswirkungen von Luftturbulenzen in Diskettenlaufwerken eines Computers

beruht. Dabei wird die Zeit gemessen, die zum Lesen eines bestimmten, festen Datenblocks auf einer Diskette benötigt wird. Durch die bei der Bewegung der Diskettenoberfläche auftretenden Luftbewegungen variiert die bei jedem neuen Versuch gemessene Zeit um mehrere Mikrosekunden. Diese Zeitdifferenz kann (zur Zeit) nicht deterministisch vorhergesagt und daher zur Konstruktion eines Seedgenerators verwendet werden. Leider läßt sich dieser Zufallsgenerator nur auf Rechnersystemen/Betriebssystemen realisieren, die einerseits sehr genaue Funktionen zur Zeitmessung (Genauigkeit im Mikrosekundenbereich) bereitstellen und andererseits eine Echtzeitmessung von Schreib-/Lesevorgängen auf Ausgabegeräten erlauben. Da uns kein solches Rechnersystem zur Verfügung stand, konnte ohne einen Eingriff in das Betriebssystem, der wiederum der geforderten Portabilität von **LiSA** zuwiderlaufen würde, dieser Zufallsgenerator nicht implementiert werden. Eine etwas speziellere Methode, die von der Rechnerumgebung abhängt, kann auf Rechnern implementiert werden, die an ein größeres Rechnernetz angeschlossen sind. Man nimmt an, daß bei einer durchschnittlichen Belastung des Netzes die Netzlast wieder Zahlenwerte liefert, deren niederwertigsten Bits als Zufallszahlen benutzt werden können. Allerdings führt dieses Verfahren im Falle einer sehr geringen Netzlast zu sehr „schlechten“ Zufallszahlen, ist also im allgemeinen nicht verwendbar. Umgekehrt liegt nun die Idee nahe, durch Nutzung von Systemfunktionen des Betriebssystems die Netzlast auf eine nicht vorhersagbare Weise zu erhöhen. Ein solches Verfahren ist jedoch neben dem Grund der geringen Geschwindigkeit alleine schon aus Gründen der Netzbelastung abzulehnen (siehe [LMS93]).

Eine weitere in der Literatur genannte Möglichkeit, an Zufallszahlen ohne Verwendung spezieller Zusatzhardware zu gelangen, besteht darin, die Längen der Zeitintervalle zwischen verschiedenen Tastaturanschlägen eines Benutzers zu messen, und jeweils das niederwertigste Bit als Zufallsbit aufzufassen (siehe [Zim95]). Aber dieses Verfahren funktioniert auf solchen Rechner- bzw. Betriebssystemen nicht, bei denen die Tastatureingabe zwischengespeichert oder gefiltert wird, bevor sie an ein Nutzerprogramm weitergereicht wird (dies passiert in manchen UNIX-Dialekten) und erfordert außerdem die Anwesenheit einer Person, die tatsächlich die Tastatur bedient.

## 5.2 Die Standardseedgenerator in **LiSA**

Der letztendlich in **LiSA** favorisierte Seedgenerator beruht auf einer Variante eines in [LMS93] vorgestellten Zufallsgenerators, der in ähnlicher Form auf den meisten Rechnersystemen mit einem Multitasking-Betriebssystem realisiert werden kann. Die grundsätzliche Arbeitsweise dieses Seedgenerators läßt sich wie folgt beschreiben: Zur Erzeugung eines Zufallsbits wird ein Prozeß gestartet, der einen Zähler in einer einfachen Schleife schrittweise erhöht. Nach einer festen Zeitschranke (in unserer Implementierung 10 Millisekunden) wird der Prozeß beendet und der Zähler ausgelesen. Das niederwertigste Bit des Zählerinhaltes ist nun unser aktuelles Zufallsbit. Tatsächlich scheint dieses

Verfahren „gute“ Zufallszahlen zu liefern. Dies liegt daran, daß der Zählerinhalt von der Zeit abhängig ist, die dem gestarteten Prozeß tatsächlich zum Zählen zur Verfügung stehen. Da das Multitasking-Betriebssystem allerdings auch Rechenzeit an andere auf dem Rechnersystem laufende Prozesse vergeben muß und diese Vergabe von der aktuellen Systemauslastung abhängt, variiert diese Zeit erheblich und ist kaum vorherzusagen.

Von seiner Programmstruktur her können wir den Seedgenerator wie unsere bereits oben beschriebenen Pseudozufallszahlengeneratoren behandeln und von der Klasse `randbase` ableiten. In diesem Sinn stellen wir den Seedgenerator als Pseudozufallszahlengenerator dar, der keine Eingabe aus einem Seedgenerator erhält. Die Realisierung des Seedgenerators findet sich in den Dateien `timerand.h` und `timerand.c` (siehe Anhang A).

Wie auch alle von uns implementierten Pseudozufallszahlengeneratoren bündelt der Seedgenerator die von ihm gewonnenen Bits byte-weise in der Funktion `GetUChar()`. Die eigentliche Erzeugung der Zufallsbits findet in `GetBit()` statt. Die Funktion `SetUserSeed` wird zur Initialisierung des Timers (10 Millisekunden) benutzt.

## 5.3 Statistische Tests

Wie bereits in Abschnitt 5.1 ausgeführt, ist es durchaus möglich, daß ein natürlicher Prozeß uns nur deshalb „zufällig“ erscheint, weil wir noch nicht genug über ihn wissen, um erkennen zu können, daß er in Wahrheit deterministisch beschreibbar ist. Um dieses Risiko beim Einsatz eines Seedgenerators möglichst gering zu halten, versucht man mittels „geeigneter“ *Tests* die „Zufälligkeit“ der Ausgaben eines Seedgenerators zu überprüfen. Diese Tests erhalten allerdings als Eingabe nur eine Ausgabe des Seedgenerators endlicher Länge. Selbst wenn ein Test eine solche Eingabe akzeptiert und „für zufällig erklärt“, bedeutet dies nicht, daß der Seedgenerators wirklich zufällige Ausgaben produziert. Tatsächlich müßten alle denkbaren Tests alle (bis auf vernachlässigbar viele) Ausgaben hinreichender Länge akzeptieren, um letztendliche Sicherheit zu garantieren. Da dies natürlich nicht in endlicher Zeit möglich ist, können Tests immer nur ein Indiz für die Qualität eines Seedgenerators sein. Für die Beschreibung der mathematischen Grundlagen und Sicherheitsangaben der von uns implementierten und verwendeten Tests verweisen wir auf [Knu81] und [Mau90]. Wir beschränken uns im folgenden nur auf eine kurze Beschreibung der Tests sowie der Ergebnisse in Bezug auf den von uns eingesetzten Seedgenerator.

Der in diesem Zusammenhang wohl bekannteste und am häufigsten eingesetzte Test ist der sogenannte *Chi-Quadrat-Test* (auch  $\chi^2$ -Test). Er erhält als Eingabe  $x$  einen Bitstring der Länge  $n$ , der aus Teilstrings  $x_i \in \{0, 1\}^\ell$ ,  $\ell \in \mathbb{N}$ ,  $\ell < n$ ,  $1 \leq i \leq \lfloor n/\ell \rfloor$ , der Länge  $\ell$  besteht. Der Test „versucht nun zu messen“, ob die Strings  $x_i$  gleichverteilt in  $\{0, 1\}^\ell$  sind. Dazu bestimmt er für jeden String  $s \in \{0, 1\}^\ell$  die Anzahl  $Y_s$  der Vorkommen von  $s$  unter allen  $x_i$ , d.h.

$$Y_s = |\{i: 1 \leq i \leq \lfloor n/\ell \rfloor, x_i = s\}|.$$

Das Quadrat der gewichteten Differenz zwischen  $Y_s$  und der erwarteten Häufigkeit (gemäß seiner Wahrscheinlichkeit bei Gleichverteilung) von  $s$  wird nun über alle betrachteten Strings  $s$  aufsummiert. Wir erhalten

$$V = \sum_{s \in \{0,1\}^\ell} \frac{(Y_s - n2^{-\ell})^2}{n2^{-\ell}}.$$

Dieser Wert  $V$  liefert nicht unbedingt ein Maß dafür, inwiefern die  $x_i$  gleichverteilt sind. Für den Fall, daß die  $x_i$  wirklich gleichverteilt sind, kann man jedoch umgekehrt Intervalle und Wahrscheinlichkeiten dafür angeben, daß  $V$  keinen Wert innerhalb dieser Intervalle annimmt. Eine entsprechende Tabelle bzw. Formeln für obere und untere Schranken von  $V$  bei gegebener Wahrscheinlichkeit finden sich z.B. in [Knu81, S. 41] (wobei wir dort  $\nu = \ell$  setzen können). Der Test akzeptiert, wenn der von ihm berechnete Wert von  $V$  innerhalb der Intervallschranken liegt, andernfalls lehnt er ab.

Wir haben eine Datei der Größe 650 Kbyte mit den Ausgaben unseres Seedgenerators gefüllt und mit dem Chi-Quadrat-Test für  $\ell = 1, \dots, 16$  getestet. Dabei sei  $p$  die Wahrscheinlichkeit dafür, daß der aktuelle Wert von  $V$  nicht in den vorgegebenen Intervallgrenzen liegt.

Wir erhielten folgendes Resultat:

Bitlänge	Wahrscheinlichkeit $p$	$V$	Akzeptanz
1	0.01	2.6049	akzeptiert
1	0.05	2.6049	akzeptiert
1	0.10	2.6049	akzeptiert
2	0.01	2.74649	akzeptiert
2	0.05	2.74649	akzeptiert
2	0.10	2.74649	akzeptiert
3	0.01	6.56874	akzeptiert
3	0.05	6.56874	akzeptiert
3	0.10	6.56874	akzeptiert
4	0.01	16.5221	akzeptiert
4	0.05	16.5221	akzeptiert
4	0.10	16.5221	akzeptiert
5	0.01	19.802	akzeptiert
5	0.05	19.802	akzeptiert
5	0.10	19.802	akzeptiert

Bitlänge	Wahrscheinlichkeit $p$	$V$	Akzeptanz
6	0.01	63.8082	akzeptiert
6	0.05	63.8082	akzeptiert
6	0.10	63.8082	akzeptiert
7	0.01	119.446	akzeptiert
7	0.05	119.446	akzeptiert
7	0.10	119.446	akzeptiert
8	0.01	267.194	akzeptiert
8	0.05	267.194	akzeptiert
8	0.10	267.194	akzeptiert
9	0.01	508.781	akzeptiert
9	0.05	508.781	akzeptiert
9	0.10	508.781	akzeptiert
10	0.01	959.508	akzeptiert
10	0.05	959.508	akzeptiert
10	0.10	959.508	akzeptiert
11	0.01	2097.62	akzeptiert
11	0.05	2097.62	akzeptiert
11	0.10	2097.62	akzeptiert
12	0.01	4194.5	akzeptiert
12	0.05	4194.5	akzeptiert
12	0.10	4194.5	akzeptiert
13	0.01	8104.13	akzeptiert
13	0.05	8104.13	akzeptiert
13	0.10	8104.13	akzeptiert
14	0.01	16485.1	akzeptiert
14	0.05	16485.1	akzeptiert
14	0.10	16485.1	akzeptiert
15	0.01	32371.8	akzeptiert
15	0.05	32371.8	akzeptiert
15	0.10	32371.8	lehnt ab
16	0.01	65658.5	akzeptiert
16	0.05	65658.5	akzeptiert
16	0.10	65658.5	akzeptiert

Ein wesentlich aktuellerer und um vieles allgemeinerer Test wird in [Mau90] vorgestellt. Die diesem, im folgenden von uns *Maurer-Test* genannten Testverfahren zugrundeliegende Idee wurde bereits in [Ziv90] vorgeschlagen. Sie besteht in der Annahme, daß die Ausgabe eines Seedgenerators „zufällig“ ist, wenn sie durch keinen Algorithmus komprimiert werden kann. Natürlich läßt sich dies nicht in endlicher Zeit überprüfen. Wir können jedoch zu jeder endlichen Ausgabefolge des Seedgenerators einen Wert bestimmen, der

ein Maß für den Kompressionsgrad eines festgewählten Komprimierungsverfahrens ist. In [Mau90] wird dazu ein Kompressionsverfahren von Elias [Eli87] und Willems [Wil89] vorgeschlagen und untersucht. Nach [Mau90] entdeckt dieser Test jede signifikante Abweichung der Ausgabeverteilung unseres Seedgenerators von der eines echten Zufallsgenerators mit hoher Wahrscheinlichkeit, falls unser Seedgenerator als „zeitunabhängige Quelle mit endlichem Speicher“ modelliert werden kann. Insbesondere ist nach [Mau90] dieser Test allgemeiner als die in [Knu81] vorgestellten Tests, zu denen neben dem Chi-Quadrat-Test auch der *Run-Test*, der *Autokorrelationstest*, der *Poker-Test* usw. gehören. Im wesentlichen mißt das Verfahren die Abstände zwischen verschiedenen Vorkommen des gleichen Strings einer vorgegebenen Länge  $L \in \mathbb{N}$  (als Teilstring der Ausgabe unseres Seedgenerators), gewichtet diese Abstände logarithmisch und berechnet ihre gewichtete Summe  $FT[L]$ . Bezeichne nun  $s \in \{0, 1\}^*$  die Eingabe des Tests, und sei o.B.d.A.  $|s| = (Q + K)L$  mit  $Q = 5 \cdot 2^L$  und  $K = \left(\frac{|s|}{L} - Q\right) \in \mathbb{N}$ . Für  $0 \leq n \leq Q + K - 1$  bezeichne  $b_n(s)$  den  $n$ -ten Teilstring von  $s$  der Länge  $L$ . Dann wird in [Mau90]  $FT[L]$  nach folgendem Algorithmus „definiert“:

### 5.1. Algorithmus

Algorithmus zur Berechnung der  $FT[L]$ -Werte

EINGABE:  $s \in \{0, 1\}^*$ ,  $L \in \mathbb{N}$ ,  $L$  teilt  $|s|$  und  $|s| = (5 \cdot 2^L + K)L$  für ein  $k \in \mathbb{N}$ .  
 AUSGABE:  $FT[L]$

```

(1) for ( $i := 0$  to  $2^L - 1$ ) do
(2)    $Tab[i] := 0$ 
(3) od
(4) for ( $n := 0$  to  $Q - 1$ ) do
(5)    $Tab[b_n(s)] := n$ 
(6) od
(7)  $sum := 0.0$ 
(8) for ( $n := Q$  to  $Q + K - 1$ ) do
(9)    $sum := sum + \log_2(n - Tab[b_n(s)])$ 
(10)   $Tab[b_n(s)] := n$ 
(11) od
(12)  $FT[L] := sum / K$ 

```

(Dabei werden in [Mau90] für  $K$  der Wert  $K = 10^4$  und für  $L$  Werte von eins bis 16 empfohlen, um die Laufzeit der Tests in einem erträglichen Rahmen zu halten.) Der so

berechnete Wert  $FT[L]$  kann (analog zum obigen Test) mit einer unteren bzw. oberen Schranke verglichen werden. Für eine genauere Beschreibung des Tests verweisen wir an dieser Stelle auf [Mau90]. Testen wir unsere oben erwähnte 650 Kbyte große Datei, so erhalten wir folgende Daten: (Dabei bezeichne  $t1[l]$  und  $t2[L]$  die obere bzw. untere Schranke von  $FT[L]$ .)

L	FT[L]	Wert t1[L]	Wert t2[L]	Akzeptanz
1	0.732539	0.731671	0.733628	akzeptiert
2	1.53724	1.53544	1.53944	akzeptiert
3	2.40118	2.39858	2.40464	akzeptiert
4	3.31146	3.30716	3.31528	akzeptiert
5	4.25174	4.24834	4.25851	akzeptiert
6	5.21743	5.21159	5.22382	akzeptiert
7	6.19727	6.18907	6.20343	akzeptiert
8	7.18392	7.17536	7.19197	akzeptiert
9	8.17378	8.16687	8.18598	akzeptiert
10	9.17345	9.16132	9.18333	akzeptiert
11	10.1674	10.1572	10.1828	akzeptiert
12	11.163	11.1536	11.1839	akzeptiert
13	12.1616	12.1495	12.1866	akzeptiert
14	13.169	13.1432	13.1921	akzeptiert
15	14.1552	14.1267	14.2083	akzeptiert
16	0	10.3954	19.9393	lehnt ab

Der Test akzeptiert für alle vorgegebenen Längen bis 15 Bits. Die Ausgabe für  $L = 16$  läßt darauf schließen, daß die Probedatei zu kurz war (siehe [Mau90]).

Wir gehen daher im folgenden davon aus, daß unser Seedgenerator für unsere kryptographischen Anwendungen geeignet ist (obwohl dies, wie oben erwähnt, selbst nach unseren positiven Testergebnissen nicht im eigentlichen Sinn bewiesen ist!).

# Kapitel 6

## Implementierte Pseudozufallszahlengeneratoren

Die meisten Pseudozufallszahlengeneratoren benötigen für ihre Berechnungen Zahlen, die den von Rechnern oder Programmiersprachen wie C++ standardmäßig zur Verfügung gestellten Größenbereich weit überschreiten. Daher muß man bei ihrer Implementierung auf eine **Langzahlarithmetik** zurückgreifen, d.h. eine Bibliothek, die Datentypen und Operationen für solche langen Zahlen bereitstellt. Über die Verwendung langer Zahlen hinaus benötigen wir auch kompliziertere Strukturen, wie z.B. modulare Arithmetiken oder elliptische Kurven. Daher erhielten wir die Vorgabe, die C++-Klassenbibliothek **LiDIA** zu benutzen (siehe [BBP95].) **LiDIA** ist eine objektorientierte Softwarebibliothek, die über eine Langzahlarithmetik und viele weitere Objekte zur Realisierung mathematischer Strukturen verfügt. Sie wurde im Fachbereich 14 – Informatik an der Universität des Saarlandes am Lehrstuhl Prof. Buchmann entwickelt. Da sie anschaulich und leicht portierbar ist und einen intuitiven Quelltext ermöglicht, ist **LiDIA** zur Implementierung von Pseudozufallszahlengeneratoren besonders geeignet. Auch wenn **LiDIA** schneller ist als vergleichbare Softwarebibliotheken, könnten einige Routinen (speziell im Bereich elliptischer Kurven) für die Verwendung in unserer Bibliothek verbessert werden.

In Abschnitt 3.3 haben wir die in dieser Arbeit programmierten Generatoren vorgestellt und in Zusammenhang mit den für die Pseudozufälligkeit notwendigen Annahmen aus Abschnitt 3.2 gebracht. Ferner haben wir bereits in Abschnitt 4.3 die allgemeine Struktur der Implementierungen der Pseudozufallszahlengeneratoren beschrieben. In den folgenden Abschnitten müssen wir daher nur noch auf Besonderheiten der jeweiligen Programme eingehen. Der C++-Quelltext findet sich dazu in Anhang A. Wir verwenden die in den genannten Abschnitten benutzten Schreibweisen und Bezeichnungen.

## 6.1 Der Blum-Blum-Shub *BBS* Generator

In jedem Sourcecode von Pseudozufallszahlengeneratoren wird zunächst der Sicherheitsparameter `SecLevel` spezifiziert (siehe Abschnitt 4.3). Beim Generator von Blum, Blum und Shub wird die Sicherheit im wesentlichen durch die Länge der verwendeten Primzahlen  $p_1$  und  $p_2$  beeinflusst. Je größer diese Zahlen sind, desto sicherer ist der Pseudozufallszahlengenerator, desto langsamer ist er aber auch. Als sinnvollen Kompromiß zwischen Sicherheit und Laufzeit stellen wir bei allen Pseudozufallszahlengeneratoren drei Sicherheitslevel bereit. Bei dem Generator nach Blum-Blum-Shub erhält `SecLevel` den Parameterwert  $-1$ , dann stellen wir für die Darstellung der Primzahlen 16 Bytes zur Verfügung. Hat `SecLevel` den Wert  $0$  bzw.  $1$ , dann sind die Primzahlen auf 32 bzw. 64 Bytes beschränkt.

Die Funktion `MakeNewSeed` muß gemäß der Beschreibung des Generators in Abschnitt 3.3.1 zwei zufällige Primzahlen, die jeweils kongruent 3 modulo 4 sein müssen, erzeugen. Wir haben zur Generierung dieser Primzahlen die Funktion `RandomBlumPrime` implementiert, die bei Eingabe einer positiven ganzen Zahl solange alle größeren Primzahlen mit Hilfe der `LiDIA`-Funktion `next_prime` aufzählt, bis eine Primzahl, die die Kongruenzbedingung erfüllt, gefunden wurde. Nachdem der Modul  $n$  als Produkt aus  $p$  und  $q$  berechnet wurde und eine zufällige Zahl  $x$  der Länge  $n$  erzeugt wurde, werden diese in dem Objekt `NewSeed` eingetragen.

## 6.2 Der RSA-Bit-Generator

Auch hier wird zunächst der Sicherheitsparameter `SecLevel` spezifiziert. Dabei wird hier mit dem Parameterwert  $-1$  die Länge der Primzahlen  $p$  und  $q$  auf 16 Bytes, dem Wert  $0$  die Länge auf 32 Bytes oder dem Parameterwert  $1$  die Länge zu Berechnung auf 64 Bytes festgelegt. Die zufälligen Primzahlen bei diesem Generator benötigen keine besondere Eigenschaft. Auch hier erfolgt die Ausgabe der Pseudozufallsbits byteweise.

## 6.3 Der modifizierte Rabin-Bit-Generator

Der Sicherheitsparameter `SecLevel` wird wie in der Implementierung des Generators von Blum, Blum und Shub (siehe Abschnitt 3.3.1) behandelt. Ferner muß auch hier ein Paar zufälliger Primzahlen gefunden werden, die jeweils kongruent 3 modulo 4 sein müssen. Dies geschieht ebenfalls analog zum Generator von Blum, Blum und Shub.

## 6.4 Der diskrete Exponentialgenerator I

Gemäß der Beschreibung dieses Generators (siehe Abschnitt 3.3.4) muß man eine zufällige Primzahl  $p$  und dann einen Erzeuger  $g$  von  $\mathbb{F}_p^*$  bestimmen. Das in [Coh93] vorgestellte Verfahren zur Bestimmung eines solchen Erzeugers benutzt die Tatsache, daß die Primfaktorzerlegung von  $p-1$  bereits bekannt ist. Im allgemeinen ist aber die Berechnung der Primfaktorzerlegung einer ganzen Zahl ein sehr schweres Problem, für das bisher kein Verfahren mit polynomieller Laufzeit bekannt ist. Wir müssen also einen Weg finden, eine Primzahl so zu erzeugen, daß wir direkt die Primfaktorzerlegung von  $p-1$  kennen. Algorithmen, die dieses leisten, wurden zunächst von Bach (siehe [Bac83]) und später ausführlicher von Maurer (siehe [Mau95]) beschrieben. Wir verwenden eine vereinfachte Version des von Maurer vorgestellten Algorithmus `FastPrime` (siehe [Mau95, S. 135]). Die zugrundeliegende Idee besteht darin, daß man mehrere zufällige Primzahlen wählt und testet, ob ihr Produkt vergrößert um eins eine Primzahl ergibt. Der diskrete Exponentialgenerator I geht dabei wie folgt vor: er erzeugt eine Primzahl einer vorgegebenen Bitlänge  $\ell \in \mathbb{N}$ , indem er zunächst (mit Hilfe des Seedgenerators) zufällige Zahlen  $\ell_1, \dots, \ell_i \in \mathbb{N}$  bestimmt, deren Summe  $\ell$  ergibt. Für jedes  $\ell_j$ ,  $1 \leq j \leq i$ , sucht er nun mit Hilfe der von `LiDIA` bereitgestellten Funktion `nextprime` eine Primzahl  $p_j$  der Bitlänge  $\ell_j$ . Danach testet er, ob  $\prod_{j=1}^i p_j + 1$  eine Primzahl ist. Ist dies nicht der Fall, so löscht er aus der Liste der  $p_j$  zufällig (entsprechend des Seedgenerators) einige, ersetzt die entsprechenden Werte  $\ell_j$  durch neue zufällig bestimmte, die die gleiche Summe wie die ursprünglichen ergeben, bestimmt neue zufällige Primzahlen der Längen  $\ell_j$  und beginnt von vorne. Wie in [Mau95] gezeigt, ist die erwartete Laufzeit dieses Verfahrens polynomiell. Außerdem sind nach [Mau95] die so gewonnenen Primzahlen fast gleichverteilt. Der Sicherheitsparameter legt beim Wert -1 die Länge der Primzahlen auf 16 Bytes, beim Wert 0 auf 32 und beim Wert 1 auf 64 Bytes fest.

## 6.5 Der diskrete Exponentialgenerator II

Der diskrete Exponentialgenerator II bestimmt zunächst eine zufällige Primzahl  $p$  und dann ein Element von  $\mathbb{F}_p^*$  von möglichst großer Ordnung. Wir gehen dabei wie folgt vor: wir bestimmen eine Primzahl  $p$ , so daß  $p-1$  aus zwei großen Primfaktoren und einem kleineren Faktor besteht. Dazu raten wir zwei große zufällige Primzahlen  $q_1$  und  $q_2$  und eine kleinere Zahl  $q_0$  und testen, ob  $q_0 q_1 q_2 + 1$  eine Primzahl ist. Ist dies nicht der Fall, beginnen wir von vorne. Wie in [Mau95] kann man zeigen, daß das Verfahren erwartete polynomielle Laufzeit hat.

Nun suchen wir mit dem in [Coh93] beschriebenen Verfahren ein Element  $g \in \mathbb{F}_p^*$ , dessen Ordnung wenigstens  $\min\{q_1, q_2\}$  ist. Danach arbeitet der diskrete Exponentialgenerator II genau wie der diskrete Exponentialgenerator I (siehe auch Abschnitt 3.3.4).

## 6.6 Der Generator auf elliptischen Kurven I

Der Generator auf elliptischen Kurven ähnelt sehr dem diskreten Exponentialgenerator II. Wir bestimmen eine zufällige Primzahl  $p$ , eine zufällige supersinguläre elliptische Kurve über  $\mathbb{F}_p$  der Form  $y^2 = x^3 + b$  mit  $b \in \mathbb{F}_p$  und einen Punkt  $g$  auf dieser Kurve von hoher Ordnung. Dazu nutzen wir den Seedgenerator zur Wahl eines zufälligen Paares  $(x, y) \in \mathbb{F}_p \times \mathbb{F}_p$ , berechnen  $b$ , testen, ob die genannte Form eine elliptische Kurve ist, und setzen dann  $g = (x, y)$ . Zur Bestimmung der Ordnung von  $g$  benutzen wir ein Verfahren, das analog zur Bestimmung der Ordnung eines Elementes aus  $\mathbb{F}_p$  (beim diskreten Exponentialgenerator II) ist. Auch hier muß man zunächst eine Primzahl  $p$  bestimmen, so daß man eine Faktorisierung der Ordnung der betrachteten Gruppe (hier der Punktegruppe der supersingulären elliptischen Kurven über  $\mathbb{F}_p$ ) kennt. Diese Ordnung ist hier gerade durch  $p+1$  gegeben (siehe [Sil86]). Zum Erzeugen dieser Primzahlen benutzen wir ein ähnliches Verfahren wie beim diskreten Exponentialgenerator II. (Der einzige Unterschied besteht darin, daß dort die betrachtete multiplikative Gruppe  $\mathbb{F}_p^*$  die Ordnung  $p-1$  hat.)

# Kapitel 7

## Zusammenfassung – Ergebnisse

Abschließend wollen wir einige Laufzeitergebnisse von unseren Pseudozufallszahlengeneratoren angeben. Mit jedem Zufallszahlengenerator wurden auf einer SUN SPARC 20 zehn mal 100 und zehn mal 1000 Bytes erzeugt. Das Benutzerprogramm *gendata* benutzt **LiSA** zur Erzeugung von echten bzw. Pseudozufallszahlen. Dabei mißt *gendata* die Zeit, die zur Initialisierung des Pseudozufallszahlengenerators nötig ist und zum anderen die Zeit, die der Generator braucht, um „echte“ oder Pseudozufallszahlen einer bestimmten Größe zu berechnen. Um die Laufzeiten besser vergleichen zu können, wählen wir den Sicherheitslevel der Pseudozufallszahlengeneratoren so, daß sie nach ihrer Initialisierung jeweils Operationen auf Zahlen einer Länge von 512 Bits ausführen (Im Falle der Generatoren Blum-Blum-Shub, modifizierter Rabin-Bit und RSA-Bit verwenden wir als Sicherheitslevel 0, bei allen sonstigen Sicherheitslevel 1). Es zeigt sich, daß der modifizierte Rabin-Bit-Generator der schnellste der von uns beschriebenen und implementierten Generatoren ist, dicht gefolgt vom Generator nach Blum, Blum und Shub. Dies liegt daran, daß beide Generatoren pro Ausgabebit nur eine Subtraktion oder eine Quadrierung benötigen, während alle anderen Generatoren aufwendige Exponentiationen oder längere Folgen von Additionen und Multiplikationen ausführen müssen. Überraschend schlecht schneidet der Generator auf elliptischen Kurven ab. Dies ist wohl zum größten Teil auf die zum jetzigen Zeitpunkt noch sehr ineffiziente Implementierung der elliptischen Kurven in **LiDIA** zurückzuführen.

Ein ähnliches Bild zeichnet sich bei der Betrachtung der Initialisierungszeiten ab. Hier führt der Generator nach Blum, Blum, Shub vor dem modifizierten Rabin-Bit-Generator. Dieser Vorsprung wird natürlich bei der Erzeugung vieler Zufallsbits durch die schnellere Laufzeit des letztgenannten Generators wettgemacht.

## 7.1 Laufzeitergebnisse des Generators nach Blum, Blum und Shub

Testlauf zur Erzeugung von 100 Bytes		
Generator	Zeit zur Initialisierung [sec]	Sekunden pro Byte
BBS	15.05	0.0043
BBS	20.26	0.0047
BBS	12.44	0.0044
BBS	12.13	0.0044
BBS	16.69	0.0064
BBS	16.12	0.0043
BBS	17.95	0.0046
BBS	16.35	0.0046
BBS	12.48	0.0043
BBS	21.16	0.0064

Testlauf zur Erzeugung von 1000 Bytes		
Generator	Zeit zur Initialisierung [sec]	Sekunden pro Byte
BBS	13.96	0.00464
BBS	14.44	0.00437
BBS	14.82	0.00643
BBS	14.55	0.0046
BBS	18.74	0.00418
BBS	18.59	0.00426
BBS	18.36	0.00434
BBS	12.48	0.0045
BBS	14.2	0.00645
BBS	14.41	0.00611

Die durchschnittliche Zeit zur Erzeugung eines Bytes bei dem Blum-Blum-Shub Generator beträgt 0.004914 Sekunden. Die durchschnittliche Initialisierungszeit beträgt 15.7595 Sekunden.

## 7.2 Laufzeitergebnisse des RSA-Bit-Generators

Testlauf zur Erzeugung von 100 Bytes		
Generator	Zeit zur Initialisierung [sec]	Sekunden pro Byte
BITRSA	51.29	0.1383
BITRSA	22.84	0.1033
BITRSA	18.85	0.113
BITRSA	37.04	0.1114
BITRSA	51.04	0.1075
BITRSA	28.99	0.1472
BITRSA	36.08	0.0967
BITRSA	29.57	0.1351
BITRSA	30.83	0.1376
BITRSA	36.75	0.1087

Testlauf zur Erzeugung von 1000 Bytes		
Generator	Zeit zur Initialisierung [sec]	Sekunden pro Byte
BITRSA	45.45	0.12118
BITRSA	18.4	0.14052
BITRSA	18.52	0.09171
BITRSA	24.33	0.11682
BITRSA	20.8	0.09098
BITRSA	18.76	0.09664
BITRSA	23	0.11882
BITRSA	26.26	0.10924
BITRSA	32.57	0.12498
BITRSA	59.44	0.12366

Die durchschnittliche Zeit zur Erzeugung eines Bytes beim Bit-RSA-Generator beträgt 0.11666 Sekunden. Die durchschnittliche Initialisierungszeit beträgt 31.5405 Sekunden.

### 7.3 Laufzeitergebnisse des modifizierten Rabin-Bit-Generators

Testlauf zur Erzeugung von 100 Bytes		
Generator	Zeit zur Initialisierung [sec]	Sekunden pro Byte
RABBIT	24.06	0037
RABBIT	12.23	0.0038
RABBIT	14.62	0.0037
RABBIT	15.73	0.0037
RABBIT	15.31	0.0022
RABBIT	15.85	0.0022
RABBIT	17.38	0.0037
RABBIT	14.3	0.0037
RABBIT	21.48	0.0038
RABBIT	16	0.0022

Testlauf zur Erzeugung von 1000 Bytes		
Generator	Zeit zur Initialisierung [sec]	Sekunden pro Byte
RABBIT	13.31	0.0037
RABBIT	17.98	0.0037
RABBIT	31.3	0.00368
RABBIT	17.1	0.00221
RABBIT	14.56	0.00369
RABBIT	19.19	0.0037
RABBIT	17.93	0.00221
RABBIT	16.44	0.0037
RABBIT	24.37	0.00221
RABBIT	15.92	0.00269

Die durchschnittliche Zeit zur Erzeugung eines Bytes beim modifizierten Rabin-Bit-Generator beträgt 0.0032095 Sekunden. Die durchschnittliche Initialisierungszeit beträgt 17.753 Sekunden.

## 7.4 Laufzeitergebnisse des diskreten Exponentialgenerators I

Testlauf zur Erzeugung von 100 Bytes		
Generator	Zeit zur Initialisierung [sec]	Sekunden pro Byte
DIEX.I	78.48	0.1108
DIEX.I	194.16	0.1112
DIEX.I	2940.2	0.1868
DIEX.I	3729.08	0.184
DIEX.I	1042.76	0.1316
DIEX.I	77.88	0.112
DIEX.I	578.76	0.148
DIEX.I	473.52	0.1848
DIEX.I	1469.28	0.1832
DIEX.I	367.4	0.1104

Testlauf zur Erzeugung von 1000 Bytes		
Generator	Zeit zur Initialisierung [sec]	Sekunden pro Byte
DIEX.I	746.08	0.18232
DIEX.I	644.2	0.18436
DIEX.I	3200.4	0.11016
DIEX.I	2245.04	0.1448
DIEX.I	149	0.12876
DIEX.I	546.32	0.11456
DIEX.I	135.56	0.15
DIEX.I	94.68	0.11368
DIEX.I	37.2	0.1634
DIEX.I	625.2	0.12688

Die durchschnittliche Zeit zur Erzeugung eines Bytes bei dem diskreten Exponentialgenerator I beträgt 0.1440084 Sekunden. Die durchschnittliche Initialisierungszeit beträgt 968.76 Sekunden.

## 7.5 Laufzeitergebnisse des diskreten Exponentialgenerator II

Testlauf zur Erzeugung von 100 Bytes		
Generator	Zeit zur Initialisierung [sec]	Sekunden pro Byte
DIEX_II	26.52	0.1408
DIEX_II	2476.12	0.1848
DIEX_II	1964.72	0.1388
DIEX_II	205.44	0.1256
DIEX_II	3001.24	0.144
DIEX_II	784.4	0.1836
DIEX_II	2585.4	0.1516
DIEX_II	3515.04	0.1416
DIEX_II	1061.76	0.1468
DIEX_II	1284.8	0.156

Testlauf zur Erzeugung von 1000 Bytes		
Generator	Zeit zur Initialisierung [sec]	Sekunden pro Byte
DIEX_II	176.44	0.14468
DIEX_II	2612.52	0.13924
DIEX_II	3147.96	0.1314
DIEX_II	338.56	0.13424
DIEX_II	53.68	0.14056
DIEX_II	1456.68	0.14148
DIEX_II	3742.36	0.17572
DIEX_II	5387.36	0.13008
DIEX_II	599.8	0.14308
DIEX_II	1819.64	0.14336

Die durchschnittliche Zeit zur Erzeugung eines Bytes bei dem diskreten Exponentialgenerator II beträgt 0.146872 Sekunden. Die durchschnittliche Initialisierungszeit beträgt 1812.022 Sekunden.

## 7.6 Laufzeitergebnisse des Generators auf elliptischen Kurven

Testlauf zur Erzeugung von 100 Bytes		
Generator	Zeit zur Initialisierung [sec]	Sekunden pro Byte
ELLI_I	300.49	9.1226
ELLI_I	166.75	9.0693
ELLI_I	154.55	9.4943
ELLI_I	129.67	9.3128
ELLI_I	556.13	8.9857
ELLI_I	422.77	9.072
ELLI_I	282.35	8.9793
ELLI_I	908.37	9.5893
ELLI_I	267.51	9.0626
ELLI_I	140.01	9.3394

Testlauf zur Erzeugung von 1000 Bytes		
Generator	Zeit zur Initialisierung [sec]	Sekunden pro Byte
ELLI_I	135.1	9.3585
ELLI_I	1077.25	9.35908
ELLI_I	271.7	9.3019
ELLI_I	190.32	9.46307
ELLI_I	185.69	9.37591
ELLI_I	791.6	9.41225
ELLI_I	153.48	9.32995
ELLI_I	155.74	9.34247
ELLI_I	376.38	9.37907
ELLI_I	164.01	9.37478

Die durchschnittliche Zeit zur Erzeugung eines Bytes bei dem Generator auf elliptischen Kurven beträgt 9.286214 Sekunden. Die durchschnittliche Initialisierungszeit beträgt 341.4935 Sekunden.

# Anhang A

## Implementierungen

### A.1 Der Generator nach Blum-Blum-Shub

```
// bbs.c
//
// 10/1995 Baerbel Mueller
//

#include "randombase.h"
#include "seedgen.h"
#include <LiDIA/bigint.h>
#include "big_defs.h"
#include "bbs.h"

int BBS_Random::MakeNewSeed(DATA &NewSeed, int SecLevel,
                           RandomBase *Rand) const
{
    bigint p,q,n,x;
    int l;

    l = (SecLevel<0)?16:((SecLevel==0)?32:64);
    // l is the length of the factors in bytes.

    RandomBlumPrime(l,p,Rand);
    RandomBlumPrime(l,q,Rand);

    n = p*q;
```

```

Random(2*1,x,Rand);
x %= n;

if (NewSeed.New(4*1+2))
{
    uchar *p = NewSeed.GetPtr();
    short2char(2*1,p);
    bigint2char(n,p,2*1);
    bigint2char(x,p,2*1);
    return 0;
}
return 1;
}

RandomBase::ReadyFlag BBS_Random::SetUserSeed(const MEM &Seed)
{
    size_t l;
    IsReadyFlag = NotReady_flag;
    if (Seed.GetSize()>=2)
    {
        uchar *p = Seed.GetPtr();
        l = char2short(p);
        // l is the length of n and x in bytes.
        if (Seed.GetSize()==2*1+2)
        {
            char2bigint(p+2,l,n);
            char2bigint(p+2+1,l,x);
            n_half = n>>1;
            IsReadyFlag = Ready_flag;
        }
    }
    return IsReadyFlag;
}

uchar BBS_Random::GetUChar()
{
    uchar c=0;
    int i,l,p;
    for (i=4;i--i)
    {
        square(x,x);
    }
}

```

```

    remainder(x,x,n);
    for (p=0,l=x.bit_length()-1;l>=0;--l)
    {
        if (x.bit(l)!=0)
            p++;
    }
    c<<= 1;
    c |= (p&1);
    c<<= 1;
    if (x>=n_half)
        c |= 1;
}
return c;
}

```

## A.2 Der RSA-Bit-Generator

```

// bitrsa.c
//
// 10/1995 Baerbel Mueller
//

#include "randombase.h"
#include "seedgen.h"
#include <LiDIA/bigint.h>
#include "big_defs.h"
#include "bitrsa.h"

int BITRSA_Random::MakeNewSeed(DATA &NewSeed, int SecLevel,
                               RandomBase *Rand) const
{
    bigint phi,e,p,q,n,x;
    int l;

    l = (SecLevel<0)?32:((SecLevel==0)?64:128);
    // l is the length of n in bytes.

    RandomPrime(l/2,p,Rand);
    RandomPrime(l/2,q,Rand);

```

```

n = p*q;
phi = (p-1)*(q-1);

do
{
    Random(1,e,Rand);
    e %= n-1;
    e = e+2;
}
while (!gcd(e,phi).is_one());

Random(1,x,Rand);
x %= n;

if (NewSeed.New(3*l+2))
{
    uchar *p = NewSeed.GetPtr();
    short2char(1,p);
    bigint2char(n,p,1);
    bigint2char(x,p,1);
    bigint2char(e,p,4);
    return 0;
}
return 1;
}

RandomBase::ReadyFlag BITRSA_Random::SetUserSeed(const MEM &Seed)
{
    size_t l;
    IsReadyFlag = NotReady_flag;
    if (Seed.GetSize()>=2)
    {
        uchar *p = Seed.GetPtr();
        l = char2short(p);
        // l is the length of n ,e and x in bytes.
        if (Seed.GetSize()==3*l+2)
        {
            char2bigint(p+2,l,n);
            char2bigint(p+2+l,l,x);
            char2bigint(p+2+2*l,l,e);

```

```

        IsReadyFlag = Ready_flag;
    }
}
return IsReadyFlag;
}

RandomBase::ReadyFlag BITRSA_Random::SetRandomSeed(int SecLevel)
{
    SEED Seed;
    IsReadyFlag = NotReady_flag;
    if (MakeNewSeed(Seed,SecLevel,0) == 0)
        SetUserSeed(Seed);
    return IsReadyFlag;
}

uchar BITRSA_Random::GetUChar()
{
    uchar c=0;
    int i;
    for (i=8;i--i)
    {
        power_mod(x,x,e,n);
        c<<=1;
        c |=x.is_odd();
    }
    return c;
}

```

### A.3 Der modifizierte Rabin-Bit-Generator

```

// rabbit.c
//
// 10/1995 Baerbel Mueller
//

#include "randombase.h"
#include "seedgen.h"
#include <LiDIA/bigint.h>
#include "big_defs.h"
#include "rabbit.h"

```

```

int RABBIT_Random::MakeNewSeed(DATA &NewSeed, int SecLevel,
                               RandomBase *Rand) const
{
    bigint p,q,n,x;
    int l;

    l = (SecLevel<0)?32:((SecLevel==0)?64:128);
    // l is the length of n in bytes.

    RandomBlumPrime(1/2,p,Rand);
    RandomBlumPrime(1/2,q,Rand);

    n = p*q;

    Random(1,x,Rand);
    x %= n;

    if (NewSeed.New(2*l+2))
    {
        uchar *p = NewSeed.GetPtr();
        short2char(1,p);
        bigint2char(n,p,l);
        bigint2char(x,p,l);
        return 0;
    }
    return 1;
}

RandomBase::ReadyFlag RABBIT_Random::SetUserSeed(const MEM &Seed)
{
    size_t l;
    IsReadyFlag = NotReady_flag;
    if (Seed.GetSize()>=2)
    {
        uchar *p = Seed.GetPtr();
        l = char2short(p);
        // l is the length of n ,e and x in bytes.
        if (Seed.GetSize()==2*l+2)
        {

```

```

        char2bigint(p+2,1,n);
        char2bigint(p+2+1,1,x);
        IsReadyFlag = Ready_flag;
    }
}
return IsReadyFlag;
}

uchar RABBIT_Random::GetUChar()
{
    uchar c=0;
    int i;
    for (i=8;i--i)
    {
        square(x,x);
        remainder(x,x,n);
        if(x >= n/2) x = n - x;
        c<<=1;
        c |=x.is_odd();
    }
    return c;
}

```

## A.4 Der diskrete Exponentialgenerator I

```

// diex.c
//
// 11/1995 Baerbel Mueller
//

#include "randombase.h"
#include "seedgen.h"
#include <LiDIA/bigint.h>
#include "big_defs.h"
#include "diex.h"

static bigint pr,q[512];
static int    n[512];
static int    numofq;

```

```

void DIEX_Random::GenNumber(RandomBase *Rand, int i, int l) const
{
    // i = Wievielter Faktor von p
    // l = Anzahl Bits des zu generierenden Faktors

    int r = (Rand?Rand:&SeedGen)->GetULong()%l + 1;

    Random((r+7)/8,q[i],Rand);
    q[i] &= (bigint(1)<<r)-1;
    q[i] |= bigint(1)<<(r-1);
    q[i] = next_prime(q[i]);
    n[i] = r;
    // cout << "F[" << i <<"] : " << q[i] << " ( = " << n[i] << " Bits)\n";
    pr *= q[i];

    if (l>r)
        GenNumber(Rand,i+1,l-r);
    else
    {
        numofq = i+1;
        cout << "Versuch mit " << numofq << " Faktoren.\n";
        for (int j=0; j<numofq; ++j)
            cout << q[j] << "\n";
        cout << "----> p = " << pr << "( = " << pr.bit_length()
            << " Bits)\n\n";
    }
}

int DIEX_Random::MakeNewSeed(DATA &NewSeed, int SecLevel,
                            RandomBase *Rand) const
{
    int l,r,i,j,z=0;
    bigint e,g,x;

    l = (SecLevel<0)?128:((SecLevel==0)?256:512);
    // l is the length of p in bits.

    pr.assign_one();
    numofq = 0;
    r = l-1;

```

```

    j = 0;
    do
    {
//cout << "Habe noch " << r << " Bits zu generieren\n";
        if (r>1)
            {
GenNumber(Rand,numofq,r);
pr <<= 1;
inc(pr);
cout << "----> p = " << pr << "( = " << pr.bit_length()
        << " Bits)\n\n";
if (is_prime(pr,8))
    break;
r=0;
        }
        pr.assign_one();
        i = 0;
        while (i<numofq)
            {
                if (j==0)
                    {
z = (Rand?Rand:&SeedGen)->GetUChar();
j = 8;
                    }
if (z&1)
            {
pr *= q[i];
++i;
            }
else
            {
//cout << "Loesche " << q[i] << " ( " << n[i] << " Bits)\n";
r += n[i];
//pr /= q[i];
--numofq;
q[i] = q[numofq];
n[i] = n[numofq];
            }
--j;
z>>=1;
        }
}

```

```

    } while (1);

    q[numofq] = 2;
    n[numofq] = 2;
    ++numofq;

/*
    bigint p,q[1024],e,x,g;
    int l,k,r,i,num,nr=0;

        q[0] = 2;
        do
            {
cout << "Versuch " << ++nr << ":\n";

k=1*8-2;
p=2;
num=1;

do
    {
        r = (Rand?Rand:&SeedGen)->GetULong()%(k-1) + 2;
        Random((r+15)/8,q[num],Rand);
        q[num] &= (bigint(1)<<r)-1;
        q[num] = next_prime(q[num]);
        k -= r;
        p *= q[num];
        ++num;
    } while( k>1);
cout << "generate " << num << " factors\n";

        inc(p);
        } while (!is_prime(p,5));
*/

cout << "p found !!!!!!!!!!!!!\n";

//step1:
    g.assign_one();
step2:
    inc(g);

```

```

        i = 0;
step3:
    power_mod(e,g,(pr-1)/q[i],pr);
    if (e.is_one())
        goto step2;
    else
++i;
    if (i<numofq)
        goto step3;

    l = byte_length(pr);

    Random(1,x,Rand);
    x %= pr;

    cout << " p = " << pr << '\n';
    cout << " g = " << g << '\n';
    cout << " x = " << x << '\n';

    if (NewSeed.New(2+3*l))
    {
        uchar *ptr = NewSeed.GetPtr();
        short2char(l,ptr);
        bigint2char(pr,ptr,l);
        bigint2char(g,ptr,l);
        bigint2char(x,ptr,l);
        return 0;
    }
    return 1;
}

RandomBase::ReadyFlag DIEX_Random::SetUserSeed(const MEM &Seed)
{
    size_t l;
    IsReadyFlag = NotReady_flag;
    if (Seed.GetSize()>=2)
    {
        uchar *ptr = Seed.GetPtr();
        l = char2short(ptr);
        // l is the length of p,g and x in bytes.
        if (Seed.GetSize()==2+3*l)

```

```

    {
        char2bigint(ptr+2,l,p);
        char2bigint(ptr+2+1,l,g);
char2bigint(ptr+2+2*1,l,x);
        k = p.bit_length()-1;
        IsReadyFlag = Ready_flag;
    }
}
return IsReadyFlag;
}

```

```

uchar DIEX_Random::GetUChar()
{
    uchar c;
    int i;

    power_mod(x,g,x,p);
    for (i=8; i; --i)
    {
        c <<= 1;
        c |= x.bit(i);
    }
    return c;
}

```

## A.5 Der diskrete Exponentialgenerator II

```

// diex2.c
//
// 03/1996 Baerbel Mueller
//

#include "randombase.h"
#include "seedgen.h"
#include <LiDIA/bigint.h>
#include "big_defs.h"
#include "diex2.h"

```

```

// static int    n;
// static int    numofq;

int DIEX2_Random::MakeNewSeed(DATA &NewSeed, int SecLevel,
                              RandomBase *Rand) const
{
    int l,k,r,i,j,z=0;
    bigint pr,q0,q1,q2,m,e1,e2,g,x;

    l = (SecLevel<0)?16:((SecLevel==0)?32:64);
    // l is the length of pr in bytes.

    do
    {
k=l*8-1;

r = k>>2;
Random((r+7)/8,q0,Rand);
q0 &= (bigint(1)<<r)-1;
q0 |= bigint(1)<<(r-1);
k-=r;

        r = (Rand?Rand:&SeedGen)->GetULong()%(k-3)+2;

        Random((r+7)/8,q1,Rand);
q1 &= (bigint(1)<<r)-1;
q1 |= bigint(1)<<(r-1);
        q1 = next_prime(q1);
k -= r;

        Random((k+7)/8,q2,Rand);
q2 &= (bigint(1)<<k)-1;
q2 |= bigint(1)<<(k-1);
        q2 = next_prime(q2);

```

```

m = q0*q1*q2*2;
    pr = m+1;
    } while (!is_prime(pr,10));

cout << "p found !!!!!!!!!!!!!\n";

/* //step1:
    g.assign_one();
step2:
    inc(g);
    i = 0;
step3:
    power_mod(e,g,(pr-1)/q[i],pr);
    if (e.is_one())
        goto step2;
    else
++i;
    if (i<numofq)
        goto step3;

*/

// Bestimme Element mit Ordnung q1 oder q2

    g.assign_one();
step2:
    inc(g);
    power_mod(e1,g,(pr-1)/q1,pr);
    power_mod(e2,g,(pr-1)/q2,pr);
    if ((e1.is_one())&&(e2.is_one()))
        goto step2;

l = byte_length(pr);

```

```

Random(l,x,Rand);
x %= pr;

cout << " p = " << pr << '\n';
cout << " g = " << g << '\n';
cout << " x = " << x << '\n';

if (NewSeed.New(2+3*1))
{
    uchar *ptr = NewSeed.GetPtr();
    short2char(l,ptr);
    bigint2char(pr,ptr,l);
    bigint2char(g,ptr,l);
    bigint2char(x,ptr,l);
    return 0;
}
return 1;
}

RandomBase::ReadyFlag DIEX2_Random::SetUserSeed(const MEM &Seed)
{
    size_t l;
    IsReadyFlag = NotReady_flag;
    if (Seed.GetSize()>=2)
    {
        uchar *ptr = Seed.GetPtr();
        l = char2short(ptr);
        // l is the length of p,g and x in bytes.
        if (Seed.GetSize()==2+3*1)
        {
            char2bigint(ptr+2,l,pr);
            char2bigint(ptr+2+1,l,g);
            char2bigint(ptr+2+2*1,l,x);
            k = pr.bit_length()-1;
            IsReadyFlag = Ready_flag;
        }
    }
    return IsReadyFlag;
}

```

```

uchar DIEX2_Random::GetUChar()
{
    uchar c;
    int i;

    power_mod(x,g,x,pr);
    for (i=8; i; --i)
        {
            c <<= 1;
            c |= x.bit(i);
        }
    return c;
}

```

## A.6 Der Generator auf elliptischen Kurven

```

// elli.c
//
// 12/1995 Baerbel Mueller
//
//

#include "randombase.h"
#include "seedgen.h"
#include <LiDIA/bigint.h>
#include "big_defs.h"
//#include "ec_weier.h"
#include "elli.h"

int ELLI_Random::MakeNewSeed(DATA &NewSeed, int SecLevel,
                             RandomBase *Rand) const
{
    bigint p,q0,q1,q2,m,b,x,y,o,factor=1;
    int l,k,r;

    bigmod bmm,xmm,ymm,dummy1,dummy2; //deklaration
    ec_curve ell_curve;
    ec_point_W punkt;

```

```

    l = (SecLevel<0)?16:((SecLevel==0)?32:64);
    // l is the length of p in bytes.

    do
        {
//cout << "Versuch " << ++nr << ":\n";

        k=l*8-1;

                //r = (Rand?Rand:&SeedGen)->GetULong()%((k>>2)-1)+2;
r = k>>2;
Random((r+7)/8,q0,Rand);
q0 &= (bigint(1)<<r)-1;
q0 |= bigint(1)<<(r-1);
k-=r;

                r = (Rand?Rand:&SeedGen)->GetULong()%(k-3)+2;

                Random((r+7)/8,q1,Rand);
q1 &= (bigint(1)<<r)-1;
q1 |= bigint(1)<<(r-1);
        q1 = next_prime(q1);
k -= r;

                Random((k+7)/8,q2,Rand);
q2 &= (bigint(1)<<k)-1;
q2 |= bigint(1)<<(k-1);
        q2 = next_prime(q2);

//cout << " q0 = " << q0 << " (= " << q0.bit_length() << " Bits)\n";
//cout << " q1 = " << q1 << " (= " << q1.bit_length() << " Bits)\n";
//cout << " q2 = " << q2 << " (= " << q2.bit_length() << " Bits)\n";

        m = q0*q1*q2*2;
//cout << " m = " << m << " (= " << m.bit_length() << " Bits)\n";
        p = m-1;
        } while (!is_prime(p,10));

//cout << "p found !!!!!!!!!!!!!\n";

    l = byte_length(p);

```

```

bigmod::set_modulus(p);

do
  {
//int t=0;
    do
  {

    Random(1,x,Rand);
    Random(1,y,Rand);
    x %= p-1;
    inc(x);
    y %= p-1;
    inc(y);

    xmm.assign(x);
    ymm.assign(y);

    square(dummy1,ymm);
    power(dummy2,xmm,3);
    dummy2=dummy1-dummy2;

    bmm.assign(dummy2);

    square(dummy1,bmm);
      dummy1 *= 27;
//cout << "x = " << xmm << "\n";
//cout << "y = " << ymm << "\n";
//cout << "b = " << bmm << "\n";
//cout << "Test " << ++t << " : " << dummy1 << "\n";
  }
    while(dummy1.is_zero());

    ell_curve.assign(0,bmm);

    ec_point_W Z(ell_curve,xmm,ymm);
    punkt.assign(Z);

    //%------

```

```

        //%% Bestimme Ordnung von Punkt (xmm,ymm)

o=1;

        ell_multiply(Z,punkt,((m/q1-1)),factor);
//cout << "2. MakeNewSeed: factor=" << factor << "\n";

        if (factor.is_one())
{
    if(!((Z.x==punkt.x)&&!(Z.y+punkt.y)))
        // m/q1 mal punkt nicht neutrales Element
        // o=o*q1;
o=0;

    else
        {
            ell_multiply(Z,punkt,((m/q2-1)),factor);
//cout << "2. MakeNewSeed: factor=" << factor << "\n";
            if (factor.is_one())
{
                if(!((Z.x==punkt.x)&&!(Z.y+punkt.y)))
                    // m/q2 mal punkt nicht neutrales Element
                    // o=o*q2;
o=0;

            }

        }

}

while(o==1);

b=mantissa(bmm);

if (NewSeed.New(2+4*1))
{
    uchar *ptr = NewSeed.GetPtr();
    short2char(1,ptr);
    bigint2char(p,ptr,1);
    bigint2char(b,ptr,1);
}

```

```

        bigint2char(x,ptr,l);
        bigint2char(y,ptr,l);

        return 0;
    }
    return 1;
}

RandomBase::ReadyFlag ELLI_Random::SetUserSeed(const MEM &Seed)
{
    size_t l;
    bigint b,x,y;
    IsReadyFlag = NotReady_flag;
    if (Seed.GetSize()>=2)
    {
        uchar *ptr = Seed.GetPtr();
        l = char2short(ptr);
        // l is the length of p,g and x in bytes.
        if (Seed.GetSize()==2+4*l)
        {
            char2bigint(ptr+2,l,p);
            char2bigint(ptr+2+1,l,b);
            char2bigint(ptr+2+2*l,l,x);
            char2bigint(ptr+2+3*l,l,y);
            IsReadyFlag = Ready_flag;
        }
    }
    bigmod::set_modulus(p);
    bmm.assign(b);
    ell_curve.assign(0,bmm);

    ec_point_W Z(ell_curve,bigmod(x),bigmod(y));
    punkt.assign(Z);
    //cout << "SetUserSeed: 1. mantissa x=" << mantissa(punkt.x)
    //                                     << "\n";
    //cout << "SetUserSeed: 1. mantissa y=" << mantissa(punkt.y)
    //                                     << "\n\n";

    return IsReadyFlag;
}

uchar ELLI_Random::GetUChar()

```

```

{
    uchar c;
    bigint factor=1;
    //ec_point_W Z;

    int i;
    for (i=8;i;--i)
    {
        //cout << "GetUChar: 1. mantissa x=" << mantissa(punkt.x) << "\n";
        //cout << "GetUChar: 1. mantissa y=" << mantissa(punkt.y)
            << "\n\n";
        ell_multiply(punkt,punkt,mantissa(punkt.y),factor);
        c<<=1;
        if(mantissa(punkt.y) > (p/2))
            c |= 1;
        //cout << "GetUChar: 2. mantissa x=" << mantissa(punkt.x) << "\n";
        //cout << "GetUChar: 2. mantissa y=" << mantissa(punkt.y)
            << "\n\n";
    }
    return c;
}

```

## A.7 Der Seedgenerator

```

// timerand.c
//
// 10/1995 Jochen Schwarz
//

#include "randombase.h"
// #include "seedgen.h"
#include "timerand.h"

volatile int Timer_Random::loop_flag;

int Timer_Random::GetBit()
{
    int c=0;
    int d=0;

```

```

do
{
    loop_flag = 1;
    while (loop_flag)
        ++c;
    c &= 1;
    if (SecLevel<=0)
break;
    loop_flag = 1;
    while (loop_flag)
        ++d;
    d &= 1;
} while (c==d);
return c;
}

void Timer_Random::StopLoop(...)
{
    loop_flag = 0;
}

RandomBase::ReadyFlag Timer_Random::SetUserSeed(const MEM &Seed)
{
    IsReadyFlag = NotReady_flag;
    if (Seed.GetSize()==2)
    {
        SecLevel = char2short(Seed.GetPtr());
        itimer.it_value.tv_sec = 0;
        itimer.it_value.tv_usec = 9800;
        itimer.it_interval = itimer.it_value;
        #if defined(USE_SIGACTION)
act.sa_handler = StopLoop;
act.sa_mask = 0;
act.sa_flags = SA_RESTART;
        #endif
        IsReadyFlag = Ready_flag;
    }
    return IsReadyFlag;
}

uchar Timer_Random::GetUChar()

```

```

{
    uchar c;
    Fill(&c,1);
    return c;
}

uchar *Timer_Random::Fill(uchar *Dest, size_t Size)
{
    uchar r;
    int i;
    itimerval old_itimer;
    #if defined(USE_SIGACTION)
        struct sigaction oldact;
        sigaction(SIGALRM,&act,&oldact);
    #else
        void (*old_signal)(int);
        old_signal = signal(SIGALRM,StopLoop);
    #endif

    setitimer(ITIMER_REAL,&itimer,&old_itimer);

    while (Size)
    {
        for (r=0,i=8;i--i)
        {
r <<= 1;
r |= GetBit();
        }
        *(Dest++) = r;
        --Size;
    }
    setitimer(ITIMER_REAL,&old_itimer,0);
    #if defined(USE_SIGACTION)
        sigaction(SIGALRM,&oldact,0);
    #else
        signal(SIGALRM,old_signal);
    #endif
    return Dest;
}

// timerand.h

```

```

//
// generates random bytes with timer device
//
// 10/1995 Jochen Schwarz
//

#ifndef TIMERAND_H
#define TIMERAND_H

#include <signal.h>
#include <sys/time.h>

class Timer_Random : public RandomBase
{
private :
    itimerval itimer;
    #if defined(USE_SIGACTION)
        struct sigaction act;
    #endif

    int SecLevel;
    int GetBit();
    static void StopLoop(...);
    static volatile int loop_flag;

public :
    Timer_Random(const MEM &Seed)
    {
        SetUserSeed(Seed);
    }
    Timer_Random()
    {
        IsReadyFlag = NotReady_flag;
    }
    ~Timer_Random()
    {
    }

    ReadyFlag SetUserSeed(const MEM &Seed);

    uchar GetUChar();

```

```
    uchar *Fill(uchar *Dest, size_t Size);  
};  
  
#endif
```

# Stichwortverzeichnis

xor-Funktion .....	8	inverses Element .....	7
algebraisch abgeschlossen .....	8	Jacobi Symbol .....	16
algebraischer Abschluß .....	8	Körper .....	8
Annahme		endlicher .....	8
für quadratische Reste .....	16	Laufzeit .....	11
für den diskreten Logarithmus		beschränkte .....	11
in $(\mathbb{Z}/p\mathbb{Z})^*$ .....	16	Legendre Symbol .....	15
für den diskreten Logarithmus		most significant bit .....	23
über elliptische Kurven .....	17	neutrales Element .....	7
für Faktorisierung .....	16	One-time-Pad .....	10
für RSA .....	17	One-way Funktion .....	14
Band .....	10	Ordnung .....	7
Charakteristik .....	8	Primkörper der Charakteristik .....	8
Ensemble .....	12	Pseudozufallszahlengenerator ....	10, 13
Eulersche Phi-Funktion .....	16	quadratischer Rest .....	15
Generator		Quadratwurzel .....	15
auf elliptischen Kurven .....	23	Seed .....	13
diskreter Exponential .....	23	Seedgenerator	
modifizierte Rabin-Bit .....	22	timerand.c .....	68
nach Blum-Blum-Shub .....	20	Turingmaschine .....	10
RSA-Bit .....	22	deterministische .....	11
Gruppe .....	7	nichtdeterministische .....	11
abelsch .....	7	probabilistische .....	11, 12
implementierter Generator		Übergang .....	11
bbs.c .....	48	Übergangsrelation .....	11
bitrsa.c .....	50	Unterscheider .....	12
diex.c .....	54	ununterscheidbar .....	12, 13
diex2.c .....	59		
elli.c .....	63		
rabbit.c .....	52		

vernachlässigbar .....	12
Wahrscheinlichkeit .....	11
Wahrscheinlichkeitsverteilung .....	12
Zustand .....	11

# Literaturverzeichnis

- [ACGS88] Alexi, W., Chor, B., Goldreich, O., and Schnorr, C. P. RSA and Rabin function: Certain parts are as hard as the whole. *SIAM Journal on Computing*, 17:194–209, 1988.
- [Agn88] Agnew, G. B. Random Sources for Cryptographic Systems. In *Proceedings of Eurocrypt'87*, pages 77–81, 1988.
- [AT&T86] AT&T. Random Number Generator. In *Data Sheet*, 1986.
- [Bac83] Bach, Eric. How to Generate Random Integers with Known Factorization. In *Proc. of the Fifteenth Annual ACM Symposium on Theory of Computing*, pages 184–188, 1983.
- [BB91] Biehl, Ingrid und Buchmann, Johannes. Introduction to theoretical Cryptography, 1991. Vorlesungsskript.
- [BBP95] Biehl, Ingrid, Buchmann, Johannes, and Papanikolaou, Thomas. LiDIA: a library for computational number theory. Technical report, SFB 124, Universität des Saarlandes, 1995.
- [BBS86] Blum, L., Blum, M., and Shub, M. A simple unpredictable Pseudo-Random-Number-Generator. *SIAM Journal on Computing*, 15:364–383, 1986.
- [BKM<sup>+</sup>96] Biehl, I., Kenn, H., Meyer, B., Müller, B., Schwarz, J., and Thiel, Chr. LiSA — Eine C++-Bibliothek für kryptographische Verfahren. In *Arbeitskonferenz Digitale Signaturen der GI-Fachgruppe 2.5.3*, 1996.
- [BM84] Blum, Manuel and Micali, Silvio. How to generate Cryptographically Strong Sequences of Pseudo-Random Bits. *SIAM Journal on Computation*, 1984.
- [Coh93] Cohen, Henri. *A Course in computational Algebraic Number Theory*. Springer, 1993.
- [DIF94] Davis, D., Ithaka, R., and Fenstermacher, P. Cryptographic Randomness from Air Turbulence in Disc Drives. In *Proc. of CRYPTO'94*, pages 114–120, 1994.

- [Eli87] Elias, P. Internal and recency rank source coding. *IEEE Trans. Inform. Theory*, pages 3–10, 1987.
- [Fis95] Fischer, Gerd. *Lineare Algebra*, volume 10. Vieweg, 1995.
- [HILL90] Hastad, J., Impagliazzo, R., Levin, L., and Luby, M. Pseudo-Random Generators from Any One-way Function. In *Ann. ACM Symp. on Theory of Computing*, pages 395–404, 1990.
- [Jv91] Jell, Thomas und von Reeken, Axel. *Objektorientiertes Programmieren mit C++*. Carl Hanser, 1991.
- [Kal86] Kaliski, Burton S. Jr. Pseudo - Random Bit Generator based on Elliptic Logarithms. In *Proceedings of CRYPTO'86*, pages 84–103, 1986.
- [Kan94] Kanne, R. Eine objektorientierte Klassenbibliothek für Kryptosysteme. Diplomarbeit, Institut für Informatik der Universität Hildesheim, 1994.
- [Ken96] Kenn, Holger. Entwurf eines komplexen Client-Server-Systems zur globalen Verteilung von Schlüsseldaten asymmetrischer Kryptosysteme. Diplomarbeit, Universität des Saarlandes, 1996. in Vorbereitung.
- [Knu81] Knuth, Donald E. *The Art of computer Programming*, volume 2. Addison-Wesley, 1981.
- [Kob94] Koblitz, Neal. *A Course in Number Theory and Cryptography*. 2. Springer, 2 edition, 1994.
- [Lev87] Levin, L. A. One-Way Functions and Pseudorandom Generators. *Combinatorica*, 7(4):357–363, 1987.
- [LMS93] Lacy, J. B., Mitchel, D. P., and Schull, W. M. CcryptoLib: Cryptograohy in software. In *UNIX Security Symposium IV Proceedings*, pages 1–7. USENIX Association, 1993.
- [LP81] Lewis, Harry R. and Papadimitriou, Christos H. *Elements of the theory of Computation*. Prentice-Hall, 1981.
- [LW88] Long, D. L. and Wigderson, A. The discret logarithm hides  $O(\log n)$  bits. *SIAM journal on Computing*, 17(2):363–372, 1988.
- [Mau90] Maurer, Ueli M. A Universal Statistical Test for Random Bit Generators. In S.A. Vanstone and A.J. Menezes, editors, *Advances in Cryptology – CRYPTO'90*, pages 409–420, Heidelberg, 1990. Springer.
- [Mau95] Maurer, U. M. Fast Generation of Prime Numbers and Secure Public-Key Cryptographic Parameters. *Journal of Cryptology*, 8(3):123–155, 1995.

- [Mey76] Meyberg, Kurt. *Algebra II*. Carl Hauser, 1976.
- [Mey92] Meyer, Bernd. Bit-Commitment-Schemes. Diplomarbeit, Universität des Saarlandes, 1992.
- [MVO91] Menezes, A., Vanstone, S., and Okamoto, T. Reducing elliptic curve logarithms to logarithms in a finite field. In *Ann. ACM Symp. on Theory of Computing*, pages 80–89, 1991.
- [RSV94] Reiffen, Scheja, und Vetter. *Algebra*. BI Hochschultaschenbuchverlag, 1994.
- [Sch96a] Schneier, Bruce. *Applied Cryptography*, chapter 16, pages 369–395. John Wiley and Sons, Inc., New York, 2 edition, 1996.
- [Sch96b] Schwarz, Jochen. Entwurf und Implementierung einer objektorientierten Softwarebibliothek kryptographischer Algorithmen. Diplomarbeit, Universität des Saarlandes, 1996. in Vorbereitung.
- [Sha49] Shannon, C. E. Communication Theory of Secret Systems. *Bell System Technical Journal*, 28(4):656–715, 1949.
- [Sil86] Silverman, J. *The Arithmetic of Elliptic Curves*. 106. Springer, 1986.
- [Sti95] Stinson, D. S. *Cryptography*. CRC Press, 1995.
- [Thi93] Thiel, Christian. Zur Identität von  $\mathcal{IP}$  und  $\mathcal{ZK}$ . Diplomarbeit, Universität des Saarlandes, 1993.
- [Wil89] Willems, F. M. J. Universal data compression and repetition times. *IEEE Trans. Inform. Theory*, pages 54–58, 1989.
- [Yao82] Yao, A. Theory and applications of trapdoor functions. In *IEEE 23rd Symposium on Foundation of Computer Science*, pages 80–91, 1982.
- [Zim95] Zimmermann, P. R. *PGP Source Code and Internals*. MIT Press, 1995.
- [Ziv90] Ziv, J. Tests for randomness and estimating the statistical model of an individual sequence. In R. M. Capocelli, editor, *Sequences*, New York, 1990. Springer.