Bitstromverschlüsselung
und
Zufallsgeneratoren

Diplomarbeit

vorgelegt dem Fachbereich Mathematik
der Johannes Gutenberg-Universität Mainz
von

Bernd Ulmann
Schwalbacherstr. 27
65307 Bad Schwalbach Hettenhain

Das Thema der Arbeit stellte
Herr Professor Dr. Pommerening

Vorwort

Die vorliegende Diplomarbeit befaßt sich mit Verfahren zur Erzeugung pseudozufälliger Zahlen, sowie deren Verwendung für kryptographische Zwecke. In diesem Rahmen wurde eine Programmbibliothek mit Funktionen zur Bitstromverschlüsselung auf der Grundlage kryptographisch sicherer Zufallsgeneratoren entwickelt.

Ich möchte an dieser Stelle allen Personen danken, die diese Arbeit gefördert und unterstützt haben. Zuallererst sei Herrn Professor Dr. K. Pommerening für meine Aufnahme als Diplomand und die interessante Themenstellung gedankt.

Dem Zentrum für Datenverarbeitung und insbesondere Herrn M. König danke ich für die Bereitstellung eines stets verläßlich arbeitenden VMS-Clusters, sowie die großzügige Quota-Vergabe. Weiterhin sei Herrn König für die Geduld gedankt, mit der er sein Büro mit mir geteilt hat.

Dank gebührt Herrn Oliver Wülk für seine große Hilfe bei der Arbeit mit LaTeX, sowie allen Korrekturlesern für ihre Mühe und Geduld mit meinem Text und Herrn Wolfgang J. Möller von der Gesellschaft für Wissenschaftliche Datenverarbeitung, Göttingen für die Hilfe bei der Portierung des GNU-MP Paketes auf VMS.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Kapitel 1

Einleitung

1.1 Vorbemerkungen

Das Bedürfnis, Nachrichten in verschlüsselter Form zu übermitteln besteht eigentlich, seit sich die Menschheit im Besitz der Schrift befindet. Zu allen Zeiten gab es Informationen, die nur für bestimmte Empfänger vorgesehen waren, jedoch über *unsichere* Kanäle übermittelt werden mußten, so daß stets viel Erfindungsreichtum auf die Unkenntlichmachung (*Steganographie*) oder Verschlüsselung (*Kryptographie*) solcher Daten verwandt wurde.

Mit dem Aufkommen moderner Kommunikationstechniken und nicht zuletzt dank der Möglichkeiten, die die elektronische Datenverarbeitung zur Verfügung stellt, wurde ein neues Kapitel in der Geschichte der Kryptologie eingeleitet. War bis dato das Datenaufkommen noch vergleichsweise gering – von den Anfängen, in denen man Nachrichten auf die Kopfhaut von Sklaven schrieb, um sie unter dem nachwachsenden Haarschopf zu verbergen, über einfache Verschiebechiffren (Caesar-Chiffre), bis hin zu komplexeren Vorgehensweisen, wie der zu Berühmtheit gelangten *Enigma* –, so waren entsprechend die zu verarbeitenden Informationsmengen von einem Ausmaß, daß sie ohne größere Schwierigkeiten von Hand oder unter Zuhilfenahme einfacher mechanischer oder elektromechanischer Hilfsmittel verarbeitet werden konnten.

Eine radikale Wende trat ein, als das Datenaufkommen Dimensionen annahm, die nicht nur den Begriff der Informations*flut* nach sich zogen, sondern auch eine Verarbeitung mit Hilfe elektronischer Datenverarbeitungsanlagen regelrecht erzwangen. Hierdurch wurden einerseits sehr viel komplexere Verschlüsselungsverfahren praktikabel, andererseits wurde auch die Entschlüsselung von Chiffren möglich, an die bislang nicht gedacht werden konnte; der Anfang dieser Entwicklung ist sicherlich in *Colossus* zu sehen – einem vergleichsweise spezialisierten Rechner, an dessen Entwicklung *Alan Turing* maßgeblich beteiligt war, der von seiner Konzeption her einzig für das Brechen des Enigma-Codes entworfen wurde; ein Unterfangen, das letztlich auch von Erfolg gekrönt war.

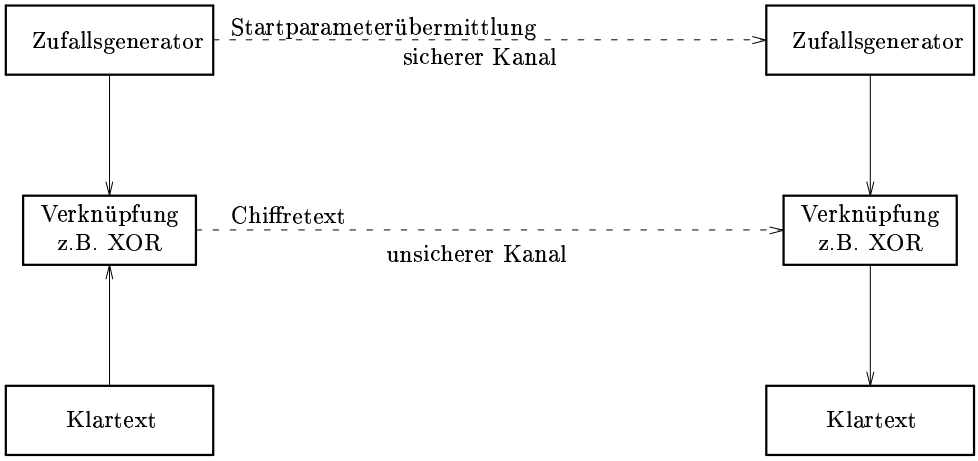

Abbildung 1.1: Pseudozufallsgeneratoren und Bitstromverschlüsselung

1.2 Bitstromverschlüsselung

Die allgemeine Idee, die einer jeden Bitstromverschlüsselung zugrunde liegt, zeigt Figur 1.1. Ein Nachrichtenstrom, der in seiner allgemeinsten Form als Bitstrom aufgefaßt werden kann, soll über einen unsicheren Nachrichtenkanal transportiert werden, so daß durch geeignete Verschlüsselungstechniken sichergestellt werden muß, daß eventuelle Abhörer keinen Nutzen aus ihrer Tätigkeit ziehen können. Zum einen stellt sich also die Forderung nach einem leistungsfähigen Verfahren, zum anderen dürfen jedoch auch ökonomische Gesichtspunkte nicht außer Acht gelassen werden. Das verwendete Chiffrierverfahren sollte demnach kein allzuschlechtes Zeitverhalten aufweisen, um auch die Verschlüsselung großer Datenmengen zu ermöglichen, wie sie als extremes Beispiel in modernen Netzwerken auftreten, die auf Ethernet oder gar FDDI basieren. Weiterhin sollten die zu verarbeitenden Daten nicht einfach blockweise mit stets ein und demselben Schlüssel chiffriert werden, um das Brechen der Verschlüsselung durch statistische Verfahren und Klartextraten nicht unnötig zu erleichtern[1].

Gerade aus dieser letzten Forderung ergeben sich weitere Problemstellungen: Wie sollen teilweise sehr große Datenmengen verschlüsselt werden, ohne wiederholt auf ein und denselben Schlüsseltext zurückgreifen zu müssen? Hierfür existieren mehrere Lösungsansätze – eine der naheliegendsten Methoden besteht darin, vorhandene Daten, wie sie sich beispielsweise auf CDs (oder in früheren

[1] Ein Extremfall hiervon ist in der Chiffrierung mit nicht wiederholtem Schlüssel zu sehen – derartige Chiffrierverfahren sind (nach Shannon) perfekt, d.h. ohne Kenntnis des Schlüssels nicht brechbar, da das Kryptogramm nicht von einer Zufallsfolge unterscheidbar ist. Im Idealfall ist folglich ein nicht zyklischer Schlüssel zu verwenden, dessen Länge der des Klartextes entspricht. Problematisch hierbei ist die Übermittlung des Schlüssels selbst - ist man in der Lage, einen Schlüssel mit einer Länge, die der des Klartextes entspricht, sicher zu übermitteln, erübrigt sich im Grunde genommen das Verschlüsseln des Klartextes, da man in diesem Fall auch selbigen anstelle des Schlüssels sicher übertragen könnte.

Zeiten in Form von Büchern) finden, zur Verschlüsselung der Nutzdaten zu verwenden. Bei diesem Verfahren muß der autorisierten Empfangsgegenstelle lediglich die Quelle der Schlüsseldaten mitgeteilt werden (beispielsweise Buchtitel, sowie Startseite/-zeile etc.), um diese in die Lage zu versetzen, die Daten rekonstruieren zu können.

Es ist einsichtig, daß dieses Verfahren einige schwerwiegende Mängel aufweist: Einerseits kann es unter einfachen Bedingungen nicht die Sicherheit gewährleisten, die in den meisten Fällen als zwingend vorausgesetzt wird; andererseits läßt auch die Flexibilität dieser Vorgehensweise zu wünschen übrig, ist doch stets mindestens ein Exemplar aller in Frage kommenden Schlüsseldatenquellen vorrätig zu halten, von etwaigen Update-Problemen zwischen Sender und Empfänger einmal ganz abgesehen (ein köstliches Beispiel hierfür findet sich in Johannes Mario Simmels *Es muß nicht immer Kaviar sein*).

Ein weiteres Problem, das sich im allgemeinen bei der Verschlüsselung stellt, ist die Frage nach der sicheren Übermittlung des verwendeten Schlüssels. Dies vereinfacht sich in dem Maße, in dem der Schlüssel kurz ist, was allerdings im Widerspruch zu der geforderten Bedingung steht, möglichst nicht wiederholte Sequenzen zur Chiffrierung zu verwenden.

Einen möglichen Lösungsansatz für all diese Probleme, der Kernpunkt dieser Arbeit sein wird, stellen Zufallsgeneratoren als Schlüsseldatenquellen dar. Sie ermöglichen es, ausgehend von einem vergleichsweise kurzen Schlüssel, eine lange, zum Chiffrieren geeignete Folge von Ausgabewerten zu liefern.

1.3 Eigenschaften der Bitstromverschlüsselung

Wie aus Abbildung 1.1 ersichtlich ist, hängt der Zustand des verwendeten Zufallsgenerators bei der gezeigten Betriebsart in keiner Weise von den zu verschlüsselnden Klartextdaten ab, so daß sich eventuelle Übertragungsfehler, die lediglich zur Verfälschung eines oder auch mehrerer Zeichen führen, nur auf das betroffene Zeichen selbst auswirken. Der Vorteil dieser Betriebsart ist also der Ausschluß von Fortsetzungsfehlern, wie sie bei anderen Verfahren durchaus gegeben sind, so daß sie sich hauptsächlich für Anwendungsfälle als geeignet erweist, in denen mit nicht zu vernachlässigender Wahrscheinlichkeit mit Übertragungsfehlern gerechnet werden muß, wie dies beispielsweise bei Satellitenverbindungen etc. der Fall ist. Diesem großen Vorteil stehen jedoch einige prinzipbedingte Nachteile gegenüber, auf die im folgenden kurz eingegangen werden soll.

Da, wie bereits erwähnt, die Ausgabefolge des Zufallsgenerators nicht vom Klartext selbst abhängt, geht die Möglichkeit der Textverkettung verloren, so daß Manipulationen des Schlüsseltextes, wie z.B. Einspielungen zuvor aufgezeichneter Schlüsseltextfolgen, nicht ohne weiteres (d.h. ohne Verwendung von Zeitmarken oder ähnlichen Mechanismen) erkannt werden können.

Ein weiteres generelles Problem entsteht aus der Tatsache, daß diese Betriebsart nicht selbstsynchronisierend ist, so daß hierfür gesondert Sorge getragen werden muß, falls zu befürchten steht, daß Nachrichten eventuell unvollständig (unter Auslassung einzelner Zeichen des Schlüsseltextes) oder mit

zusätzlichen Zeichen übermittelt werden. Andernfalls kann der Rest der Nachricht nach Verlust der Synchronisation zwischen Sender und Empfänger nicht mehr korrekt entschlüsselt werden.

Bei der Anwendung dieser Betriebsart muß darauf geachtet werden, daß nicht mehrere Nachrichten mit demselben Initialisierungswert beginnend, verschlüsselt werden. Angenommen, es werden zwei Nachrichten $\vec{m}$ und $\vec{m'}$ ausgehend von ein und demselben Startwert verschlüsselt, lassen sich aus den entsprechenden Schlüsseltexten $\vec{c}$ und $\vec{c'}$ eventuell über die bitweise Addition modulo 2

$$\vec{c} \oplus \vec{c'} = \vec{m} \oplus \vec{m'} \tag{1.1}$$

Rückschlüsse auf die zugrundeliegenden Klartexte $\vec{m}$ und $\vec{m'}$ ziehen.

Das folgende Kapitel soll in Vorbereitung auf Kapitel 3, das die im Rahmen dieser Arbeit betrachteten und implementierten Verfahren darlegt, einen allgemeinen Überblick über Entwicklung und Eigenschaften von Zufallsgeneratoren geben, wobei ausgehend von historischen Betrachtungen auch aktuelle Verfahren behandelt werden.

Kapitel 2

Zufallsgeneratoren

Der Begriff der Zufälligkeit spielt in wachsendem Maße nicht nur in der Informatik eine Rolle (zur Entwicklung probabilistischer Algorithmen), sondern auch in der modernen Kryptographie, worauf in den folgenden Kapiteln das Hauptaugenmerk liegen soll, da die in diesem Gebiet gestellten Anforderungen zumeist weit über rein statistische Betrachtungen hinausgehen. Ein Zufallsgenerator mag ein statistisch durchaus brauchbares Verhalten besitzen, kryptographisch jedoch unzumutbare Sicherheitsmängel aufweisen.

Zunächst sollte eine allgemeine Bemerkung zum Begriff des *Zufallsgenerators* an sich gemacht werden: Es ist leicht einsichtig, daß wirklich zufällige Bitfolgen nie von einem deterministischen Algorithmus erzeugt werden können – ihre Vorhersagbarkeit durch das sie erzeugende Verfahren straft die Bezeichnung *Zufall* Lügen, so daß zunächst nach einer geeigneteren Definition dieses Begriffes gesucht werden soll. Dieser findet sich in der Bezeichnung *Pseudozufallszahlen*, die treffend den Kern der Sache beschreibt – es handelt sich hierbei um Zahlen (bzw. Zahlenfolgen), die gewissen statistischen Tests genügen, aber dennoch streng deterministisch sind. Um für kryptographische Zwecke einsetzbar zu sein, müssen jedoch nicht nur diese rein statistischen Aussagen gesichert sein – darüberhinaus muß eine gewisse Form der *Unvorhersagbarkeit* vorliegen, worauf noch eingegangen werden soll.

Über die Erzeugung von Pseudozufallszahlen ist in [16, S.212] zu lesen: *It may seem perverse to use a computer, that most precise and deterministic of all machines conceived by human mind, to produce "random" numbers. More than perverse, it may seem to be a conceptual impossibility. Any program, after all, will produce output that is entirely predictable, hence not truly "random".*

Zur Erzeugung solcher Pseudozufallszahlen existiert eine Reihe mathematischer Verfahren, denen allen gemein ist, daß sie einen vergleichsweise kurzen Startwert *seed* (der wirklich zufällig gewählt werden kann) in eine längere Folge von Pseudozufallszahlen expandieren. Eine der Bedingungen, die an einen solchen Generator gestellt wird, ist, daß seine Ausgabebitkette nur *schwer* von einer wirklichen Zufallsfolge unterschieden werden kann. Präziser läßt sich also sagen:

Ein **Pseudozufallsgenerator** G ist ein deterministischer Algorithmus, der – ausgehend von einem Startwert X_n der Länge n Bits – eine zufällig anmutende Bitkette X_m mit m Bits erzeugt (wobei $m > n$ gilt), die von keinem Algorithmus T mit polynomialem Zeitverhalten von einer Kette echter Zufallsbits unterscheidbar ist[1].

Der Test-Algorithmus T erzeugt nun für jede Eingabe einen Ausgabewert von 0 bzw. 1 – man sagt, er *akzeptiert* die Eingabe, falls sie in einer 1 resultiert (im anderen Fall wird von der *Ablehnung* dieser Eingabe gesprochen). Angenommen, eine Folge von Eingabewerten ist von der Gestalt, daß für alle Elemente – außer endlich vielen – die Wahrscheinlichkeit, von T akzeptiert zu werden, in etwa der entspricht, mit der eine Folge echter Zufallswerte angenommen wird, so läßt sich schreiben:

Für jeden beliebigen Test-Algorithmus T mit polynomialem Aufwand, jede Konstante $c > 0$ und hinreichend großes n gilt:

$$|\text{Prob}\,(T(G(X_n)) = 1) - \text{Prob}\,(T(X_m) = 1)\,| < n^{-c}. \tag{2.1}$$

Hierbei stellt X_m eine wirkliche Zufallsfolge der Länge m, etwa durch faire Münzwürfe erzeugt, dar. Ist diese Bedingung erfüllt, kann folglich die Ausgabe eines solchen Generators eine echte Zufallsfolge ersetzen, wovon nicht zuletzt in probabilistischen Algorithmen Gebrauch gemacht wird.

Eine der wichtigsten Anforderungen an einen kryptographisch verwendbaren Pseudozufallsgenerator stellt die bereits erwähnte Unvorhersagbarkeit der von ihm generierten Ausgabefolge dar, d.h. kein Algorithmus mit polynomialem Aufwand darf in der Lage sein, das nächste Folgenglied dieses Ausgabestromes mit einer Wahrscheinlichkeit, die signifikant größer als $\frac{1}{2}$ ist, vorherzusagen. Hierbei können dem Algorithmus beliebig große Stücke der bisher erzeugten Folge bekannt sein, mithin auch alle bisherigen Glieder (!), um Plaintextattacks vorzubeugen.

Zufallsgeneratoren, die dieser erweiterten Bedingung genügen, werden *perfekt* genannt. Ein Pseudozufallsgenerator kann nun als *Spiel* zwischen dem eigentlichen Generator G und einem Vorhersage-Algorithmus T aufgefaßt werden, der versucht, das nächste Bit der Bitfolge vorherzusagen. Hierbei ist dem Algorithmus T außer den Startparametern alles über den betrachteten Generator G bekannt, so daß eine Geheimhaltung des verwendeten Generator-Algorithmus nicht vonnöten ist. Ist P ein beliebiges Polynom, so heißt G *perfekt* oder auch *sicher*, wenn kein Tester T ein Bit der Ausgabefolge von G mit einer Wahrscheinlichkeit von mehr als

$$\frac{1}{2} + \frac{1}{P(n)} \tag{2.2}$$

[1] Offensichtlich existiert für jeden denkbaren solchen Generator ein Test-Algorithmus, der diese Aufgabe vollzieht – für eine gegebene Teilfolge der Ausgabewerte des betrachteten Generators vollzieht dieses Verfahren eine Tiefensuche über alle möglichen Startwerte und vergleicht die solchermaßen selbst erzeugten Bitfolgen mit der vorgegebenen Kette. Nachteil dieser Vorgehensweise ist, daß ein solcher Algorithmus leider nicht in polynomialer Zeit abarbeitbar ist und somit der gestellten Bedingung nicht genügt.

vorhersagen kann[2].

2.1 Historische Betrachtungen

Den folgenden Abschnitten liegt in der Hauptsache [9] zugrunde, dessen Ausführungen weit über das hier Gesagte hinausgehen – auch finden sich dort detaillierte Ausführungen zu den Periodenlängen[3] der einzelnen Verfahren, sowie zu ihren numerischen Eigenschaften, die durch ausführliche statistische Untersuchungen abgerundet werden (für eingehendere Betrachtungen sei an dieser Stelle auf [9] verwiesen).

Die erste uns bekannte Person, die ein stochastisches Experiment (wenn auch nur in Form eines Gedankenexperimentes) zur Berechnung eines Problems verwandte, war *Buffon* im Jahre 1777. Er untersuchte folgende Fragestellung:

Gegeben sei eine große Anzahl von Nadeln der Länge L, sowie ein Gitternetz mit einem Gitterlinienabstand von D, wobei stets $L < D$ gelte. Wie groß ist die Wahrscheinlichkeit P, eine Gitterlinie zu treffen, wenn die Nadeln auf das Gitter geworfen werden?

Als Antwort hierauf fand Buffon:

$$P = \frac{2L}{\pi D}. \tag{2.3}$$

Wie man sieht, bietet sich mit dieser Aufgabenstellung eine Möglichkeit, π durch Verwendung eines Zufallsexperimentes zu ermitteln, was zwar sicherlich kein empfehlenswerter Ansatz ist, aber dennoch einen ersten Eindruck von der Leistungsfähigkeit solcher Verfahren vermittelt.

Zur Zeit des Zweiten Weltkrieges stieg der Bedarf an Zufallszahlen drastisch an, da neue Fragestellungen aufgeworfen wurden, die nicht durch konkrete Experimente lösbar waren. Sei es, daß diese Experimente zu kostspielig, zu aufwendig oder zu gefährlich sind, durch geeignete Simulationen kann man dennoch in vielen Fällen zu brauchbaren Lösungen gelangen.

Von Neumann, *Ulam* und *Metropolis* verwandten als erste in großem Stil Zufallszahlen für eine Rechentechnik, die als *Monte-Carlo*-Verfahren bekannt wurde. Ihre Entwicklung war eine Folge des Manhattan-Projektes, das die Konstruktion der ersten Atombombe zum Ziel hatte.

Nachfolgend wurden Monte-Carlo-Verfahren in immer mehr Gebieten erfolgreich eingesetzt, wie beispielsweise in der Simulation von Warteschlangen, in Verkehrsfluß-Analysen oder bei der Lösung logistischer Probleme usf.

Durch die weitläufige Anwendung derartiger Verfahren entstand ein schwer zu befriedigender Bedarf an *guten* Zufallszahlen, so daß im folgenden auf Methoden zu deren Erzeugung eingegangen werden soll:

in bislang ungekannten Mengen. Auf die Erzeugung von Zufallszahlen soll im folgenden weiter eingegangen werden:

[2]Hierbei stellt n die Länge des Startwertes dar.

[3]Unter dem Begriff der *Periode* eines Zufallsgenerators versteht man die kleinste Zahl $n \in \mathbb{N}$, für die gilt: $x_{n+i} = x_i \quad \forall i \in \mathbb{N}$.

2.1.1 Erzeugung von Zufallszahlen - ein Überblick

Die früheste eingesetzte Methode zur Erzeugung von Zufallszahlen waren **Würfel**. Von einfachen 6-flächigen Spielwürfeln bis hin zu exakt gearbeiteten Sonderformen, wie den ikosaedrischen Würfeln der *Japanese Standards Association*, die in 3-er Gruppen vertrieben wurden und so mit einem Wurf die Generierung einer dreistelligen Dezimalzahl erlaubten, wobei jeder Würfel zweimal die Ziffern 0 bis 9 aufwies.

Eines der größten Probleme bei der Anwendung von Würfeln ist die Notwendigkeit, durch ständige Kontrollen sicherstellen zu müssen, daß sich die Eigenschaften des *Zufallsgenerators* nicht durch Umwelteinflüsse (Verschleiß, etc.) verändert haben[4].

Zu Beginn unseres Jahrhunderts entstanden die ersten **Tabellenwerke**, die eine bislang ungekannte Menge an Zufallszahlen einer vergleichsweise breiten Öffentlichkeit zugänglich machten. Das erste derartige Tabellenwerk wurde von *Tippett* im Jahre 1927 erstellt und basierte auf der Verwendung von Zufallsziffern aus Volkszählungsdaten. Dieses Tabellenwerk verwandte 41600 solcher Ziffern, die in 10400 vierstellige Zufallszahlen zusammengefaßt wurden.

Weitere Tabellen dieser Art entstanden in den darauf folgenden Jahren:

Kadyrow: 50000 Stellen, ebenfalls auf Volkszählungsdaten beruhend.

Fisher und Yates: Erstmalige Verwendung von Ziffern aus Logarithmentafeln.

Kendall und Smith: Ausgangsdaten wurden mit Hilfe von Telefonbüchern gewonnen.

RAND: Dieses wohl berühmteste aller Tabellenwerke für Zufallszahlen entstand im Jahre 1955 und umfaßte *eine Million* Zufallsziffern, die auf Magnetband bzw. Lochkarte verfügbar waren.

Manche dieser Tabellenwerke, deren Zahlen gleichverteilt sein sollten, stellten sich erst einige Zeit nach ihrer Veröffentlichung als fehlerhaft heraus. So enthielten die Tabellen von *Fisher und Yates* bemerkenswerte Häufungen der Ziffer 6, die nachträglich per Hand korrigiert werden mußten.

Da der Bedarf an Zufallsziffern für Tabellen wie die letztgenannte nicht mehr aus herkömmlichen Datenquellen gedeckt werden konnte, mußten neue Wege zur Erzeugung von Zufallswerten gefunden werden. Kurzzeitig erfreuten sich physikalische Einrichtungen einer gewissen Beliebtheit, gestatteten sie es doch, Zufallszahlen in bislang ungekannter Menge zu erzeugen. Wie bereits erwähnt, kranken alle diese Verfahren an der Problematik, daß regelmäßig überprüft werden muß, ob sich die Charakteristika des zugrundeliegenden Meßobjektes nicht unzulässig geändert haben.

Drei realisierte Methoden seien beispielhaft angeführt:

[4]Dieses Problem tritt prinzipiell bei allen physikalischen Methoden zur Zufallszahlenerzeugung auf.

Rauschgeneratoren: Diese Methode basierte auf der Erzeugung von rosa Rauschen mit Hilfe spezieller Einrichtungen, wie z.B. Rauschdioden. Angewandt wurde ein derartiges Verfahren bei der Erstellung des **RAND**-Tabellenwerkes.

Radioaktive Zerfälle: Hierbei wird die Anzahl registrierter Zerfälle eines radioaktiven Präparates in einer konstanten Zeitspanne gemessen. Dieses Beispiel zeigt vielleicht am deutlichsten, wie wichtig die ständige Kontrolle der Eigenschaften eines solchen physikalischen Prozesses ist: Zum einen unterliegt das untersuchte radioaktive Präparat dem Einfluß seiner Halbwertszeit, zum anderen wirken sich auch die Alterungserscheinungen der verwandten Registriereinrichtungen (z.B. *Geiger-Müller*-Zählrohre o.ä.) in erheblichem Maße auf die Qualität der erzeugten Zufallszahlen aus[5].

Neonglimmlampen: Hier wurde das stochastische Zünden von Neon-Glimmlampen ausgenutzt. Leider blieb die Anwendung dieser Einrichtung auf ein einziges realisiertes Beispiel (*ERNIE* – *E*lectronic- *R*andom-*N*umber-Generator-*E*quipment-*I*ndicator) beschränkt, was nicht zuletzt auf das schlechte Zeitverhalten dieser Methode zurückzuführen war.

Arithmetische und algorithmische Verfahren: Allen solchen Verfahren ist gemeinsam, daß sie eine gewisse Anzahl *echter* Zufallswerte als Startwerte benötigen, die beispielsweise einer Tabelle entnommen werden können. Prinzipiell lassen sich hierbei zwei verschiedene Vorgehensweisen unterscheiden:

- Eines der ersten vorgeschlagenen und auch einfachsten arithmetischen Verfahren besteht in der Anwendung einer deterministischen Rechenvorschrift auf gegebene Zufallszahlen, wobei diese lediglich neu kombiniert werden, d.h. dieses Verfahren erzeugt keine *neuen* Werte. Ein Beispiel mag den grundlegenden Gedankengang verdeutlichen:

  Es seien zwei Eingabekanäle gegeben, die bei jedem Lesezugriff einen Zufallswert liefern, wobei diese zwei festen Listen entnommen werden, die zyklisch abgearbeitet werden. Nun kann mit Hilfe zweier Schleifen jeweils die Summe aus zwei gelesenen Zufallszahlen modulo N gebildet werden, wobei N üblicherweise der Größe des verwendeten Maschinenwortes entspricht.

  Weisen die beiden verwendeten Tabellen Längen von p_1 resp. p_2 auf, so erhält man eine Ausgabefolge der Länge $p_1 \cdot p_2$.

- **Iterative Verfahren:** Diesen Verfahren ist gemein, daß sie mit erheblich weniger Startwerten als das zuvor genannte auskommen (das ja lediglich seine Startwerte neu kombiniert). Sie haben die allgemeine Form:

$$x_{n+1} = f(x_n, x_{n-1}, \ldots, x_{n-m}), \qquad m \geq 0. \tag{2.4}$$

  Solche Verfahren erzeugen also iterativ neue Werte, die in den nachfolgenden Schritten ganz oder teilweise als Startwerte verwandt werden, so

[5] Trotz dieser offenkundigen Nachteile wurde die beschriebene Vorgehensweise einige Zeit lang in den Rechnern der *FACOM-128* Reihe des japanischen Herstellers Fujitsu eingesetzt.

daß die Periodenlänge der erzeugten Ausgabefolge nicht von der Anzahl der verwendeten Startwerte abhängt, was einen entscheidenden Vorteil gegenüber dem erstgenannten Verfahren darstellt.

Einen weiteren Ansatzpunkt für die Erzeugung von Pseudozufallszahlen stellt die Verwendung von Ziffernfolgen irrationaler Zahlen dar. Einige heutzutage verwendete Hashalgorithmen, wie beispielsweise *MD2* etc., verwenden für die Erstellung bestimmter Permutationsmatrizen Ziffernfolgen von π.

Es ist klar, daß sich wirkliche Zufallszahlen nur unter Zuhilfenahme physikalischer Einrichtungen mit den nachfolgend kurz aufgelisteten Vor- und Nachteilen erzeugen lassen:

Nachteile:

- Derartige Verfahren müssen einer ständigen Kontrolle unterliegen, um auch über längere Zeiträume hinweg brauchbare Ergebnisse zu liefern.
- Ihre Resultate sind prinzipiell nicht reproduzierbar.

Vorteil:

- *Große Periodenlänge* – derlei Verfahren weisen im Grunde keine Periode im eigentlichen Sinne auf.

Arithmetische Verfahren zur Erzeugung von Zufallszahlen weisen prinzipiell nicht die beiden genannten Nachteile der physikalischen Verfahren auf. Sie müssen jedoch auf das Bestehen statistischer Tests und ihre Periodenlänge hin untersucht werden.

Um *arithmetisch* erzeugte Zufallszahlen von aus physikalischen Experimenten gewonnenen unterscheiden zu können, werden sie meist als *Pseudozufallszahlen* bezeichnet, wie dies auch im folgenden geschieht.

2.2 Arithmetische Verfahren – ein historischer Überblick

2.2.1 Das Middle-Square-Verfahren

Dieses wohl früheste Beispiel eines arithmetischen Verfahrens zur Erzeugung von Zufallszahlen wurde im Jahre 1946 von *John von Neumann* vorgeschlagen.

Sei b die Basis des betreffenden Zahlensystems. Das *Middle-Square*-Verfahren beruht nun auf der Verwendung $2a$-stelliger Zahlen, die quadriert werden, wobei die mittleren $2a$ Stellen des resultierenden $4a$-stelligen Ergebnisses als Eingabe für den jeweils nächsten Schritt verwendet werden[6].

Formal läßt sich also schreiben:

$$x_{n+1} = \left\lfloor \frac{x_n^2}{b^a} \right\rfloor - \left\lfloor \frac{x_n^2}{b^{3a}} \right\rfloor b^{2a}. \tag{2.5}$$

[6]Der Originalvorschlag sah $a = 3$ vor.

Vorteil: Effizienz – für jeden generierten Wert wird nur eine einzige Multiplikation benötigt.

Nachteil: Extrem kurze Periodenlänge. Die Ausgabefolge stürzt entweder leicht in die 0 ab oder bildet Zyklen sehr kleiner Länge, wie Abbildung 2.1 (nach [9, S.40]) am Beispiel $a = 1$ verdeutlicht.

Um das Problem der kurzen Periodenlänge zu umgehen, muß ein derartiger Generator sehr häufig mit einer Zufallszahl neu initialisiert werden[7].

Aufgrund der nicht behebbaren Probleme, die der Middle-Square-Generator aufwies, wurden im Laufe der Zeit weitere Verfahren entwickelt, wobei die Klasse der *Kongruenzgeneratoren* die wichtigste Gruppe dieser Methoden darstellt:

2.2.2 Kongruenzgeneratoren

Die allgemeine Form eines Kongruenzgenerators beruht auf der rekursiven Anwendung einer Rechenvorschrift auf einen gegebenen Startwert, wobei die Ausgabe eines jeden Berechnungsschrittes aus dem Rest des betreffenden Funktionswertes zu einem festen Modul besteht.

Der allgemeine Kongruenzgenerator hat folgende Gestalt:

$$x_{n+1} = a_0 x_n + a_1 x_{n-1} + \ldots + a_n x_0 + b \bmod N. \tag{2.6}$$

Dieser Generator wird mit einem n-elementigen Startvektor initialisiert und erzeugt darauf basierend eine Folge von Pseudozufallszahlen. Wie bei allen derartigen Verfahren hängt die Qualität der generierten Ausgabefolge in extremem Maße von der geeigneten Wahl der Startparameter ab.

Im folgenden seien einige verbreitete Sonderformen der obigen allgemeinen Vorschrift angeführt:

Der additive Kongruenzgenerator:

$$x_{n+1} = x_n + x_{n-1} \bmod N \tag{2.7}$$

Dieser Generator wird zumeist als *Fibonacci*-Generator bezeichnet und in folgender verallgemeinerter Form angewandt:

$$x_{n+1} = x_n + x_{n-j} \bmod N, \qquad 0 \leq j \leq n. \tag{2.8}$$

Der multiplikative Kongruenzgenerator:

$$x_{n+1} = a x_n \bmod N \tag{2.9}$$

Bei dieser Methode läßt sich das k-te Folgenglied leicht wahlfrei ansprechen:

$$x_{n+k} = a^k x_n \bmod N. \tag{2.10}$$

Ausgehend von diesen beiden Kongruenzgeneratoren wurde die folgende Mischform entwickelt (sicherlich die am weitesten verbreitete und bestuntersuchte):

[7] Bemerkenswerterweise war trotz dieser Nachteile ein derartiger Generator bis vor wenigen Jahren am *CERN* im Einsatz.

Abbildung 2.1: Das Middle-Square-Verfahren für $a = 1$

Der lineare Kongruenzgenerator:

$$x_{n+1} = ax_n + b \bmod N \tag{2.11}$$

Durch wiederholte Anwendung dieser Iteration erhält man

$$x_{n+k} = a_k x_n + b_k \bmod N, \tag{2.12}$$

wobei für a_k und b_k gilt:

$$a_k = a^k \bmod N \qquad \text{und} \tag{2.13}$$

$$b_k = \frac{a^k - 1}{a - 1} b \bmod N. \tag{2.14}$$

Im folgenden sei noch eine Sonderform des allgemeinen Kongruenzgenerators beschrieben:

Der quadratische Kongruenzgenerator:

Diese Form eines Pseudozufallsgenerators weist die Gestalt

$$x_{n+1} = ax_n^2 + bx_n + c \bmod N \tag{2.15}$$

auf. Nach [2] sind die Ausgabefolgen dieses Verfahrens mit $N = 2^\omega$ und $a, b, c \in \mathbb{Z}_N$ und Startwert $x_0 \in \mathbb{Z}_N$, wobei $\omega \in \mathbb{N}$ mit $\omega \geq 2$ gilt, genau dann reinperiodisch[8] mit maximaler Periodenlänge, falls gilt[9]:

$$a \equiv 0 \pmod 2 \tag{2.16}$$

$$b \equiv c \equiv 1 \pmod 2 \tag{2.17}$$

$$a + b \equiv 1 \pmod 4. \tag{2.18}$$

Eine Sonderform dieser Klasse von Kongruenzgeneratoren stellt der in 3.1 ausführlicher behandelte *Blum-Blum-Shub*-Generator dar.

In neuerer Zeit werden auch Verallgemeinerungen dieser Generatorklasse untersucht und implementiert, die auf Polynomen höheren Grades beruhen, wie beispielsweise das Verfahren von *Micali-Schnorr*, auf das in Abschnitt 3.2 eingegangen wird.

2.3 Statistische Tests

Gleichverteilungstests: Eine der wesentlichsten Eigenschaften, die eine von einem Pseudozufallsgenerator erzeugte Ausgabefolge in der Mehrzahl aller Fälle aufweisen sollte, ist ihre *gleichmäßige Verteilung* im Intervall $[0, 1)$ [10].

[8] Der Begriff *reinperiodisch* kennzeichnet die Tatsache, daß ein solcher Generator keine Vorperiode aufweist.

[9] Hiermit ergibt sich, daß die durch (2.15) definierte Abbildung von $\mathbb{Z}_N$ nach $\mathbb{Z}_N$ eine Bijektion darstellt.

[10] Zufallszahlen, die gleichverteilt in diesem Intervall liegen, werden gemeinhin als *Standardzufallszahlen* bezeichnet.

Durch geeignete Normierung lassen sich die zu untersuchenden Ausgabefolgen stets so transformieren, daß sie in dem genannten Intervall zu liegen kommen. Gleichverteilungstests kommt heute die wohl größte Bedeutung bei der Untersuchung von Zufallsgeneratoren zu, wohingegen die meisten der nachstehend beschriebenen Testverfahren lediglich von historischem Interesse sind.

Werden die betrachteten Pseudozufallszahlen als Realisierungen unabhängiger, identisch $\mathrm{R}(0,1)$ verteilter Zufallsvariablen betrachtet, muß man bei der Beurteilung ihrer Gleichverteilungseigenschaft das statistische Problem lösen, eine Verteilungsaussage aufgrund dieser unabhängigen Realisierungen zu prüfen.

Hierfür wird in den meisten Fällen die sogenannte *Kolmogorov-Smirnov*-Statistik eingesetzt. Sie entspricht dem Abstand zwischen empirischer und angenommener Verteilungsfunktion, ist also in diesem Fall ein direktes Maß für die zu untersuchende Gleichverteilungseigenschaft des betrachteten Generators.

Als empirische Verteilungsfunktion wird zumeist eine Treppenfunktion verwendet, die wie folgt definiert wird (hierbei entspricht N der Anzahl der Meßpunkte x_i mit $1 \leq i \leq N$) und sich mit wachsendem N ihrer zugrundeliegenden Verteilungsfunktion annähert:

$$F_N(x) = \frac{1}{N} \cdot \#\{1 \leq \ i \leq N \mid x_i < x\}\,. \tag{2.19}$$

$F_N(x)$ wird als die zugrundeliegende empirische Verteilungsfunktion bezeichnet. Im Falle der Gleichverteilungstests gilt zumeist

$$x \in \mathbb{R} \text{ und } x_1, x_2, \ldots, x_N \in [0,1)\ . \tag{2.20}$$

Ist nun

$$F(x) = \begin{cases} 0 & : \quad x < 0 \\ x & : \quad 0 \leq x \leq 1 \\ 1 & : \quad x > 1 \end{cases} \tag{2.21}$$

die Verteilungsfunktion der $\mathrm{R}(0,1)$-Verteilung, dann vergleicht die Kolmogorov-Smirnov-Statistik die theoretische Verteilungsfunktion mit den gemessenen Daten und weist folgende Form auf:

$$\Delta_N = \sup_{x \in \mathbb{R}} |F_N(x) - F(x)| = \sup_{x \in [0,1]} |F_N(x) - x|. \tag{2.22}$$

Δ_N wird als *Nulldiskrepanz* von $x_1, x_2, \ldots, x_N$ bezeichnet. Eine Verallgemeinerung dieses Begriffes findet sich in der *Diskrepanz*, die als D_N geschrieben wird[11].

Sollen Auswertungen mit Hilfe des Kolmogorov-Smirnov-Tests vollzogen werden, ist zu beachten, daß die Stichprobengröße in den meisten Fällen sehr klein gewählt werden sollte, worauf [19, S.72f.] hinweist:

[11] Zur Durchführung von Gleichverteilungstests auf der Basis von Diskrepanzbetrachtungen siehe [2].

Der Kolmogorov-Smirnov-Test *... sollte tunlichst nur für Stichproben* $n < 100$ *verwandt werden [...]*

Weiterhin setzt das Verfahren eine stetige Variable, deren Verteilung empirisch ermittelt wird, voraus – eine Bedingung, auf die allerdings in manchen Fällen verzichtet werden kann, wie ein Zitat aus *Statistics* von *Robert L. Winkler* und *William L. Hays* zeigt:

Technically - the Kolmogorov-Smirnov test requires that the underlying variable be continuous, but it has been shown, that the violation of this assumption leeds only to very slight errors on the conservative side.

χ^2-Test: Dieses Testverfahren ermöglicht es, einen Vergleich zwischen erwarteten Häufigkeitskategorien und gemessenen Daten zu ziehen. Um nun Meßdaten mit erwarteten Häufigkeiten zu vergleichen, müssen zunächst die hierbei erwarteten Häufigkeitswerte ermittelt werden. Sowohl Meßdaten als auch die erwarteten Häufigkeiten werden über sogenannten *Zellen* betrachtet. Hierbei handelt es sich um Intervalle gleicher Länge, innerhalb derer alle aufgetretenen Ereignisse zusammengefaßt und gezählt werden.

E_i entspricht im folgenden der erwarteten Anzahl von Fällen in der i-ten Kategorie (Zelle) unter der betreffenden Nullhypothese (beispielsweise Gleichverteilung).

Analog hierzu ist O_i die gemessene Anzahl von Fällen in der i-ten Kategorie. Mit diesen Werten kann nun χ^2 für k Kategorien gemäß

$$\chi^2 = \sum_{i=1}^{k} \frac{(O_i - E_i)^2}{E_i} \tag{2.23}$$

ermittelt werden.

Stimmen die gemessenen Häufigkeiten mit den erwarteten gut überein, hat dies kleine Werte für χ^2 zur Folge, während große Werte auf eine schlechte Übereinstimmung hinweisen.

Poker-Tests: Dieses Verfahren ist heutzutage lediglich von historischem Interesse. Es bezieht sich auf die Ziffern des zu prüfenden Verfahrens, die in Fünfergruppen angeordnet werden. Innerhalb dieser Gruppen werden Kombinationen untersucht, die beim Pokerspiel von Bedeutung sind ([9, S.70]).

Lücken-Tests: Sei b die Basis des zugrundeliegenden Zahlensystems. Nach [9, S.71] existieren zwei grundlegende Varianten dieses Tests[12]:

Ziffernweise Betrachtung: In diesem Fall wird eine *Lücke* als Abstand zwischen zwei gleichen Ziffern definiert. Ist die Wahrscheinlichkeit für eine Lücke der Größe r gleich P_r, so ergibt sich:

$$P_r = \left(1 - \frac{1}{b}\right)^b \frac{1}{b}. \tag{2.24}$$

[12] Es wird Gleichverteilung der zu untersuchenden Ziffernfolgen angenommen.

Zahlenweise Betrachtung: Hierfür muß zunächst der Begriff des *Maximums* innerhalb einer Zahlenfolge als der Wert definiert werden, der von zwei kleineren Werten eingeschlossen ist. Analog hierzu ergibt sich der Begriff *Minimum*.

Eine Lücke wird nun als der Abstand zwischen zwei aufeinanderfolgenden Maxima bzw. Minima definiert.

Coupon-Sammler-Test: Die Bezeichnung dieses Tests leitet sich von dem in den Vereinigten Staaten weit verbreiteten System der Warengutscheine ab ([9, S.73]).

In einer Folge von Ziffern in einem Zahlensystem zur Basis b wird die Anzahl aufeinanderfolgender Ziffern bestimmt, die betrachtet werden müssen, um wenigstens eine der b möglichen Ziffern zu beobachten.

Wurde eine derartige Runde erfolgreich beendet, wird – ausgehend von der nächsten Ziffer der Folge – eine weitere gestartet.

Wird die Wahrscheinlichkeit, daß genau r Ziffern benötigt werden, um eine komplette Reihe aller b Ziffern zu erhalten, mit P_r bezeichnet, so ergibt sich unter der Voraussetzung, daß die Ziffern gleichverteilt sind[13]:

$$P_r = \frac{1}{b^{r-1}} \sum_{i=1}^{b-2} (-1)^i \binom{b-1}{i} (b-1-i)^{r-i}, \qquad r \geq b. \tag{2.25}$$

Visuelle Tests: Auch diese Methode ist lediglich von historischem Interesse, da sie doch einer tiefergehenden Exaktheit entbehrt und von ausgefeilteren analytischen Methoden abgelöst wurde.

Sie beruhte auf der Positionierung von Punkten auf einem graphischen Ausgabegerät mit Hilfe der erzeugten Zahlenfolge des betrachteten Pseudozufallsgenerators. Wurden hierbei Muster, wie beispielsweise Anhäufungen von Punkten in bestimmten Regionen oder Linienbildung[14] etc. beobachtet, so wurde das betreffende Verfahren als untauglich verworfen.

Im weiteren sei das Verfahren des linearen Kongruenzgenerators etwas eingehender beleuchtet, um die grundsätzlichen Fragestellungen und Methoden darzustellen:

2.4 Der lineare Kongruenzgenerator (LKG)

Wie bereits angesprochen, gehört diese Form eines Pseudozufallsgenerators zu den meistverbreiteten Verfahren, was zum einen auf die vergleichsweise leichte Implementierbarkeit, zum anderen auf sein gutes Laufzeitverhalten zurückzuführen ist.

Für mathematische Anwendungen ist dieser Generatortyp durchaus gut geeignet, da er eine Vielzahl statistischer Tests bezüglich der Verteilung der von

[13] Für den Fall $b = 10$ existieren Tabellen für P_r mit $r \in [10, 75] \subset \mathbb{N}$.

[14] Linienbildungen in diesem Sinne kommen natürlich bei jedem linearen Kongruenzgenerator vor – siehe auch 2.5.

ihm erzeugten Zahlenfolgen erfüllt. (Problematisch ist eine Gitterstruktur genannte Eigenschaft, auf die in 2.5 kurz eingegangen wird.)

Für kryptographische Anwendungen gilt dies hingegen nicht uneingeschränkt, da die von ihm erzeugten Ausgabefolgen in keiner Weise die in Bezug auf die Unvorhersagbarkeit gestellten Anforderungen erfüllen. Selbst bei Kenntnis nur eines kleinen Teils der erzeugten Zahlenfolge ist es möglich, sie in beide Richtungen vorherzusagen und somit verschlüsselte Texte in Klartext zu verwandeln. Dennoch ist er als einführendes Beispiel in die Problematik gut geeignet, so daß etwas näher auf ihn und seine Besonderheiten eingegangen werden soll (als Beispiel für die Betrachtung der Periode sei vereinfachend der Fall eines multiplikativen Kongruenzgenerators angenommen).

2.4.1 Die Periode des multiplikativen Kongruenzgenerators

Es sei stets $P \geq 5$ prim und $\mathbb{Z}_P^* = \{1, 2, \ldots, P-1\}$. Für einen Multiplikator $a \in \mathbb{Z}_P^*$ und einen Startwert $x_0 \in \mathbb{Z}_P^*$ ist ein multiplikativer Kongruenzgenerator durch eine Folge $(x_n)_{n\geq 0} \in \mathbb{Z}_P^*$ gegeben, die wie folgt rekursiv definiert ist:

$$x_{n+1} = ax_n \bmod P. \tag{2.26}$$

Allgemein gilt also für $n \geq 0$:

$$x_n \equiv a^n x_0 \pmod{P}. \tag{2.27}$$

Nach dem kleinen Fermatschen Satz gilt stets

$$a^{p-1} \equiv 1 \pmod{P} \qquad \forall\, a \in \mathbb{Z}_P^*, \tag{2.28}$$

woraus sich in diesem Fall sofort

$$x_{P-1} = x_0 \tag{2.29}$$

ergibt. Dies bedeutet, daß die Folgen des hier betrachteten vereinfachten linearen Kongruenzgenerators stets **reinperiodisch** sind, d.h. keine Vorperiode aufweisen.

Im folgenden sei π die Periodenlänge des zu untersuchenden Generators. Sie ist allgemein definiert durch

$$\pi = \min\{n \in \mathbb{N} \mid x_n = x_0\}. \tag{2.30}$$

Mit den obigen Betrachtungen folgt

$$\pi = \min\{n \in \mathbb{N} \mid a^n x_0 \equiv x_0 \pmod{P}\}, \tag{2.31}$$

d.h. aber

$$\pi = \min\{n \in \mathbb{N} \mid a^n \equiv 1 \pmod{P}\}. \tag{2.32}$$

$a \in \mathbb{Z}_P^*$ heißt *Primitivelement*, falls

$$a^n \not\equiv 1 \pmod{P} \qquad \forall\, 1 \leq n < P-1 \tag{2.33}$$

erfüllt ist. Insgesamt folgt also, daß die Periodenlänge des hier untersuchten Beispielgenerators maximal ist, falls a ein Primitivelement ist. In diesem Fall gilt

$$\pi = P - 1. \tag{2.34}$$

Nach den bisherigen Betrachtungen über die Periodenlänge des multiplikativen Kongruenzgenerators seien noch einige kurze Bemerkungen, den linearen Kongruenzgenerator der Form

$$x_{n+1} = ax_n + b \bmod N \qquad \text{mit } N \in \mathbb{N} + 1 \tag{2.35}$$

betreffend, angebracht:

2.5 $x_{n+1} = ax_n + b$ mod N

Wie bereits angemerkt, besteht dieser Generator für geeignet gewählte Parameter a, b, sowie N eine Reihe statistischer Tests, die ihn für eine Vielzahl von Anwendungen durchaus brauchbar erscheinen lassen. Problematisch ist allerdings folgende Eigenschaft, die als *Gitterstruktur* bezeichnet werden kann:

Werden je k konsekutiv aufeinanderfolgende Pseudozufallszahlen eines linearen Kongruenzgenerators verwandt, um einen Punkt im k-dimensionalen Raum zu setzen (hierbei liege jede Koordinate nach geeigneter Normierung beispielsweise zwischen 0 und 1), so neigen diese Punkte – entgegen ersten Erwartungen – nicht dazu, den Raum gleichmäßig auszufüllen und sind nicht voneinander unabhängig. Vielmehr kommen sie auf $k-1$-dimensionalen Hyperebenen zu liegen, wobei es (nach [16, S.214]) höchstens $\sqrt[k]{m}$ solcher Ebenen gibt. Betrachtet man folglich einen linearen Kongruenzgenerator mit Modul 32768 im Dreidimensionalen, so existieren im Bestfall 32 zweidimensionale Ebenen. Ein bekannter Hersteller von Großrechnern, dessen mitgelieferter Pseudozufallsgenerator verblüffend schlechte Eigenschaften in dieser Hinsicht aufwies, antwortete auf eine entsprechende Anfrage nach [16, S.215]:

We guarantee that each number is random individually, but we don't guarantee that more than one of them is random!

Eine Methode zur Vermeidung dieser Problematik wurde von *Knuth* [10] vorgeschlagen:

2.5.1 Verbesserung des linearen Kongruenzgenerators

Zur Erzeugung ganzzahliger Pseudozufallszahlen z im Intervall $[1, l]$, $l \in \mathbb{N}$, wird nach Knuth zur Vermeidung der eben erwähnten Gitterstruktur zweckmäßigerweise wie folgt vorgegangen:

Es wird ein Integer-Array der Länge l vereinbart, das im Initialisierungsschritt des Verfahrens mit Pseudozufallszahlen z des verwendeten linearen Kongruenzgenerators belegt wird, die (nach geeigneter Normierung) folgende Bedingung erfüllen:

$$z \in [1, l]. \tag{2.36}$$

Nach erfolgter Initialisierung werden nun paarweise neue (ebenso normierte) Zufallszahlen erzeugt, deren erste der Adressierung eines der obigen Arrayelemente dient, dessen Inhalt als Pseudozufallszahl des verbesserten Verfahrens ausgegeben wird. Die zweite wird im Anschluß hieran verwandt, um das so adressierte Element mit einem neuen Wert zu belegen.

Das eben geschilderte Verfahren hat sich in der Praxis bewährt und sollte in allen Fällen zur Anwendung kommen, in denen die Unabhängigkeit der einzelnen Folgenglieder unabdingbar ist, bzw. die oben beschriebenen Gitterstrukturen nicht akzeptabel erscheinen.

Eine weitere Schwachstelle linearer Kongruenzgeneratoren ist nach [16, S.215] die Tatsache, daß die niederwertigen Bits eines jeden Folgengliedes zumeist *weniger zufällig*[15] als die höherwertigen Bits sind. Aus diesem Grunde sollte man beispielsweise zur Erzeugung ganzzahliger Zufallszahlen im Intervall $[1, 10]$ stets die Anweisung

```
z = 1 + (int) (10. * ran ())
```

und nie

```
z = 1 + ((int) (1000000. * ran ()) mod 10)
```

anwenden, da in letzterem Fall lediglich die niederwertigen Bits eines jeden Ausgabeelementes zur Anwendung kommen.

Außer den beschriebenen Schwierigkeiten linearer Kongruenzgeneratoren in rein mathematischen Applikationen wurden weitere Schwachstellen bei Anwendungen in der Kryptologie aufgedeckt. Diese beziehen sich auf die Bedingung der *Unvorhersagbarkeit*. Werden durch Klartextraten Teile der Ausgabefolge des verwendeten Verfahrens erraten, stellt sich einerseits die Frage nach der Analyse des LKG bei *bekanntem* Modul, andererseits die nach der Aufdeckung seiner Parameter bei *unbekanntem* Modul.

Zur Verdeutlichung dieser Problematik sei der einfachste Fall der Analyse bei bekanntem Modul kurz betrachtet:

Da einer der Hauptaspekte bei der Implementierung eines normalen Pseudozufallsgenerators die Optimierung seines Laufzeitverhaltens ist, wird man in den meisten Fällen bestrebt sein, die Modulo-Additionen als solche gar nicht auszuführen, sondern den Modul so zu wählen, daß er beispielsweise der Wortlänge der zugrundeliegenden Hardware oder einem Vielfachen bzw. Teil hiervon entspricht, so daß die Annahme eines bekannten Moduls nicht allzu unrealistisch ist.

Nach der allgemeinen Iterationsvorschrift gilt

$$x_1 = ax_0 + b \bmod N \tag{2.37}$$

$$x_2 = ax_1 + b \bmod N, \tag{2.38}$$

mithin also

$$x_2 - x_1 = a(x_1 - x_0) \bmod N, \tag{2.39}$$

[15] So unmathematisch dieser Begriff sein mag.

woraus sich der Multiplikator sofort zu

$$a = \frac{x_2 - x_1}{x_1 - x_0} \bmod N \tag{2.40}$$

ergibt. Zu seiner Bestimmung war also lediglich die Kenntnis dreier aufeinanderfolgender Ausgabeelemente notwendig. Die Berechnung des Inkrements gestaltet sich ähnlich leicht zu

$$b = x_1 - ax_0 \bmod N. \tag{2.41}$$

Hiermit sind nun alle Parameter des betrachteten linearen Kongruenzgenerators bekannt, die von ihm generierte Ausgabefolge kann also beliebig vorhergesagt werden. Die benötigten bekannten Folgenglieder lassen sich in der Praxis beispielsweise durch Klartextraten etc. erhalten. Weiteres hierzu, sowie zur Analyse bei unbekanntem Modul findet sich ausführlich in [15, S.196ff.], worauf an dieser Stelle verwiesen sei.

Bemerkenswert ist, daß sich der lineare Kongruenzgenerator selbst bei Geheimhaltung eines Großteils der Bits der von ihm erzeugten Ausgabefolge unter Umständen analysieren läßt! Nach [15, S.206] kann der Modul mit *großer* Wahrscheinlichkeit bestimmt werden, wenn mehr als lediglich $\frac{2}{5}$ der Bits eines jeden Folgenelementes ausgegeben werden!

All dies läßt dieses Verfahren der Zufallszahlengenerierung für kryptographische Anwendungen ungeeignet erscheinen, da es potentiellen Angreifern nicht unmöglich ist, das Bildungsgesetz der zu Verschlüsselungszwecken gebildeten Ausgabefolge zu bestimmen, so daß andere Pseudozufallsgeneratoren entwickelt wurden, von denen einige wichtige Vertreter im folgenden behandelt werden.

Kapitel 3

Die Verfahren im Einzelnen

3.1 Der Blum-Blum-Shub–Generator

Der folgende Abschnitt beruht auf einem 1986 von L. Blum, M. Blum, sowie M. Shub [1] veröffentlichten Artikel, in dem erstmalig ein Pseudozufallsgenerator auf der Basis des Quadratrestproblems vorgeschlagen wurde.

3.1.1 Vorbemerkungen

Zunächst sollen einige allgemeine Betrachtungen angestellt werden, die für die Behandlung des Blum-Blum-Shub–Generators notwendig sind.

Definition 1:

Ist für $x, a, N \in \mathbb{N}$

$$a^2 \equiv x \pmod{N} \tag{3.1}$$

lösbar, so heißt x *Quadratrest* modulo N. Ist (3.1) hingegen nicht lösbar, heißt x entsprechend *quadratischer Nichtrest* modulo N.

Seien P, Q prim mit $P \neq Q$ und $P > 2$, $Q > 2$. Weiter seien $N = PQ$ und

$$\mathbb{Z}_N^* = \{x \in \mathbb{Z} \mid 0 < x < N,\ \mathrm{ggT}(x, N) = 1\}\,. \tag{3.2}$$

Dann ist die im folgenden mit QR_N bezeichnete Menge der Quadratreste modulo N eine multiplikative Untergruppe der Ordnung $\frac{\varphi(N)}{4}$ von $\mathbb{Z}_N^*$[1]. Jeder Quadratrest besitzt vier unterschiedliche Quadratwurzeln, $\pm x \bmod N$, sowie $\pm y \bmod N$ (siehe hierzu beispielsweise [15, S.214]).

Definition 2:

Eine Zahl N mit $N = PQ$ mit P und Q prim, $P \neq Q$ und $P \equiv Q \equiv 3 \pmod 4$ wird als *Blum-Zahl* bezeichnet.

[1] $\mathbb{Z}_N^*$ besitzt die Kardinalität $\varphi(N)$.

Lemma 1:

Sei N eine Blum-Zahl, d.h. es gilt (wie im folgenden stets) $P \equiv Q \equiv 3 \pmod 4$; dann besitzt jeder Quadratrest mod N **genau eine** Quadratwurzel, die selbst wieder Quadratrest ist und mit $\sqrt{x}$ bezeichnet wird.

Beweis:

Ein ausführlicher Beweis hierzu findet sich in [15, S.214].

Das Quadratrest-Problem

Seien P, Q prim mit $P \neq Q$, sowie $P > 2$, $Q > 2$ und $N = PQ$. Die Elemente aus $\mathbb{Z}_N^*$ sind durch das ihnen zugeordnete Jacobi-Symbol in zwei Hälften unterteilt: Die eine mit Jacobi-Symbol $(+1)$, im folgenden mit $\mathbb{Z}_N^*(+1)$ bezeichnet, die andere analog mit (-1) und Bezeichnung $\mathbb{Z}_N^*(-1)$.

Ein Viertel der Elemente aus $\mathbb{Z}_N^*$ sind wiederum Quadratreste (sie liegen alle in $\mathbb{Z}_N^*(+1)$). Das Quadratrest-Problem mit den Parametern N und x besteht nun in der Entscheidung, ob ein vorgegebenes $x \in \mathbb{Z}_N^*(+1)$ Quadratrest ist.

Die Quadratrest-Vermutung

Im folgenden sei $\mathbf{P}[N, x]$ ein beliebiger probabilistischer Algorithmus, der für gegebenes N und x, die jeweils eine Länge von n Bits, $n \in \mathbb{N}$, besitzen, eine 0 beziehungsweise 1 als Ausgabewert zurückliefert. Desweiteren seien $0 < \delta < 1$ eine Konstante, $t \in \mathbb{N}$ und N wie eben das Produkt zweier Primzahlen P und Q mit $P \equiv Q \equiv 3 \pmod 4$, $P > 2$, $Q > 2$.

Dann gilt für die Wahrscheinlichkeit, daß $\mathbf{P}[N, x]$ die Quadratrest-Eigenschaft von x korrekt bestimmt, für hinreichend großes $n \in \mathbb{N} + 1$ und alle Zahlen N – außer einem δ-Anteil hiervon – der Länge n, sowie zufällig ausgewähltes $x \in \mathbb{Z}_N^*(+1)$:

$$\frac{\sum\limits_{x \in \mathbb{Z}_P^*(+1)} \text{Prob}\left(\mathbf{P}[N, x] \text{ ist falsch}\right)}{\frac{\varphi(N)}{2}} < \frac{1}{n^t}. \tag{3.3}$$

Der chinesische Restsatz

Es seien $m_1, m_2, \ldots, m_k \in \mathbb{N}+1$, sowie $a_1, a_2, \ldots, a_k \in \mathbb{Z}$ mit $k \in \mathbb{N}+1$ gegeben. Gilt nun

$$m = m_1 m_2 \cdots m_k \tag{3.4}$$

und

$$\text{ggT}(m_i, m_j) = 1 \qquad \forall i \neq j, \tag{3.5}$$

so existiert genau ein $x \in [0 : m-1]$ mit folgender Eigenschaft:

$$x \equiv a_i \pmod{m_i}, \qquad i = 1, \ldots, k. \tag{3.6}$$

Der Beweis dieses Satzes findet sich beispielsweise in [7, S.295]; dort findet sich auch eine probate Methode, ein solches x konkret zu bestimmen:

Setze $n_i := \frac{m}{m_i}$. Hiermit gilt sofort $\mathrm{ggT}(n_i, m_i) = 1$, woraus folgt:

$$\exists x_i \in \mathbb{Z} \ : \ r_i := x_i n_i \equiv 1 \pmod{m_i}. \tag{3.7}$$

Für diese r_i gilt weiterhin

$$r_i \equiv 0 \pmod{m_j} \qquad \forall j = 1, \ldots, k, \ j \neq i. \tag{3.8}$$

Gelingt es also, diese x_i zu bestimmen, kann eine Lösung für (3.6) durch

$$x = \sum_{i=1}^{k} a_i r_i \bmod m \tag{3.9}$$

bestimmt werden.

Nach diesen Vorbetrachtungen kann nun das dem Blum-Blum-Shub–Generator zugrundeliegende Verfahren dargestellt werden:

3.1.2 Das Verfahren

Definition 3:

Sei

$$\mathcal{N} = \{N \in \mathbb{N} \mid N = PQ, P, Q \text{ prim}, |P| = |Q|, P \equiv Q \equiv 3 \pmod{4}\} \tag{3.10}$$

die Menge der sogenannten *Parameterwerte* (hierbei sei mit $|\bullet|$ die Größenordnung des betreffenden Wertes bezeichnet),

$$\mathcal{X}_N = \left\{x^2 \bmod N \mid x \in \mathbb{Z}_N^*\right\} \tag{3.11}$$

die Menge der Quadratreste mod N für $N \in \mathcal{N}$ und

$$\mathcal{X} = \bigcup_{N \in \mathcal{N}}{}^{\cdot} \ \mathcal{X}_N \tag{3.12}$$

der Startwertraum.

Weiterhin sei eine Transformation $T : \mathcal{X} \to \mathcal{X}$ definiert durch

$$T(x) = x^2 \bmod N \qquad \forall x \in \mathcal{X}_N. \tag{3.13}$$

T ist in polynomialer Zeit berechenbar und darüberhinaus eine Permutation auf $\mathcal{X}_N$ (siehe Lemma 1). Sei weiterhin ein ebenfalls in polynomialer Zeit berechenbares $B : \mathcal{X} \to \{0, 1\}$ definiert durch

$$B(x) = \mathrm{lsb}(x).^2 \tag{3.14}$$

$\langle \mathcal{X}, T, B \rangle$ definiert nun einen Pseudozufallsgenerator zur Basis 2 (vermöge B), der allgemein als x^2 mod N–Generator bzw. *Blum-Blum-Shub*–Generator (im

[2]Hierbei liefert $\mathrm{lsb}(x)$ (*least significant bit*) das niederwertigste Bit von x zurück – diese Funktion wird zuweilen – durchaus abweichend vom normalen Sprachgebrauch – auch mit $\mathrm{parity}(x)$ bezeichnet.

folgenden häufig auch nur *BBS-Generator*) bezeichnet wird und auf wiederholter Anwendung der Iteration

$$x_{i+1} = x_i^2 \bmod N \tag{3.15}$$

und Extraktion des jeweils untersten Bits[3] von x_{i+1} durch $B(x_{i+1})$ beruht. Die solchermaßen erzeugte Folge von Pseudozufallsbits sei mit $b_1, b_2, b_3, \ldots$ bezeichnet.

Beispiel 1:

Es sei $N = 7 * 11 = 77$. Der oben beschriebene Generator wird mit dem Startwert $x_0 = 9$ gestartet, was als Ausgabe folgende Werte liefert: $x_1 = 4, x_2 = 5, x_3 = 3, x_4 = 9, x_5 = 4, \ldots$. Die Periode π der solchermaßen erzeugten Ausgabefolge beträgt 4, die korrespondierende Bitfolge der $b_1, \ldots, b_4$ besitzt die Werte $0, 1, 1, 1$.

Es ist klar, daß die Bestimmung der Ausgabefolge $x_1, x_2, \ldots$ des BBS-Generators bei bekanntem N, sowie vorgegebenem Startwert x_0 vorwärts effizient möglich ist. In umgekehrter Richtung (d.h. Bestimmung von $x_{k-1}, x_{k-2}, \ldots, x_1$ ausgehend von einem vorgegebenen Wert x_k) kann sie jedoch nur bei Kenntnis der Primfaktoren P und Q von N berechnet werden, was die Grundlage zu einem auf dem BBS-Generator beruhenden *Public-Key-System* darstellt[4]:

Beispiel 2:

Im folgenden sei N (es gilt $N = PQ, P \equiv Q \equiv 3 \pmod 4$), P und Q sind von der gleichen Größenordnung, $P \neq Q$) veröffentlicht, wohingegen P und Q geheimgehalten werden. Zur Verschlüsselung einer Folge von Bits $\vec{m} = (m_1, m_2, \ldots, m_k)$ ist nun zunächst ein Startwert $x \in \mathbb{Z}_N^*$ geeignet auszuwählen, der modulo N quadriert wird, um einen Quadratrest x_0 zu erhalten. Mit den Startwerten (N, x_0) wird nun der BBS-Generator gestartet, um eine Folge $\vec{b} = (b_1, b_2, \ldots, b_k)$ zu erzeugen, mit deren Hilfe der Klartext $\vec{m}$ durch eine Exklusiv-Oder-Verknüpfung verschlüsselt werden kann:

$$\vec{c} = \vec{m} \oplus \vec{b} = (m_1 \oplus b_1, m_2 \oplus b_2, \ldots, m_k \oplus b_k). \tag{3.16}$$

Der $k+1$-te Ausgabewert x_{k+1} des Generators wird nun noch an den so gewonnenen Chiffre-Text $\vec{c}$ angehängt und zusammen mit diesem an den Empfänger der Nachricht übermittelt. Zu beachten ist, daß für die **Verschlüsselung** lediglich der Modul N, nicht jedoch seine Primfaktorzerlegung P, Q bekannt sein mußte, so daß N veröffentlicht und somit als *Public-Key* eingesetzt werden kann.

[3]Wie gezeigt werden wird, können auch mehrere Bits pro Iterationsschritt extrahiert werden, ohne die kryptographische Sicherheit zu gefährden.

[4]Das Konzept der *Public-Key-Kryptosysteme*, das den Schlüsseltausch zwischen den beteiligten Kommunikationspartnern überflüssig macht, wurde erstmals 1976 von *Diffie* und *Hellmann* vorgeschlagen.

Die **Entschlüsselung** dieses Chiffre-Textes erfordert nun die Fähigkeit, die Folge der Generator-Ausgabewerte ausgehend von dem mitübermittelten x_{k+1} in Rückrichtung zu bestimmen. Hierzu ist nur der Besitzer der Primzerlegung des öffentlichen Schlüssels N in der Lage, dessen Faktoren P und Q als *Private-Key* behandelt und geheim gehalten werden. Ausgehend von der restaurierten Folge $x_1, x_2, \ldots, x_k$ kann nun wieder die Bitfolge $\vec{b}$ und mit ihr effizient der Klartext $\vec{m}$ durch nochmalige bitweise Addition von $\vec{b}$ zu $\vec{c}$ modulo 2 bestimmt werden.

3.1.3 Folgenberechnung rückwärts

Im folgenden soll ein Verfahren zur Berechnung der Ausgabefolge des Generators in umgekehrter Richtung vorgestellt werden, wie es zur Entschlüsselung von Texten gemäß obigem Beispiel eingesetzt werden kann:

Satz 1:

Es existiert ein effizienter, deterministischer Algorithmus A, der für gegebenes N (N sei wie in Beispiel 2 definiert), dessen Primfaktorzerlegung in zwei Primzahlen P, Q mit $P \equiv Q \equiv 3 \pmod 4$ und einen beliebigen Quadratrest $x_0 \in \mathbb{Z}_N^*$ den zugehörigen, eindeutigen Quadratrest x_{-1} bestimmt, so daß gilt:

$$A(P, Q, x_0) = x_{-1} \tag{3.17}$$

mit

$$x_{-1}^2 \bmod N = x_0. \tag{3.18}$$

Beweis:

Nach Lemma 1 ist die Abbildung $f : QR_N \to QR_N$ mit $f(x) = x^2 \bmod N$ eine Bijektion. Der Beweis der Existenz eines solchen Algorithmus A sei im folgenden durch Angabe eines konkreten Beispiels gegeben:

Mit den in Lemma 1 genannten Eingaben führt A folgende Rechnung durch: Berechne ein

$$x_P = \sqrt{x_0} \bmod P \qquad \text{mit} \qquad x_p \in \mathbb{Z}_P^*, \tag{3.19}$$

für dessen Jacobi-Symbol gilt:

$$\left(\frac{x_P}{P}\right) = +1. \tag{3.20}$$

Die eigentliche Berechnung läßt sich zurückführen auf[5]

$$x_P = \sqrt{x_0} \bmod P = \pm x_0^{\frac{P+1}{4}} \bmod P. \tag{3.21}$$

Berechne analog x_Q zu

$$x_Q = \sqrt{x_0} \bmod Q = \pm x_0^{\frac{P+1}{4}} \bmod Q. \tag{3.22}$$

[5]Nach Voraussetzung gilt $P \equiv 3 \pmod 4$.

Mit Hilfe des euklidischen Algorithmus sind nun zwei Zahlen $u, v \in \mathbb{N}$ zu berechnen, so daß gilt:

$$Pu + Qv \equiv 1 \pmod{N}. \tag{3.23}$$

Nun kann

$$x_N = x_P Qv + x_Q Pu \bmod N \tag{3.24}$$

bestimmt werden, das $\sqrt{x_0} \bmod N$ und darüberhinaus ein echter Quadratrest in Bezug auf N ist, da es Quadratrest sowohl für P, als auch für Q ist.
□

Der obige Satz zeigt, daß die Kenntnis der Zerlegung von N in P und Q eine hinreichende Bedingung für die effiziente Berechnung der Folge der x_i in Rückrichtung ist. Andererseits läßt sich zeigen, daß die Fähigkeit zur Berechnung der Folgenglieder in dieser Richtung ausreicht, N zu faktorisieren:

Bemerkung 1:

Angenommen, die Folge $x_k, x_{k-1}, x_{k-2}, \ldots, x_1$ kann ausgehend von einem vorgegebenen x_{k+1} effizient berechnet werden, dann sei ein $x \in \mathbb{Z}_N^*$ mit Jacobi-Symbol $(\frac{x}{N}) = -1$ zufällig ausgewählt.

Nun wird $x_0 = x^2 \bmod N$ gesetzt und hiermit x_{-1} berechnet, was nach Voraussetzung möglich ist. Ausgehend hiervon kann nun N durch Berechnung von

$$\mathrm{ggT}(x + x_{-1}, N) = P \text{ oder } Q \tag{3.25}$$

faktorisiert werden (siehe [6, S.146]).

3.1.4 Zur kryptographischen Sicherheit

Nachdem der Begriff der kryptographischen Sicherheit bereits in Kapitel 2 eingeführt wurde, kann nun der Nachweis erbracht werden, daß der $x^2 \bmod N$–Generator unter der Annahme, daß die Quadratrest-Vermutung korrekt ist, in der vorgestellten Form kryptographisch sicher ist. Hierzu sind zunächst einige vorbereitende Definitionen und Betrachtungen notwendig, die das benötigte Handwerkszeug für den am Ende gegebenen Widerspruchsbeweis liefern:

Definition 4:

Sei ein ε mit $0 < \varepsilon \leq \frac{1}{2}$ gegeben. Dann hat eine in polynomialer Zeit berechenbare Prozedur $\mathbf{P}$ einen ε-*Vorteil* hinsichtlich des Ratens von $\mathrm{lsb}(x_{-1})$ für gegebenes $x_0 \in QR_N$, falls gilt:

$$\frac{\sum\limits_{x_0 \in QR_N} \mathrm{Prob}\left(\mathbf{P}[N, x_0] = \mathrm{lsb}(x_{-1})\right)}{\frac{\varphi(N)}{4}} \geq \frac{1}{2} + \varepsilon. \tag{3.26}$$

Analog hierzu läßt sich ein ε-Vorteil im Raten der Quadratrest-Eigenschaft definieren.

Den Zusammenhang zwischen diesen beiden Definitionen stellt das folgende Lemma her:

Lemma 2:

Ein wie oben definierter ε-Vorteil im Raten von $\mathrm{lsb}(x_{-1})$ für vorgegebenes $x_0 \in QR_N$ läßt sich in einen ε-Vorteil für das Raten der Quadratrest-Eigenschaft eines $x \in \mathbb{Z}_N^*(+1)$ umwandeln.

Beweis:

Im folgenden sei $x \in \mathbb{Z}_N^*(+1)$ das Element, dessen Quadratrest-Eigenschaft bestimmt werden soll. Setze $x_0 = x^2 \bmod N$.

Da $P \equiv Q \equiv 3 \pmod 4$ gilt, sind die in $\mathbb{Z}_N^*(+1)$ liegenden Quadratwurzeln von $x^2 \bmod N$ gleich x bzw. $(N-x)$. Da nun N als Produkt zweier Primzahlen P, Q mit $P > 2$ und $Q > 2$ ungerade ist, gilt

$$\mathrm{lsb}(x) \neq \mathrm{lsb}(N-x). \tag{3.27}$$

Nur eine dieser beiden Wurzeln ist ihrerseits wieder Quadratrest, d.h. gleich x_{-1}. Ebenso ist nur für eine von beiden das niederwertigste Bit gleich dem von x_{-1}, so daß allgemein folgt:

$$x \in QR_N \iff x = x_{-1} \iff \mathrm{lsb}(x) = \mathrm{lsb}(x_{-1}). \tag{3.28}$$

□

Das folgende Lemma von *Goldwasser* und *Micali* erlaubt es, einen ε-Vorteil im Raten der Quadratrest-Eigenschaft in einen $\frac{1}{2} - \varepsilon$-Vorteil hierfür zu verwandeln, was am Ende zum gesuchten Widerspruch zur Quadratrest-Vermutung führen und somit den Beweis der kryptographischen Sicherheit des BBS-Generators unter dieser Voraussetzung beschließen wird.

Lemma 3:

Ein ε-Vorteil bezüglich des Ratens der Quadratrest-Eigenschaft eines vorgegebenen $x \in \mathbb{Z}_N^*(+1)$ nach Definition 4 kann in einen $\frac{1}{2} - \varepsilon$-Vorteil hierfür konvertiert werden.

Beweisskizze:

Sei $\mathbf{P}[N, x]$ ein probabilistischer Algorithmus, der für N ($N = PQ, P, Q$ prim, $P \neq Q, P \equiv Q \equiv 3 \pmod 4$) und $x \in \mathbb{Z}_N^*(+1)$ in polynomialer Zeit die Quadratrest-Eigenschaft von x mit einem ε-Vorteil rät. Es können also in polynomialer Zeit die Elemente $x \in QR_N$, sowie $x \in \mathbb{Z}_N^*(+1) - QR_N$ getestet werden, indem zufällig Elemente $x \in \mathbb{Z}_N^*$ ausgewählt, quadriert und negiert (mod N) werden.

Bestimme hiermit zwei Konstanten A und B

$$A \approx \frac{\sum\limits_{x \in QR_N} \text{Prob}\left(\mathbf{P}[N,x]=1\right)}{\frac{\varphi(N)}{4}} \tag{3.29}$$

$$B \approx \frac{\sum\limits_{\substack{x \in \mathbb{Z}_N^*(+1) \\ x \notin QR_N}} \text{Prob}\left(\mathbf{P}[N,x]=1\right)}{\frac{\varphi(N)}{4}} \tag{3.30}$$

mit $A - B \geq 2\varepsilon$.

Nach diesen Vorbereitungen sei jetzt $x \in \mathbb{Z}_N^*(+1)$ ein konkretes Element, dessen Quadratrest-Eigenschaft mod N bestimmt werden soll. Wähle hierfür zufällig und gleichverteilt Werte $r \in \mathbb{Z}_N^*$ aus und berechne $y = xr^2 \bmod N$. Nun wird durch Nachprüfen von $\mathbf{P}[N,y] = 1$ jedes y auf seine Quadratrest-Eigenschaft getestet. Vergleiche im nächsten Schritt die so gewonnenen Anteile verwertbarer Ergebnisse mit A bzw. B, um auf diese Art und Weise den gewünschten, auf $\frac{1}{2} - \varepsilon$ verstärkten Vorteil im Raten der Quadratrest-Eigenschaft zu erhalten.
□

Der letzte vorbereitende Schritt zum Beweis der kryptographischen Sicherheit ist die Einführung des Begriffs des *Prädiktors*:

Definition 5:

Ein *Prädiktor* $\mathcal{P}$ ist ganz allgemein ein in polynomialer Zeit berechenbarer, probabilistischer Algorithmus, der nach Eingabe von N, $N = PQ$ mit $P > 2$, $Q > 2$ prim, sowie $P \equiv Q \equiv 3 \pmod 4$, sowie $b_1, b_2, \ldots, b_k$ mit $b_i \in \{0,1\}$ und polynomial durch die Größenordnung von N beschränktes k als Rückgabewert 0 resp. 1 liefert.

$\mathcal{P}$ hat einen ε-Vorteil im Vorhersagen der Ausgabesequenzen des BBS-Generators nach *links*, wenn für alle derartigen k (außer höchstens wenigen hiervon) analog zu Definition 4 gilt:

$$\frac{\sum\limits_{x \in QR_N} \text{Prob}\left(\mathcal{P}[N, b_1(x), b_2(x), \ldots, b_k(x)] = b_0(x)\right)}{\frac{\varphi(N)}{4}} \geq \frac{1}{2} + \varepsilon. \tag{3.31}$$

Hierbei stellt $b_i(x)$ jeweils das niederwertigste Bit von x_i im i-ten Iterationsschritt des BBS-Verfahrens dar, d.h.

$$b_i(x) = \text{lsb}(x^{2^i} \bmod N). \tag{3.32}$$

Nach all diesen Vorarbeiten ist nun das Rüstzeug vorhanden, den eigentlichen Beweis der kryptographischen Sicherheit des $x^2 \bmod N$-Generators nach [1] zu erbringen:

Satz 2:

Unter der Voraussetzung, daß die Quadratrest-Vermutung korrekt ist, stellt der Blum-Blum-Shub–Generator einen krpytographisch sicheren Pseudozufallsgenerator dar.

Sei also $\mathcal{P}$ ein probabilistischer, in polynomialer Zeit berechenbarer Prädiktor, δ eine Konstante mit $0 < \delta < 1$ und $t \in \mathbb{N}$. Dann hat $\mathcal{P}$ höchstens einen $\frac{1}{n^t}$-Vorteil in der Voraussage der Sequenz nach links für hinreichend großes n und alle Zahlen N der Länge n, außer einem δ-Anteil von diesen.

Beweisskizze:

Annahme: Es sei $\mathcal{P}$ ein Prädiktor mit einem ε-Vorteil in der Voraussage der Ausgabesequenz des BBS-Generators, d.h. es existiere ein Prädiktor mit einem ε-Vorteil im Raten von $\mathrm{lsb}(x_{-1})$ für gegebenes x_0.

Nach Lemma 2 läßt sich dieser Prädiktor $\mathcal{P}$ in einen Algorithmus zum Raten der Quadratrest-Eigenschaft mit ε-Vorteil umwandeln.

Dieser ε-Vorteil kann nun mit Hilfe von Lemma 3 in einen verstärkten $\frac{1}{2} - \varepsilon$-Vorteil verwandelt werden, was im Widerspruch zu der als korrekt angenommenen Quadratrest-Vermutung steht!
□

Analog zu dem nun bekannten x^2 mod N-Generator läßt sich eine modifizierte Version desselben entwickeln, die ebenfalls unter der Annahme einer korrekten Quadratrest-Vermutung kryptographisch sicher ist:

Der modifizierte x^2 mod N-Generator

Sei $N = PQ, P \equiv Q \equiv 3 \pmod 4, P > 2, Q > 2, P, Q$ prim. Somit gilt $N \equiv 1 \pmod 4$, d.h. sowohl $x \in \mathbb{Z}_N^*$, als auch $-x \in \mathbb{Z}_N^*$ haben dasselbe Jacobi-Symbol, jedoch ein verschiedenes niederwertigstes Bit, da $N \equiv 1 \pmod N$ gilt.

Wie bereits erwähnt, teilt also das niederwertigste Bit $\mathbb{Z}_N^*(+1)$ in zwei Hälften, was im nicht-modifizierten BBS-Generator verwendet wird. Analog hierzu läßt sich eine Funktion $\mathrm{msb}(x)$ definieren, die ebenfalls diese Teilung von $\mathbb{Z}_N^*(+1)$ vollzieht:

$$\mathrm{msb}(x) = \begin{cases} 0 & : \quad x < \frac{N-1}{2} \\ 1 & : \quad x \geq \frac{N-1}{2} \end{cases} \tag{3.33}$$

Genau einer der Werte x bzw. $-x$ ist wieder ein Quadratrest, d.h. analog zum nicht-modifizierten x^2 mod N-Generator läßt sich schließen, daß dieser modifizierte Generator unter der Voraussetzung, daß die Quadratrest-Annahme korrekt ist, kryptographisch sicher ist.

Folgerung

Durch Vereinigung dieser beiden Varianten des x^2 mod N-Generators läßt sich ein verwandter Generator erzeugen, der ebenfalls unter der Annahme einer korrekten Quadratrestvermutung kryptographisch sicher ist, jedoch in jedem Iterationsschritt 2 Ausgabebits – $\mathrm{lsb}(x)$ und $\mathrm{location}(x)$ – liefert. Offen ist die

Frage, wieviele Bits bei gegebener Modullänge in jedem Schritt extrahiert werden können, ohne die unverzichtbare kryptographische Sicherheit einzubüßen!

3.1.5 Die Periodenlänge des BBS-Generators

Der folgende Abschnitt befaßt sich mit der Periodenlänge des BBS-Generators. Das Wissen um diesen Parameter ermöglicht beispielsweise die konkrete Überprüfung, ob die von einem solchen Generator mit gegebenen Startwerten erzeugte Ausgabefolge hinreichend lang ist, um einen vorliegenden Datensatz wiederholungslos zu chiffrieren.

Definition 6:

Für einen gegebenen Quadratrest $x_0 \bmod N$ mit $N = PQ$ und $P \neq Q$, $P \equiv Q \equiv 3 \pmod 4$ sei die Periodenlänge der Ausgabesequenz $x_1, x_2, x_3, \ldots$ des mit den Parametern (N, x_0) gestarteten $x^2 \bmod N$-Generators mit

$$\pi(x_0)$$

bezeichnet.

Definition 7: (Die Carmichael λ-Funktion)

Sei $M = 2^e P_1^{e_1} \cdots P_k^{e_k}$, mit $P_1, \ldots, P_k$ prim, $P_i \neq P_j$ für $i \neq j$, $i, j \in \mathbb{N}$ und $e \in \mathbb{N}$.

Die *Carmichael-λ-Funktion* wird nun definiert durch:

$$\lambda(2^e) = \begin{cases} 2^{e-1} & : \quad e \in \{1, 2\} \\ 2^{e-2} & : \quad e > 2 \end{cases} \tag{3.34}$$

und

$$\lambda(M) = \mathrm{kgV}(\lambda(2^e), (P_1 - 1)P_1^{e_1 - 1}, \ldots, (P_k - 1)P_k^{e_k - 1}). \tag{3.35}$$

Bemerkung 2:

Für den nachfolgenden Beweis wird die Carmichaelsche Verallgemeinerung des Eulerschen Theorems benötigt, die an dieser Stelle kurz vorgestellt werden soll:

Die Eulersche φ-Funktion $\varphi(M)$ gibt die Anzahl der zu M relativ primen Elemente aus dem Intervall $[1 : M]$ an. Das Euler-Theorem besagt nun für $a \in \mathbb{Z} \setminus \{0\}$ und $M \in \mathbb{N} + 1$:

$$\mathrm{ggT}(a, M) = 1 \implies a^{\varphi(M)} \equiv 1 \pmod M. \tag{3.36}$$

Die Carmichaelsche Verallgemeinerung des Eulerschen Theorems

Seien nun M und a wie in Bemerkung 2 definiert. Dann besagt die Carmichaelsche Verallgemeinerung des Eulerschen Theorems:

$$\mathrm{ggT}(a, M) = 1 \implies a^{\lambda(M)} \equiv 1 \pmod M. \tag{3.37}$$

Satz 3:

Sei N wie stets, x_0 ein Quadratrest modulo N und $\pi = \pi(x_0)$ die Periode der Sequenz $x_0, x_1, x_2, \ldots$.

Dann gilt:

$$\pi(x_0) \mid \lambda(\lambda(N)). \tag{3.38}$$

Beweis:

Sei im folgenden durch

$$\mathrm{ord}_N x = \min\left\{i \in [1 : \varphi(m)] \mid a^i \equiv 1 \pmod{m}\right\} \tag{3.39}$$

die Ordnung von x mod N bezeichnet. Dann ist $\mathrm{ord}_N x$ ungerade für $x \in \mathbb{Z}_N^*(+1)$, da

$$\mathrm{ord}_N x_i = \mathrm{ord}_N x_{i+1} \tag{3.40}$$

– dies gilt, da $\mathrm{ord}_N x_{i+1} \mid \mathrm{ord}_N x_i$ und die Folge $x_0, x_1, x_2, \ldots$ zyklisch ist – und

$$\left.\begin{array}{r} 2^n \mid \mathrm{ord}_N x_i \\ 2^{n+1} \nmid \mathrm{ord}_N x_i \end{array}\right\} \Longrightarrow \left\{\begin{array}{r} 2^{n-1} \mid \mathrm{ord}_N x_{i+1} \\ 2^n \nmid \mathrm{ord}_N x_{i+1} \end{array}\right. \qquad \forall n \in \mathbb{N}. \tag{3.41}$$

Mit Carmichaels Erweiterung des Eulerschen Theorems folgt somit

$$2^{\lambda(\mathrm{ord}_N x_0)} \equiv 1 \pmod{\mathrm{ord}_N x_0}. \tag{3.42}$$

Da aber π die kleinste ganze Zahl mit $x_0 = x_0^{2^\pi} \bmod N$ ist, ist π auch die kleinste ganze Zahl mit $2^\pi \equiv 1 \pmod{\mathrm{ord}_N x_0}$, d.h.:

$$\pi \mid \lambda(\mathrm{ord}_N x_0). \tag{3.43}$$

Andererseits folgt aus $\mathrm{ord}_N x_0 \mid \lambda(N)\ \forall x_0 \in \mathbb{Z}_N^*$, daß $\lambda(\mathrm{ord}_N x_0) \mid \lambda(\lambda(N))$, so daß sich insgesamt ergibt:

$$\pi \mid \lambda(\lambda(N)). \tag{3.44}$$

□

Nachdem nun $\pi \mid \lambda(\lambda(N))$ gezeigt ist, stellt sich die Frage, unter welchen Bedingungen diese Relation auch in umgekehrter Richtung korrekt ist, um für diese Fälle eine konkrete Berechnungsmethode für die Periodenlänge des BBS-Generators zu erhalten:

Satz 4:

Seien N, π und x_0 wie in Satz 3. Mache folgende Voraussetzungen:

1. N sei so gewählt, daß gilt:

$$\mathrm{ord}_{\frac{\lambda(N)}{2}}(2) = \lambda(\lambda(N)). \tag{3.45}$$

2. x_0 sei ein geeignet gewählter Quadratrest, so daß

$$\mathrm{ord}_N x_0 = \frac{\lambda(N)}{2} \tag{3.46}$$

gilt.

Dann gilt $\lambda(\lambda(N)) \mid \pi$, woraus mit Satz 3

$$\lambda(\lambda(N)) = \pi \tag{3.47}$$

folgt.

Unter diesen Voraussetzungen ist die Berechnung von $\lambda(\lambda(n))$ folglich eine probate Methode, die Periodenlänge des $x^2 \bmod N$-Generators zu bestimmen.

Beweis:

Nach Voraussetzung 2 des Satzes ist $\frac{\lambda(N)}{2}$ die kleinste ganze Zahl, für die $x_0^{\frac{\lambda(N)}{2}} \bmod N = 1$ gilt. Da jedoch $x_\pi = x_0^{2^\pi} \bmod N = x_0$ gilt, ist $x_0^{2^\pi - 1} \bmod N = 1$. Daraus folgt $\frac{\lambda(N)}{2} \mid 2^\pi - 1$ und somit

$$2^\pi \bmod \left(\frac{\lambda(N)}{2}\right) = 1. \tag{3.48}$$

Nach Voraussetzung 1 des Satzes ist $\lambda(\lambda(N))$ die kleinste ganze Zahl, für die

$$2^{\lambda(\lambda(N))} \bmod \left(\frac{\lambda(N)}{2}\right) = 1 \tag{3.49}$$

erfüllt ist. Da aber nach (3.48) $2^\pi \bmod \left(\frac{\lambda(N)}{2}\right) = 1$ gilt, folgt hieraus

$$\lambda(\lambda(N)) \mid \pi. \tag{3.50}$$

□

Nun stellt sich abschließend noch die Frage, wie, bzw. mit welchem Aufwand, die für diesen Satz benötigten Bedingungen im Einzelnen zu erfüllen sind, um mit dergestalt gewählten Parametern des BBS-Generators dessen Periodenlänge bestimmen zu können.

Bemerkung 3:

Ein Großteil aller Quadratreste $x_0 \in \mathbb{Z}_N^*$ erfüllt die zweite Voraussetzung von Satz 4. Hingegen ist es ungleich schwerer, ein geeignetes N zu finden, das der ersten Bedingung genügt. Die Frage nach Aufwand und Methodik der Erfüllung dieser Voraussetzungen kann jedoch mit folgender Definition und dem anschließenden Satz beantwortet werden:

Definition 8:

Eine Primzahl P heiße *speziell*, wenn sie von folgender Gestalt ist:

$$P = 2P_1 + 1 \qquad \text{mit} \qquad P_1 = 2P_2 + 1. \tag{3.51}$$

Hierbei sind P_1 und P_2 zwei ungerade Primzahlen.

Eine Zahl $N = PQ$ heiße *speziell*, wenn sowohl P, als auch Q speziell sind und zudem $P \equiv Q \equiv 3 \pmod 4$ gilt[6].

Beispiel 3:

Sowohl $P = 23$ als auch $Q = 47$ sind spezielle Primzahlen nach obiger Definition. Da zudem $P \equiv Q \equiv 3 \pmod 4$ erfüllt ist, ist auch $N = PQ = 1081$ speziell.

Satz 5:

Sei N im Sinne obiger Definition speziell und 2 ein Quadratrest in Hinblick auf P_1 *oder* Q_1[7].

Dann gilt

$$\operatorname{ord}_{\frac{\lambda(N)}{2}}(2) = \lambda(\lambda(N)) \tag{3.52}$$

und somit $\lambda(\lambda(N)) = \pi(x_0)$ nach Satz 4 für geeignetes x_0.

Beweis:

Sei N also speziell, dann gilt

$$\lambda(N) = \operatorname{kgV}(2P_1, 2Q_1) = 2P_1Q_1 \tag{3.53}$$

und entsprechend

$$\lambda(\lambda(N)) = \operatorname{kgV}(2P_2, 2Q_2) = 2P_2Q_2. \tag{3.54}$$

Es gilt $\lambda(\frac{\lambda(N)}{2}) = \lambda(\lambda(N))$, woraus $\operatorname{ord}_{\frac{\lambda(N)}{2}}(2) \mid \lambda(\lambda(N))$ folgt, d.h.

$$\operatorname{ord}_{\frac{\lambda(N)}{2}}(2) \mid 2P_2Q_2. \tag{3.55}$$

Nun bleibt (mit Hilfe eines Widerspruchsbeweises) zu zeigen, daß darüberhinaus sogar $\operatorname{ord}_{\frac{\lambda(N)}{2}}(2) = 2P_2Q_2$ gilt.

Annahme: $\operatorname{ord}_{\frac{\lambda(N)}{2}}(2) \neq 2P_2Q_2$. Damit gilt entweder

$$\operatorname{ord}_{\frac{\lambda(N)}{2}}(2) = P_2Q_2 \tag{3.56}$$

oder

$$\operatorname{ord}_{\frac{\lambda(N)}{2}}(2) \mid 2P_2 \tag{3.57}$$

[6] Aus $P \neq Q$ folgt sofort $P_2 \neq Q_2$.

[7] Nach [1] ist diese Bedingung für etwa $\frac{3}{4}$ aller speziellen Zahlen erfüllt.

oder

$$\text{ord}_{\frac{\lambda(N)}{2}}(2) \mid 2Q_2. \tag{3.58}$$

Sei nun o.B.d.A.

$$\text{ord}_{\frac{\lambda(N)}{2}}(2) \mid 2P_2 \tag{3.59}$$

oder

$$\text{ord}_{\frac{\lambda(N)}{2}}(2) = P_2Q_2. \tag{3.60}$$

Dann lassen sich folgende zwei Fälle unterscheiden:

1. Fall: $\text{ord}_{\frac{\lambda(N)}{2}}(2) \mid 2P_2$:

Daraus folgt

$$2^{2P_2} \equiv 1 \left(\text{mod } \frac{\lambda(N)}{2}\right) \equiv 1 \text{ (mod } P_1Q_1). \tag{3.61}$$

Dies führt zu

$$2^{2P_2} \equiv 1 \text{ (mod } Q_1). \tag{3.62}$$

Aus dem kleinen Fermatschen Satz und $Q_1 = 2Q_2 + 1$ ergibt sich $2^{2Q_2} \equiv 1 \text{ (mod } Q_1)$, woraus

$$2^{\text{ggT}(2P_2,2Q_2)} \equiv 1 \text{ (mod } Q_1) \tag{3.63}$$

folgt. Das heißt

$$2^2 \equiv 1 \text{ (mod } Q_1). \tag{3.64}$$

Dies stellt einen **Widerspruch** dar, da nach Voraussetzung $Q_2 > 2$ und damit $Q_1 > 6$ gilt!

2. Fall: $\textbf{ord}_{\frac{\lambda(N)}{2}}(2) = P_2Q_2$**:**

Daraus folgt

$$2^{P_2Q_2} \equiv 1 \text{ (mod } P_1Q_1). \tag{3.65}$$

Hieraus ergibt sich

$$2^{P_2Q_2} \equiv 1 \text{ (mod } Q_1) \tag{3.66}$$

und damit

$$2^{Q_2} \not\equiv -1 \text{ (mod } Q_1), \tag{3.67}$$

da P_2 nach Voraussetzung ungerade ist.

Somit gilt

$$2^{\frac{Q_1-1}{2}} \not\equiv -1 \text{ (mod } Q_1). \tag{3.68}$$

2 ist also Quadratrest in Bezug auf Q_1. Analog hierzu ist 2 ebenfalls Quadratrest zum Modul P_1.

Widerspruch!

□

Ausgehend von diesen Ergebnissen lassen sich nun Algorithmen entwickeln, mit deren Hilfe die Periodenlänge $\pi(x_0)$, bzw. der Wert des i-ten Ausgabelementes x_i des BBS-Generators konkret bestimmt werden können. Wie im Kapitel über die Implementation der Verfahren deutlich wird, sind derlei Funktionen für den praktischen Einsatz nahezu unerläßlich.

Satz 6:

Es existiert ein effizienter, deterministischer Algorithmus A, der für gegebene $N, \lambda(N), \lambda(\lambda(N))$ und die Faktorisierung von $\lambda(\lambda(N))$ die Periode $\pi(x_0)$ des Blum-Blum-Shub–Generators für jeden Quadratrest $x_0 \in \mathbb{Z}_N^*$ bestimmt, d.h.

$$A[N, \lambda(N), \lambda(\lambda(N)), \text{Faktorisierung von } \lambda(\lambda(N)), x_0] = \pi(x_0). \tag{3.69}$$

Beweis:

Im folgenden sei stets $\pi = \pi(x_0)$. Klar ist:

$$x_i = x_0^{2^i} \bmod N \tag{3.70}$$

und

$$x_\pi = x_0^{2^\pi} \bmod N = x_0. \tag{3.71}$$

Hieraus und aus $\pi \mid \lambda(\lambda(N))$ (siehe Satz 3) folgt

$$x_0^{2^{\lambda(\lambda(N))}} \bmod N = x_0. \tag{3.72}$$

Nach Carmichaels Verallgemeinerung des Eulerschen Theorems (Bemerkung 2) gilt $a^{\lambda(N)} \equiv 1 \pmod N$, falls $\text{ggT}(a, N) = 1$ gilt. Dies ergibt

$$x_0^{\lambda(N)} \equiv 1 \pmod N. \tag{3.73}$$

Das heißt

$$x_0^{2^{\lambda(\lambda(N))} + k\lambda(N)} \bmod N = x_0, \tag{3.74}$$

und damit

$$x_0^{2^{\lambda(\lambda(N))} \bmod \lambda(N)} \bmod N = x_0. \tag{3.75}$$

Zur Bestimmung der Periodenlänge π kann nun wie folgt vorgegangen werden:

- Suche das größte $d \in \mathbb{N}$ mit $d \mid \lambda(\lambda(N))$, so daß gilt:

$$x_0^{2^{\lambda(\lambda(N))+d} \bmod \lambda(N)} \bmod N = x_0. \tag{3.76}$$

- Dann berechnet sich die Periodenlänge π zu:

$$\pi = \frac{\lambda(\lambda(N))}{d}. \tag{3.77}$$

□

Nachdem nun ein Verfahren zur Bestimmung der Periodenlänge angegeben wurde, interessiert noch die Möglichkeit, wahlfrei auf beliebige Elemente der produzierten Ausgabefolge zugreifen zu können, was für die Implementation eines auf diesem Verfahren beruhenden Public-Key-Systems von essentieller Bedeutung ist (siehe Abschnitt 4.2.3). Geeignete Methoden hierfür werden mit den beiden folgenden Sätzen zur Verfügung gestellt:

Satz 7:

Es existiert ein deterministischer, effizienter Algorithmus A, der für gegebenes $N, \lambda(N)$, jeden Quadratrest $x_0 \in \mathbb{Z}_N^*$ und beliebig vorgegebenes $i \in \mathbb{N}$ den Wert x_i berechnet, d.h.

$$A[N, \lambda(N), x_0, i] = x_i. \tag{3.78}$$

Beweis:

$$x_i = x_0^{2^i \bmod \lambda(N)} \bmod N. \tag{3.79}$$

□

Satz 8:

Es existiert ein deterministischer, effizienter Algorithmus A, der für gegebenes $N, \lambda(N)$, jeden Quadratrest $x_0 \in \mathbb{Z}_N^*$ und beliebig vorgegebenes $i \in \mathbb{N}$ das Element x_{-i} berechnet, d.h.

$$A[N, \lambda(N), x_0, i] = x_{-i}. \tag{3.80}$$

Beweis:

Nach dem Beweis von Satz 1 gilt

$$\sqrt{x_0} \bmod P = \pm x_0^{\frac{P+1}{4}} \bmod P. \tag{3.81}$$

Genau eine dieser beiden Wurzeln ist nun wieder ein Quadratrest. Daraus folgt

$$x_{-1} \bmod P = \begin{cases} x_0^{\frac{P+1}{4}} \bmod P & \text{oder} \\ -x_0^{\frac{P+1}{4}} \bmod P. \end{cases} \tag{3.82}$$

Analog hierzu ergibt sich

$$x_{-2} \bmod P = \pm\, x_0^{(\frac{P+1}{4})^2} \bmod P, \tag{3.83}$$

u.s.f. Wird dieses Verfahren fortgesetzt, erhält man letztlich

$$x_{-i} \bmod P = \pm\, x_0^{(\frac{P+1}{4})^i} \bmod P = \pm\, x_0^{(\frac{P+1}{4})^i \bmod (P-1)} \bmod P \tag{3.84}$$

und entsprechend

$$x_{-i} \bmod Q = \pm\, x_0^{(\frac{Q+1}{4})^i} \bmod Q = \pm\, x_0^{(\frac{Q+1}{4})^i \bmod (Q-1)} \bmod Q. \tag{3.85}$$

Mit Hilfe des chinesischen Restsatzes kann nun x_{-i} aus $x_{-i} \bmod P$ und $x_{-i} \bmod Q$ effizient berechnet werden.
□

3.2 Das Verfahren nach Micali-Schnorr

Dem eben behandelten Verfahren nach Blum-Blum-Shub ist der folgende Pseudozufallsgenerator in Hinblick auf seine Effizienz weit überlegen, da er im Unterschied zu diesem eine Vielzahl von Bits anstatt nur einiger weniger in jedem Iterationsschritt auszugeben in der Lage ist, ohne seine kryptographische Sicherheit einzubüßen.

Entwickelt wurde das Verfahren von *Micali* und *Schnorr*, auf deren Veröffentlichung [11] der anschließende Abschnitt beruht.

3.2.1 Der Generator

Zunächst wird das diesem Pseudozufallsgenerator zugrundeliegende Verfahren näher erläutert. Hierzu sei nun $P(x)$ ein Polynom vom Grade $d \geq 2$ mit Koeffizienten aus $\mathbb{N}$. Weiterhin sei $N \in \mathbb{N}$ ein Modul der Länge n Bits (somit gilt $2^{n-1} \leq N < 2^n$). Das Verfahren von Micali-Schnorr basiert nun auf einem allgemeinen Kongruenzgenerator der Gestalt

$$f(x) = P(x) \bmod N \quad \text{mit } x \in [1, M], \tag{3.86}$$

wie er schon in (2.6) erwähnt wurde.

Für die folgenden Betrachtungen gelte stets $M \in \mathbb{N}$ mit $M < N$, jedoch M hinreichend groß.

Die beiden Hauptanforderungen an diesen allgemeinen Generator sind einerseits die Gleichverteilung seiner Ausgabeelemente und andererseits seine kryptographische Sicherheit, woraus sich die folgenden Voraussetzungen ergeben:

1. Der verwendete Modul N muß schwer faktorisierbar sein, da mit Kenntnis der Primfaktoren von N (3.86) invertiert werden kann[8].

2. Das Intervall $[1, M]$ muß genügend groß gewählt werden, um nicht durch vollständiges Durchsuchen des Intervalls den verwendeten Startwert der auf f beruhenden Iteration auffinden zu können. Desweiteren muß $\frac{P(x)}{N}$ durch geeignete Wahl von M für fast alle $x \in M$ hinreichend groß sein, um Abbildung (3.86) nicht invertieren zu können.

3. $P(x)$ darf kein quadratisches Polynom sein. Ist $P(x) = Q(x)^2$ für ein beliebiges Polynom Q, so gilt für sein Jacobi-Symbol

$$\left(\frac{P(x)}{N}\right) = 1 \quad \forall\, x, \tag{3.87}$$

während für ein zufälliges und gleichverteilt aus $[1, N]$ ausgewähltes y gilt:

$$\text{Prob}\left(\left(\frac{y}{N}\right) = 1\right) = \text{Prob}\left(\left(\frac{y}{N}\right) = -1\right). \tag{3.88}$$

[8] Auf dieser Grundlange wäre allerdings eine Erweiterung des im folgenden beschriebenen Verfahrens zu einem vollständigen Public-Key-System denkbar; ähnlich zu dem im vorangegangenen Abschnitt beschriebenen System auf der Grundlage des $x^2 \bmod N$-Generators.

Mit Hilfe von (3.87) und (3.88) ließe sich unter diesen Voraussetzungen ein Algorithmus entwerfen, der es gestattet, in polynomialer Zeit die Ausgabe des auf (3.86) beruhenden Generators (siehe (3.95)) von echten, gleichverteilten Zufallszahlen $y \in [1, N]$ zu unterscheiden!

Ebenfalls darf $P(x)$ keine lineare Transformation der Gestalt $P(x) = aQ(x)^2 + b$ eines quadratischen Polynoms darstellen, da in diesem Falle aus $P(x)$ mod N wieder $Q(x)$ mod N zurückgewonnen werden kann, so daß im Anschluß daran analog zum vorangegangenen Absatz verfahren werden kann.

Entsprechend den obigen Bedingungen wird nun N zufällig und gleichverteilt aus der Menge

$$S_n = \{N \in \mathbb{N} | N = PQ, P \text{ und } Q \text{ prim}, P \neq Q\} \tag{3.89}$$

ausgewählt. Hierbei gilt für die Längen der Werte P und Q:

$$|P| = \left\lfloor \frac{n}{2} \right\rfloor \tag{3.90}$$

$$|Q| = \left\lceil \frac{n}{2} \right\rceil. \tag{3.91}$$

Da P und Q somit von derselben Größenordnung sind, wird die Faktorisierbarkeit von N erschwert – hierdurch wird dem ersten Punkt der gestellten Anforderungen entsprochen.

Um Punkt 2 obiger Bedingungen Rechnung zu tragen, wird $M \in \mathbb{N}$ nun proportional zu $2^{\lfloor \frac{2n}{d} \rfloor}$ gewählt, d.h. M ist proportional zu $N^{\frac{2}{d}}$ für alle $N \in S_n$.

In [11] wird für $P(x)$ nun eine ganz bestimmte Klasse von Polynomen untersucht, die die oben genannten Anforderungen an $P(x)$ erfüllen. Es handelt sich um die sogenannten *RSA-Polynome*:

Das von *Rivest*, *Shamir* und *Adleman* im Jahre 1978 entwickelte RSA-Verfahren basiert auf der multiplikativen Gruppe

$$\mathbb{Z}_N^* = \{x \pmod N \mid \text{ggT}(x, N) = 1\}, \tag{3.92}$$

wobei $N = PQ$ mit P und Q prim und $P,\ Q > 2$ gilt (die Gruppenordnung von $\mathbb{Z}_N^*$ ist $\varphi(N)$).

Ein RSA-Polynom $f(x)$ hat folgende allgemeine Gestalt:

$$f(x) = x^d \bmod N,\ d, x, N \in \mathbb{N}. \tag{3.93}$$

Für den Exponenten d muß darüber hinaus

$$\text{ggT}(d, \varphi(N)) = 1 \tag{3.94}$$

erfüllt sein, womit Punkt 3 der oben genannten Bedingungen für die Sicherheit des Verfahrens erfüllt ist.

Der Pseudozufallsgenerator nach Micali-Schnorr beruht nun auf folgender Iteration:

$$x_{i+1} = x_i^d \bmod N \text{ mit } \mathrm{ggT}(d, \varphi(N)) = 1,\ x, d \in \mathbb{N}. \tag{3.95}$$

Hierfür wird zunächst ein $\lfloor \frac{2n}{d} \rfloor$ Bits langer Startwert x_0 zufällig aus dem Intervall $[1, 2^{\lfloor \frac{2n}{d} \rfloor} - 1]$ gewählt, der durch (3.95) in einen n Bits langen Ausgabewert umgewandelt wird.

Von diesen n Ausgabebits eines jeden Iterationsschrittes werden nun die oberen $\lfloor \frac{2n}{d} \rfloor$ Bits als Startwert für die nächste Iteration verwendet, während die verbleibenden $n - \lfloor \frac{2n}{d} \rfloor$ unteren Bits die Ausgabe dieses Schrittes darstellen (ein Vorgehen, das Abbildung 3.1 verdeutlicht).

Zu den Sicherheitsbetrachtungen des oben beschriebenen Pseudozufallszahlengenerators findet sich in der Arbeit von Micali und Schnorr folgende Hypothese als Kernpunkt:

Hypothese 1:

Sei $\mathbb{N} \ni d \geq 3$, $d \bmod 2 = 1$. Für zufälliges $N \in S_n, n \in \mathbb{N}$ mit $\mathrm{ggT}(d, \varphi(N)) = 1$ und für alle M proportional zu $2^{\lfloor \frac{2n}{d} \rfloor}$ gilt:

Die Gleichverteilung auf $[1, N]$ ist von keinem in polynomialer Zeit berechenbaren Algorithmus A von der Verteilung von $x^d \bmod N$ für zufälliges $x \in [1, M]$ unterscheidbar (siehe auch (2.1)).

Mit Hilfe des folgenden Lemmas nach *Alexi*, *Chor*, *Goldreich* und *Schnorr* (1984) wird im Verlauf der langwierigen Beweisführung in [11] der Bezug zwischen obiger Hypothese und der Sicherheit des RSA-Verfahrens hergestellt:

Lemma 4:

Seien $d, N \in \mathbb{N}$ mit $\mathrm{ggT}(d, N) = 1$. Dann kann jeder probabilistische Algorithmus A, der für die Eingabe $x^d \pmod N$ einen ε-Vorteil im Raten des niederwertigsten Bits der Nachricht x besitzt, in einen probabilistischen Algorithmus A' zur Entschlüsselung beliebiger RSA-Chiffretexte umgewandelt werden.

Zur Frage, inwieweit die Größe des Startwertintervalls $[1, M]$ ausschlaggebend für die Sicherheit dieses Verfahrens ist, wird in [11] gezeigt, daß der maximale Wert für M von $2^{\lfloor \frac{2n}{d} \rfloor} - 1$ bis auf $N \cdot 2^{-\lceil \log\ n \rceil}$ verkleinert werden kann, ohne die geforderte kryptographische Sicherheit einzubüßen.

Eine weitere Einschränkung auf beispielsweise lediglich $\lfloor \frac{n}{d} \rfloor$ Bits lange Startwerte ist allerdings nicht möglich, da für derartige x stets $x^d \leq N$ gilt, und somit RSA-Chiffretexte $x^d \bmod N$ für diesen Fall leicht dechiffrierbar sind.

Die folgende Definition führt einen anschaulichen Begriff für die Eigenschaft, Pseudozufallszahlengenerator zu sein, ein:

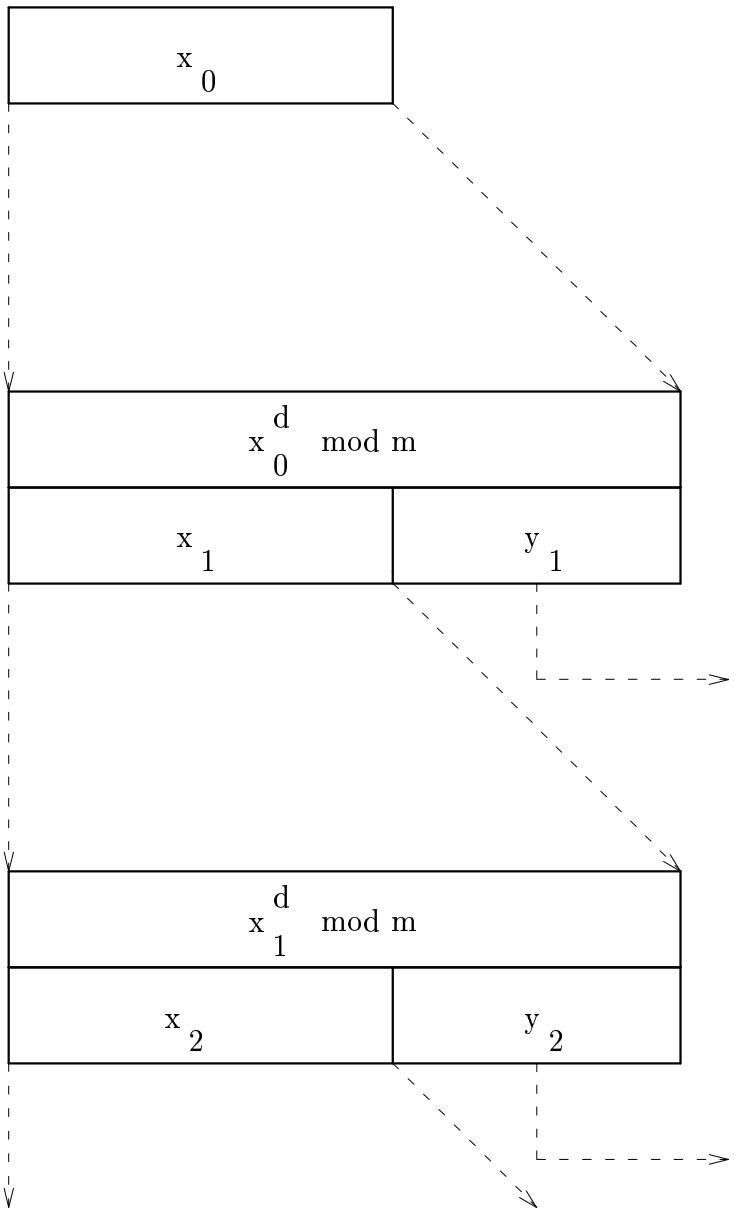

Abbildung 3.1: Das Verfahren nach *Micali-Schnorr*

Definition 9:

Ein Polynom $P(x)$ besitzt die *Generatoreigenschaft*, falls für zufälliges $N \in S_n$, für alle $M \in \mathbb{N}$ proportional zu $2^{\lfloor \frac{2n}{d} \rfloor}$ und zufälliges $x \in [1, M]$ die Werte von $P(x)$ mod N in $[1, N]$ pseudozufällig sind.

Das Ende der langwierigen Schlußfolgerungen aus [11] zum Nachweis der kryptographischen Sicherheit des beschriebenen Verfahrens bildet folgende Aussage:

Bemerkung 4:

RSA-Polynome $P(x) = x^d$ mit $\mathrm{ggT}(d, \varphi(N)) = 1$ und $d \geq 3$, $d, N \in \mathbb{N}$, besitzen die in Definition 9 eingeführte Generatoreigenschaft. Für $M = 2^{\lfloor \frac{2n}{d} \rfloor} - 1$ expandieren diese zufällige Startwerte $x \in [1, M]$ in pseudozufällige Zahlen aus dem Intervall $[1, N]$.

Der nachfolgende Abschnitt vertieft das Konzept des Micali-Schnorr–Generators:

3.2.2 Der sequentielle Polynomial-Generator

Wie bereits kurz angesprochen wurde, werden die höchstwertigen $\lfloor \frac{2n}{d} \rfloor$ Bits eines jeden Iterationsschrittes der Funktion (3.95) als Startwert für den jeweils nächsten Schritt verwandt, während die verbleibenden unteren $n - \lfloor \frac{2n}{d} \rfloor$ Bits ausgegeben werden. Im folgenden seien diese einzelnen Iterationsstartwerte im i-ten Schritt mit x_i bezeichnet, wohingegen $\mathrm{Out}(x_i) \in \{0,1\}^{n - \lfloor \frac{2n}{d} \rfloor}$ die zugehörige Ausgabebitkette der betreffenden Stufe repräsentiert.

Diese Form des Polynomgenerators nach Micali-Schnorr wird allgemein auch als *sequentieller Polynomgenerator*[9] bezeichnet. Eine Nomenklatur, die von folgender Definition aufgegriffen und erweitert wird:

Definition 10:

Die *k-Ausgabe* des sequentiellen Polynomgenerators sei mit

$$\mathrm{SPG}_{k,P}(x, N) = \prod_{i=1}^{k} \mathrm{Out}(x_i) \tag{3.96}$$

bezeichnet. Sie stellt die Konkatenation aller Ausgaben der k ersten Schritte des Verfahrens dar.

Für diesen Begriff der k-Ausgabe zeigt [11] nun folgenden Satz:

[9] Diese Bezeichnung unterscheidet die eben besprochene Version von der in Anhang E eingeführten parallelisierten Variante dieses Pseudozufallsgenerators.

Satz 9:

Das Polynom $P(x)$ besitze die Generatoreigenschaft wie in Definition 9. Dann gilt für zufälliges $N \in S_n$, zufälliges $x \in [1, 2^{\lfloor \frac{2n}{d} \rfloor} - 1]$ und polynomial beschränktes k:

Die k-Ausgabe des sequentiellen Polynomgenerators ist pseudozufällig.

Der in [11] gegebene Beweis zu dieser Aussage gestaltet sich sehr geradlinig: Nach Bemerkung 4 ist für zufälliges $N \in S_n$ und ebenfalls zufälliges $x_1 \in [1, 2^{\lfloor \frac{2n}{d} \rfloor} - 1]$ der Wert von $P(x_1)$ mod N pseudozufällig.

Damit ist auch $\mathrm{Out}(x_1) \in \{0,1\}^{n - \lfloor \frac{2n}{d} \rfloor}$ und $x_2 \in [1, 2^{\lfloor \frac{2n}{d} \rfloor} - 1]$ pseudozufällig. Da $P(x)$ nach Voraussetzung die Generatoreigenschaft besitzt und x_2 pseudozufällig ist, gilt dies auch für

$$z := (\mathrm{Out}(x_1)\mathrm{Out}(x_2), x_3) = (\mathrm{SPG}_{2,P}(x_1, N), x_3). \tag{3.97}$$

Im weiteren Verlauf dieser Beweisführung wird nun ein in polynomialer Zeit berechenbarer Test-Algorithmus T für $P(x_1)$ mod N bzw. $P(x_2)$ mod N entworfen, mit dessen Hilfe z untersucht wird. Ist dieses mit einem ε-Vorteil von einer zufälligen Bitkette unterscheidbar, so kann entweder $P(x_1)$ mod N oder $P(x_2)$ mod N mit einem $\frac{\varepsilon}{2}$-Vorteil zurückgewiesen werden.

Ausgehend hiervon wird nun durch Induktion nach k gezeigt, daß auch

$$(\mathrm{SPG}_{k,P}(x_1, N), x_{k+1}) \tag{3.98}$$

pseudozufällig für jedes feste k ist. Diese Schlußfolgerung wird auf polynomial beschränktes k erweitert, indem Testalgorithmen angenommen werden, die jeweils $P(x_1)$ mod N mit einem $\frac{\varepsilon}{k}$-Vorteil zurückweisen können.

3.3 Das Verfahren nach Impagliazzo-Naor

Wie bereits die beiden im vorangegangenen betrachteten Verfahren nach Blum-Blum-Shub bzw. Micali-Schnorr beruht eine Vielzahl an Kryptosystemen auf der Unlösbarkeit gewisser zahlentheoretischer Probleme, wie der Faktorisierung, dem Problem des diskreten Logarithmus, etc., was jedoch keineswegs den Schluß nahelegen soll, es seien keine anderen Grundlagen für die Konstruktion sicherer Verschlüsselungssysteme denkbar.

Problematisch bei den meisten Verfahren auf der Basis solcher zahlentheoretischer Fragestellungen ist nicht zuletzt ihr Laufzeitverhalten, das in vielen Fällen als ineffizient bezeichnet werden muß, setzen sie doch unter anderem zu bearbeitende Wertebereiche voraus, die weit jenseits aller Maschinengegebenheiten einer digitalen Rechenanlage liegen.

Ein weiterer Punkt, der nicht gänzlich unbeachtet bleiben sollte, ist die – zugegebenermaßen sehr unwahrscheinliche – Möglichkeit, daß in Zukunft praktikable Algorithmen zur Lösung dieser grundlegenden zahlentheoretischen Probleme entwickelt und somit die auf ihnen beruhenden kryptographischen Verfahren ihrer Grundlage entzogen werden, so daß nicht zuletzt aus sicherheitstechnischen

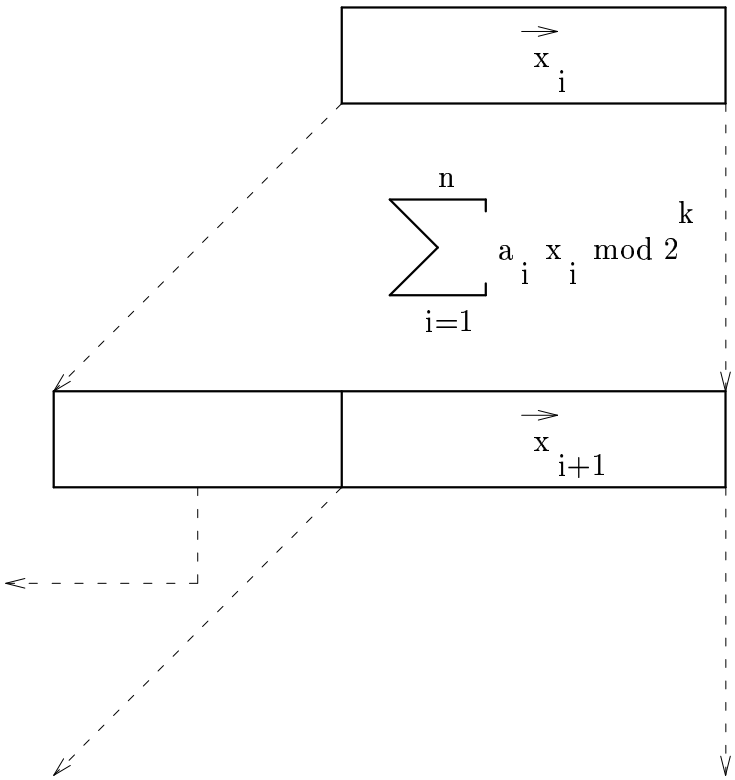

Abbildung 3.2: Das Verfahren nach *Impagliazzo-Naor*

Gesichtspunkten die Entwicklung und Verwendung von Chiffriermethoden, die auf anderen Verfahren beruhen, anzustreben ist.

Eine Alternative zu diesen Verfahren wurde bereits im Jahre 1978 von *Merkle* und *Hellmann* in Form eines Kryptosystems auf Basis des Knapsack-Problems vorgeschlagen, auf das hier jedoch nicht näher eingegangen werden soll[10].

Das in diesem Abschnitt vorgestellte Verfahren von *Impagliazzo* und *Naor* (siehe [8]) stellt keine Fortentwicklung oder Erweiterung dieser Verfahren dar, sondern gibt Algorithmen zur Erzeugung perfekter Zufallszahlen bzw. Hashfunktionen auf Basis des Knapsack-Problems an. Ein Vorteil, den dieses System beispielsweise dem Verfahren von Micali-Schnorr gemein hat, ist die Fähigkeit, in jedem Iterationsschritt mehrere Ausgabebits zu erzeugen.

Im folgenden sei nun einführend das Knapsack-Problem beschrieben, um die Voraussetzungen für das Verfahren von Impagliazzo-Naor zu schaffen:

3.3.1 Das Knapsack-Problem

Seien $n \in \mathbb{N}$ Zahlen $\vec{a} = (a_1, a_2, \ldots, a_n)$, $a_i \in (\mathbb{N}+1)$, für $i = 1(1)n$ mit einer Länge von jeweils $l(n) \in \mathbb{N}$ Bits, sowie eine Zahl $T \in \mathbb{N}$ gegeben. Nun ist ausgehend hiervon ein $\vec{x} = (x_1, x_2, \ldots, x_n) \in \{0,1\}^n$ zu bestimmen, so daß gilt:

$$T = \vec{a}\vec{x} = \sum_{i=1}^{n} a_i x_i. \tag{3.99}$$

Diese Problemstellung wird im allgemeinen *0-1-Knapsack-Problem* genannt und mit $K(\vec{a}, T)$ bezeichnet. Hierbei heißt $\vec{a}$ *Knapsack* und $\vec{x}$ – im Falle seiner Existenz – *Lösung* von $K(\vec{a}, T)$.

Der hierzu notwendige Lösungsaufwand wird nicht durch die Einschränkung vermindert, daß alle Additionen Modulo $2^{l(n)}$ ausgeführt werden, so daß sich das Problem auch wie folgt darstellen läßt:

Es sei die Funktion

$$f(\vec{a}, \vec{x}) = \vec{a}, \sum_{i=1}^{n} a_i x_i \bmod 2^{l(n)} \tag{3.100}$$

zu invertieren, wobei die $a_1, \cdots, a_n$ die festen Parameter des Verfahrens (analog beispielsweise zum RSA-Modul) darstellen. Hierbei stellt f die Konkatenation von $\vec{a}$ und $\sum_{i=1}^{n} a_i x_i \bmod 2^{l(n)}$ dar, wodurch die Voraussetzung für seine iterative Anwendung geschaffen werden, da unter diesen Gegebenheiten stets f auf seine vorige Ausgabe angewandt werden kann.

f ist dann eine Abbildung, die einen n Bits langen Startwert $\vec{x}$ auf eine $l(n)$ Bits lange Bitkette abbildet, wobei für die im folgenden betrachteten, auf dieser Grundlage beruhenden Pseudozufallsgeneratoren stets $l(n) > n$ gelte[11].

[10] Weiteres hierzu findet sich beispielsweise in [7, S.216ff.].

[11] $l(n) < n$ hat zur Folge, daß die Funktion die Länge ihres Startwertes reduziert, wodurch beispielsweise Hash-Funktionen implementiert werden können. Hierfür sei jedoch auf [8] verwiesen.

Schreibe auch[12]

$$f_{\vec{a}}(\vec{x}) = \vec{a}, \sum_{i=1}^{n} a_i x_i \bmod 2^{l(n)}. \tag{3.101}$$

Das Verfahren von Impagliazzo-Naor beruht nun auf der iterativen Anwendung der oben beschriebenen Funktion f. Mit einem geeigneten Startwert $\vec{x}$ wird die Summation gewissermaßen gesteuert, da nur Summanden betrachtet werden, denen an entsprechender Position des Bitverktors $\vec{x}$ ein gesetztes Bit zugeordnet ist. Die unteren n Bits des solchermaßen erhaltenen Resultats werden als neuer Steuervektor $\vec{x}$ für den nächsten Iterationsschritt verwandt, während die verbleibenden $n - l(n)$ oberen Bits ausgegeben und als Pseudozufallszahlen genutzt werden können.

Der folgende Abschnitt zeigt einige Bedingungen auf, die an einen geeigneten Parametersatz $\vec{a}$ für das obige Verfahren gestellt werden müssen, um seine Sicherheit zu gewährleisten.

3.3.2 Parameterwahl

Trotz der NP-Vollständigkeit des Knapsackproblems, ist nicht gewährleistet, daß es nicht unter gewissen Voraussetzungen mit Hilfe probabilistischer Algorithmen mit praktikablem Zeitverhalten für eine nicht vernachlässigbare Anzahl von Fällen lösbar wäre.

So wurde beispielsweise nach [8] für den Spezialfall $l(n) > n^2$ von *Legarias, Odlyzko* und *Brickell* ein solcher Algorithmus angegeben, der in fast allen Fällen zu einer Lösung des Problems führt.

Allein durch dieses Beispiel ist eine Art obere Schranke für die Wahl von $l(n)$ gegeben, die unter Wahrung der Sicherheit des Verfahrens nicht überschritten werden darf. Andererseits darf $l(n)$ auch nicht zu klein gewählt werden, um ein vollständiges Durchsuchen des Parameterraumes zu unterbinden. Der Auswahl eines geeigneten Parametersatzes muß folglich viel Aufmerksamkeit geschenkt werden, um die kryptographische Sicherheit der abgeleiteten Verfahren zu gewährleisten.

Nach [8] ist als obere Schranke $l(n) \leq c \cdot n$ mit $\frac{1}{3} < c \leq 1.5$ als durchaus sicher anzusehen, wobei für die Verwendung des Verfahrens als Pseudozufallsgenerator die Konstatne c stets größer 1 gewählt werden muß. Die Grundlage für die Verwendung des iterativ angewandten f mit gegebenem Startwert als Pseudozufallsgenerator bildet der folgende Satz aus [8]:

Satz 10:

Angenommen, das Knapsack-Problem mit $l(n) = (1 + c)n$ für $c > 0$ ist eine Einwegfunktion (siehe (D.1)), so ist es gleichzeitig ein Pseudozufallsgenerator.

Beweisskizze:

Die in [8] ausgeführte Beweisskizze geht vereinfachend von einer Primzahl p als Modul der Summation aus. Weiterhin wird vorausgesetzt, daß die Parameter

[12] Für alle folgenden Betrachtungen sei der Parametersatz $\vec{a}$ fest.

zufällig und gleichverteilt aus dem Intervall $[0, p-1]$ ausgewählt werden.

Annahme: f ist nicht pseudozufällig. (Ziel der anschließenden Überlegungen ist ein auf dieser Annahme beruhender Widerspruchsbeweis).

Für die folgenden Betrachtungen aus [8] muß zunächst der Begriff des inneren Produktes eingeführt werden:

Definition 11:

$r \cdot x$ heißt *inneres Produkt* modulo 2 für $x, r \in \{0,1\}^n$, falls gilt:

$$r \cdot x = \begin{cases} 1 \text{ falls die Anzahl der } i \text{ mit } r_i = x_i = 1 \text{ ungerade ist,} \\ 0 \text{ sonst.} \end{cases} \tag{3.102}$$

Hierbei bezeichnet r_i bzw. x_i das i-te Bit von r resp. x.

Lemma 5:

Nach *Goldreich* und *Levin* gilt nun für eine Einwegfunktion f, für jeden in polynomialer Zeit berechenbaren Algorithmus A und alle Polynome P:

$$\text{Prob}\left(A(f(x), r) = r \cdot x\right) < \frac{1}{2} + \frac{1}{P(n)} \tag{3.103}$$

für alle n, außer endlich vielen.

Der Widerspruchsbeweis wird auf folgendes hinauslaufen: Sind r und $f_{\vec{a}}(s)$ (3.101) bekannt, kann ein Test-Algorithmus verwandt werden, um das innere Produkt $r \cdot s$ mit einer Wahrscheinlichkeit größer gleich $\frac{1}{2} + \frac{1}{P(n)}$ vorherzusagen, wodurch mit Lemma 5 ein Widerspruch zu dem als Einwegfunktion vorausgesetzten f entsteht[13].

Der Test-Algorithmus:

Der vorgeschlagene Testalgorithmus erhält als Eingabe $\vec{a} = a_1, a_2, \ldots, a_n \in [0, p-1]$, sowie einen vorgegebenen Wert $T \in [0, p-1]$, der das Resultat einer Summation über einer Teilmenge der a_i darstellen kann. Seine Ausgabe ist mit einer Wahrscheinlichkeit von $\frac{1}{2} + \frac{1}{P(n)}$ gleich 1, falls T wirklich die Summe einer zufällig ausgewählten Teilmenge der a_i darstellt. Der Tester liefert hingegen eine 1 mit einer Wahrscheinlichkeit von genau $\frac{1}{2}$, falls T direkt zufällig aus $[0, p-1]$ gewählt wurde und bietet somit eine Möglichkeit, Ausgaben des Pseudozufallsgenerators von echten Zufallszahlen zu unterscheiden.

Dieser Test-Algorithmus wird nun dazu eingesetzt werden, die Größe der Schnittmenge der durch r und s ausgewählten Teilmengen der a_i und somit auch die Parität dieser Menge vorherzusagen. Hierzu wird folgender Vorhersagealgorithmus konstruiert:

[13] Beschreiben r und s Teilmengen der a_i, kommt dem inneren Produkt $r \cdot s$ folgende Bedeutung zu: Enthält die Schnittmenge der durch r und s ausgewählten Elemente a_i eine ungerade Anzahl von Elementen, gilt $r \cdot s = 1$, sonst 0.

Der Vorhersagealgorithmus:

Nach Eingabe von $f_{\vec{a}}(s)$ und r wählt die zu konstruierende Rechenvorschrift zunächst ein $k \in \{0, \ldots, n\}$, das die geratene Größe der Schnittmenge darstellt, sowie ein $x \in \{0, \ldots, p-1\}$ aus.

Nun wird für $i = 1(1)n$ folgende Zuweisung durchgeführt:

$$b_i := \begin{cases} a_i + x, & \text{falls } r_i = 1 \text{ gilt,} \\ a_i & \text{sonst.} \end{cases} \tag{3.104}$$

Anschließend wird der obige Tester mit Eingabe der b_i und $f_{\vec{a}}(s) + kx \bmod p$ gestartet. Gibt er eine 1 aus, wird die Parität von k zurückgeliefert, sonst ihre Negation.

Nach [8] sagt dieser Algorithmus das gesuchte innere Produkt mit einer Wahrscheinlichkeit von mindestens $\frac{1}{2} + \frac{1}{nP(n)}$ voraus, so daß hiermit der gesuchte Widerspruch erzielt wurde. □

3.4 DES, IDEA und MDx im OFB-Mode

Die in den drei vorangegangenen Abschnitten beschriebenen Verfahren wurden konkret für die Implementation von Pseudozufallszahlengeneratoren entwickelt. Dennoch können auch andere Verfahren und Funktionen für die Erzeugung von Pseudozufallszahlen eingesetzt werden, wenn dies zugegebenermaßen auch in vielen Fällen nicht den ursprünglichen Intentionen der Entwickler entspricht.

So eignen sich beispielsweise Blockchiffreverfahren wie das DES- oder IDEA-System hervorragend für den Einsatz als Pseudozufallsgenerator – ein Anwendungsgebiet, für das diese Funktionen aufgrund einiger ihrer Voraussetzungen sogar ausnehmend gut geeignet sind. Probleme, wie die bereits in Abschnitt 2.5 angesprochene Gitterstruktur linearer Kongruenzgeneratoren, sind beispielsweise nicht zu erwarten, ist doch das Ziel eines jeden Verschlüsselungssystems, in den Chiffretextdaten keinerlei Informationen über den zugrundeliegenden Klartext preiszugeben. Im folgenden sei lediglich das DES-System als Beispiel für eine Blockchiffre ausführlich beschrieben; die Funktionsprinzipien des ihm nicht unähnlichen IDEA-Verfahrens finden sich beispielsweise in [14].

Die als *OFB*-Modus[14] bezeichnete Betriebsart beruht auf der iterativen Anwendung eines Verschlüsselungssystems (DES, etc.) auf seine eigene Ausgabe – der Zustand eines solchen Systems ist einzig und allein von dem eingesetzten Schlüssel, sowie dem initialen Startwert der Iteration abhängig.

3.4.1 Das DES-Verfahren

Der *Data-Encryption-Standard* (kurz *DES*) stellt eine Weiterentwicklung der sogenannten *Lucifer-Chiffre* (siehe hierzu [14]) dar und wurde im Jahre 1977 vom National Bureau of Standards (NBS) in Zusammenarbeit mit *IBM* entworfen und genormt [12][13].

[14] *O*utput-*F*eed*b*ack-Modus

Es handelt sich um ein Verschlüsselungsverfahren, das auf Datenblöcken einer festen Länge von 8 Bytes arbeitet. Jeweils ein solcher Block wird unter Verwendung eines (ebenfalls acht Bytes langen) Schlüssels verarbeitet und in einen entsprechenden Chiffretextblock umgewandelt.

Das DES-Blockchiffreverfahren setzt sich prinzipiell aus den im folgenden näher beschriebenen drei Grundeinheiten zusammen[13][7, S.114ff.]:

Schlüssel-Aufspaltung:

Der sogenannte *Masterkey* des DES-Algorithmus weist, wie bereits erwähnt, eine Länge von 64 Bits auf, von denen jedoch lediglich 56 Bits konkret als Schlüssel genutzt werden können, da den 8 jeweils höchstwertigen Bits eine Paritätsfunktion zukommt, so daß sie nicht frei wählbar sind. In diesem Schritt werden nun die verwendbaren 56 Bits des Masterkeys in 16 verschiedene Teilschlüssel (*Sub-Masterkeys*) mit einer Länge von jeweils 48 Bits aufgeteilt. Als oberstes Prinzip hierbei gilt, daß niemals Bits des Masterkeys verändert oder miteinander kombiniert werden - sie werden lediglich mit Hilfe geeigneter Permutationen aufgeteilt.

Aus Sicherheitsgründen muß gewährleistet sein, daß diese 16 Teilschlüssel paarweise verschieden sind, was durch Auswahl jeweils verschiedener Bit-Teilmengen des Masterkeys erreicht wird. Es sei $\vec{k} = (k_1, k_2, \ldots, k_{64})$ der verwendete Masterkey, wobei das jeweils 8-te Bit eines jeden Bytes (d.h. $k_{i=8\ (8)\ 64}$), wie schon gesagt, ein Paritätsbit darstellt. Diese werden durch Anwendung des sogenannten *PC1*-Operators (permuted choice 1) entfernt, die verbleibenden 56 Bits zugleich einer initialen Permutation unterworfen:

$$\vec{l} = \mathrm{PC1}(k). \tag{3.105}$$

Der so erhaltene Bitvektor $\vec{l}$ wird nun in eine linke ($\vec{c}_0$) und eine rechte Hälfte ($\vec{d}_0$) aufgespalten. Aus dem entstandenen Paar $(\vec{c}_0, \vec{d}_0)$ werden die benötigten 16 Teilschlüssel $(\vec{c}_i, \vec{d}_i),\ i = 1\ (1)\ 16$ rekursiv bestimmt:

$(\vec{c}_i, \vec{d}_i)$ erhält man aus $(\vec{c}_{i-1}, \vec{d}_{i-1})$ durch zirkuläre Linksshifts der beiden Hälften $\vec{c}_{i-1}$ bzw. $\vec{d}_{i-1}$ um jeweils eine bzw. zwei Bitpositionen, wobei jede Hälfte für sich allein geshiftet wird. Die Anzahl der auszuführenden Shifts ist hierbei wie folgt festgelegt: Außer für die Iterationen 1, 2, 9 und 16, die aus nur einem Shift bestehen, werden stets 2 Shifts ausgeführt.

Anschließend wird der effektive Teilschlüssel $\vec{k}_i$ aus $(\vec{c}_i, \vec{d}_i)$ durch Entfernen der Bits 9, 18, 22, 25, 35, 38 und 43 erzeugt. Die so erhaltenen 48 Bits werden noch einer letzten Permutation *PC2* (permuted choice 2) unterworfen und bilden hiernach den gesuchten Teilschlüssel.

Die Chiffrierfunktion:

Der zu verschlüsselnde 64 Bit-Block $\vec{x}$ wird zunächst einer *Initialpermutation* IP unterworfen:

$$\vec{x}' = \mathrm{IP}(\vec{x}). \tag{3.106}$$

Hieran schließt sich folgende Verschlüsselungsiteration an: $\vec{x}'$ wird in einen linken Block $\vec{l}$ und einen rechten Block $\vec{r}$ mit einer Länge von jeweils 32 Bits

aufgespalten und nachstehender Iteration unterworfen:

Setze

$$\vec{l}_0 = \vec{l} \tag{3.107}$$
$$\vec{r}_0 = \vec{r} \tag{3.108}$$

und führe folgende Rechnung durch, wobei die $\vec{k}_i$ die bereits erwähnten 16 Sub-Masterkeys darstellen:

$$\vec{l}_i = \vec{r}_{i-1} \tag{3.109}$$
$$\vec{r}_i = \vec{l}_{i-1} \oplus \mathrm{f}(\vec{r}_{i-1}, \vec{k}_i) \qquad \text{für } i = 1 \text{ (1) } 16. \tag{3.110}$$

An dieser Stelle sei bemerkt, daß zur Verwürfelung (*scrambling*) und Verschlüsselung der Eingabebits prinzipiell 12 derartige Runden ausreichen – die zusätzlichen 4 Runden dienen der Erhöhung der Sicherheit des Verfahrens.

Auf den nach 16 Runden erhaltenen Ergebnisbitvektor $\vec{y} = \vec{r}_{16}\vec{l}_{16}$ wird nun noch die zu IP inverse Permutation IP^{-1} angewandt, so daß der eigentliche Ausgabeblock $\vec{o}$ die Gestalt

$$\vec{o} = \mathrm{IP}^{-1}(\vec{y}) \tag{3.111}$$

aufweist. Zur Vervollständigung der Beschreibung des DES-Verfahrens muß noch die bereits verwandte Chiffrierfunktion f eingeführt werden:

Die Funktion f:

Für das bereits in (3.110) angewandte f gilt allgemein

$$\mathrm{f} : \{0,1\}^{32} \times \{0,1\}^{48} \longrightarrow \{0,1\}^{32}. \tag{3.112}$$

Sei $\vec{k}_i$ der Sub-Masterkey im i-ten Schritt und $\vec{r}_{i-1}$ die rechte Hälfte der Ausgabe der $i-1$-ten Runde des Iterationsverfahrens[15]. $\vec{r}_{i-1}$ wird durch eine lineare Expansionsabbildung E, die ausgewählte Bits ihrer Eingabe dupliziert, auf einen 48 Bit-Block $\vec{z}$ erweitert:

$$\vec{z} = \mathrm{E}(\vec{r}_{i-1}). \tag{3.113}$$

Im Anschluß hieran werden $\vec{k}_i$ und $\vec{z}$ binär modulo 2 zueinander addiert:

$$\vec{a}_{i-1} = \vec{k}_i \oplus \mathrm{E}(\vec{r}_{i-1}). \tag{3.114}$$

Der so erzeugte Vektor $\vec{a}$ wird nun in 8 Blöcke $(\vec{a}'_1, \vec{a}'_2, \ldots, \vec{a}'_8)$ mit einer Länge von jeweils 6 Bits zerlegt, wobei $\vec{a}$ die Konkatenation der $a_{i=1 \text{ (1) } 8}$ darstellt.

Im folgenden Bearbeitungsschritt, der das eigentliche Herz des gesamten Verfahrens darstellt, werden diese 6-Bit-Blöcke auf jeweils 4 Bits reduziert. Dies vollzieht sich mit Hilfe der 8 sogenannten *S-Boxen*[16] $(\mathrm{S}_1, \ldots, \mathrm{S}_8)$, wobei jede

[15] Beachte, daß $\vec{r}_{i-1}$ eine Länge von 32 Bits, $\vec{k}_i$ eine von 48 Bits aufweist.

[16] Diese S-Boxen stellen nicht nur den kryptographisch wichtigsten Teil des gesamten Verfahrens dar, sondern gleichzeitig auch den größten Kritikpunkt, da ihre Entwurfskriterien nie vollständig veröffentlicht wurden!

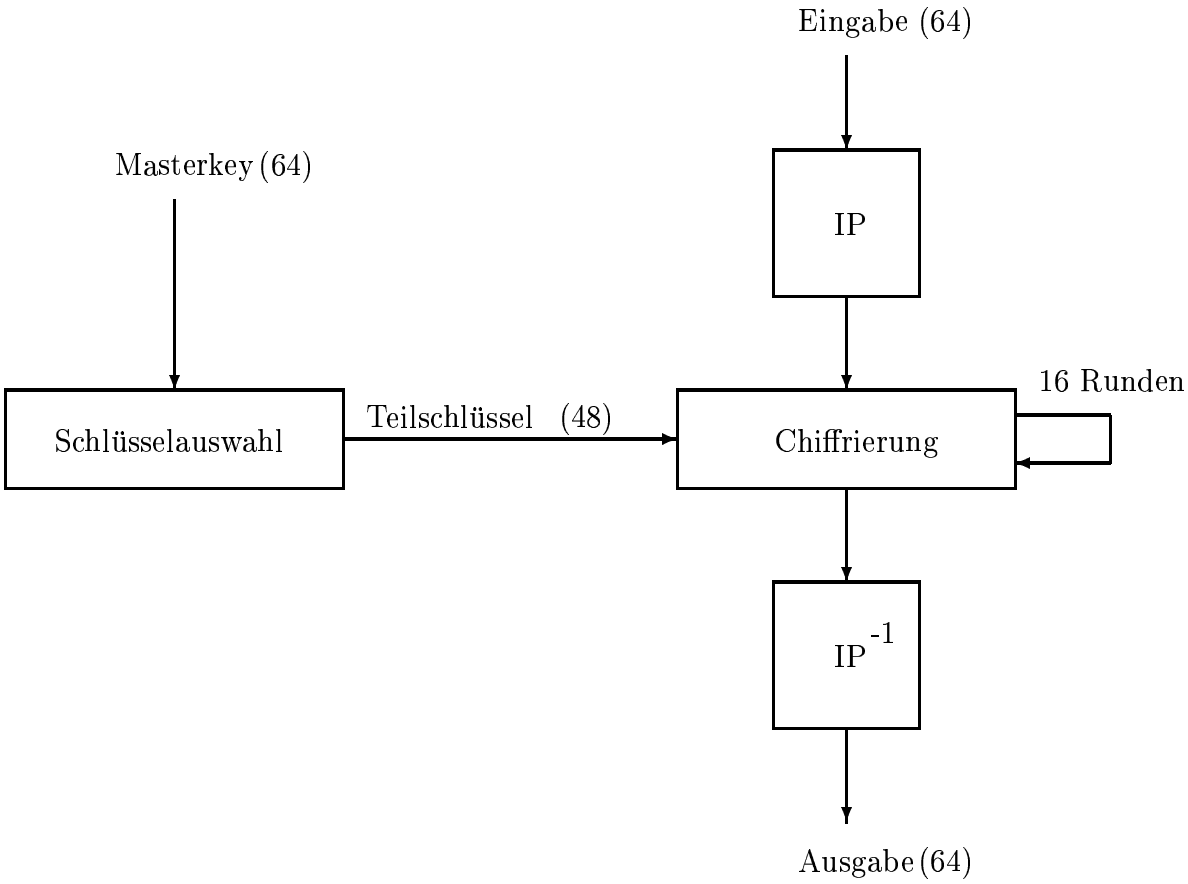

Abbildung 3.3: Prinzipieller Aufbau des DES-Verfahrens

dieser *Boxen* durch eine 4×16-Matrix beschrieben wird, in der jede Zeile eine Substitution

$$S_j^k \ : \ \{0,1\}^4 \ \longrightarrow \ \{0,1\}^4 \tag{3.115}$$

darstellt, wobei k die entsprechende Zeilennummer repräsentiert.

Aus den 8 S-Boxen ergeben sich auf diese Weise 32 Permutationen. Ist nun ein solcher 6-Bit-Vektor $\vec{a}'_j$ gegeben, so bestimmen seine Bits $1 \ldots 6$ die Zeilennummer und entsprechend die Bits $2 \ldots 5$ die Spaltennummer der S-Box S_j. Der so adressierte Wert aus S_j stellt den gesuchten 4-Bit-Ausgabewert dar.

Durch Konkatenation der in den insgesamt 8 Schritten gelieferten 4-Bit-Werte wird nun der endgültige, 32 Bits lange Ausgabevektor $\vec{b}$ erzeugt. Dieser wird zuletzt noch einer abschließenden Ausgabe-Permutation P unterworfen.

Der eben beschriebene Algorithmus des DES-Verfahrens ist noch einmal in Figur 3.3 dargestellt; dort werden die 16 Runden, die für jeden zu verschlüsselnden Block zu durchlaufen sind, verdeutlicht.

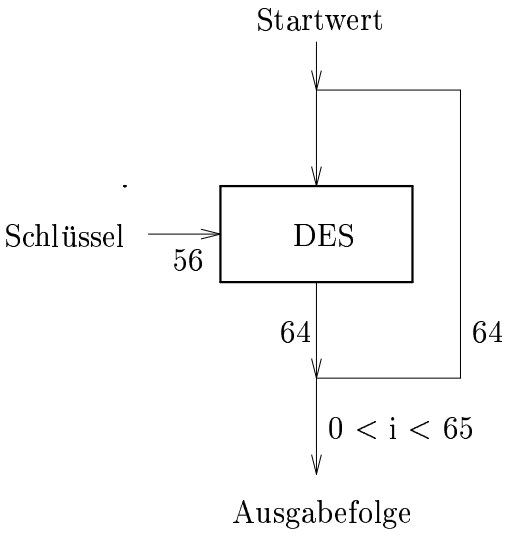

Abbildung 3.4: DES als Pseudozufallsgenerator

3.4.2 Das DES-Verfahren als Pseudozufallsgenerator

Die Verwendung dieses Blockchiffre-Verfahrens zur Erzeugung von Pseudozufallszahlen ist nicht allein auf kryptographische Anwendungsfälle beschränkt, wie [16, S.240ff.] zeigt. Hier wird ein Pseudozufallsgenerator, basierend auf diesem Verschlüsselungssystem für Anwendungen im naturwissenschaftlichen Bereich implementiert[17].

Ausgehend von einem festen Schlüssel wird der als Pseudozufallsgenerator verwandte DES-Algorithmus rekursiv auf seine 64-Bit Ausgabe angewandt, die zu Beginn des ersten Iterationsschrittes auf einen vom Benutzer bestimmten Wert gesetzt wird, wodurch sich eine eigentliche Schlüssellänge von 16 Bytes ergibt.

Zur Verschlüsselung werden nun in jedem Schritt i extrahierte Bits des jeweiligen Iterationswerts mit $1 \leq i \leq 64$ extrahiert und modulo 2 zu den korrespondierenden Klartextbits addiert.

Das IDEA-Verfahren und die MDx-Hashfunktionen

Wie bereits erwähnt, wird im folgenden nicht auf die Besonderheiten des IDEA-Blockchiffre-Verfahrens eingegangen; ebenso wird der interne Aufbau der MD2-, MD4-, sowie MD5-Hashfunktionen nicht weiter erläutert.

Es sei lediglich bemerkt, daß sich der Aufbau eines Pseudozufallszahlengenerators auf Basis des IDEA-Systems analog zu dem in 3.4.2 Gesagten gestaltet. Auch hier verwendet das Blockchiffresystem einen festen Schlüssel, während

[17] Es ist klar, daß ein derartiges Vorgehen nicht die Effizienz anderer, einfacherer Verfahren (*LKG*, *Micali-Schnorr*, etc.) erreichen kann; dennoch ist es von derselben Sicherheit, wie das zugrundeliegende DES-System, zu dem sich ausführliche Sicherheitsbetrachtungen in [7, S.149ff.] finden.

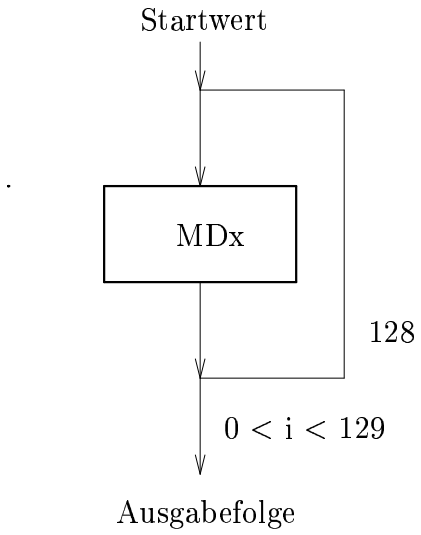

Abbildung 3.5: Die MD-HAshfunktionen als Pseudozufallsgenerator

es iterativ auf seine jeweils letzte Ausgabe angewandt wird. Im Unterschied zu DES benutzt IDEA allerdings einen 16 Bytes langen Schlüssel, arbeitet jedoch ebenfalls auf Blöcken einer Länge von 64 Bits. In jedem Iterationsschritt können somit zwischen einem und vierundsechzig pseudozufällige Bits gewonnen werden; die Effizienz dieses Chiffrierverfahrens überschreitet die des DES-Verfahrens.

Unterschiedlich in ihrer Verwendung gestalten sich hierzu die MDx-Hashfunktionen[18], die ohne Steuerung durch einen Schlüssel einen Eingabewert auf einen Bitvektor der festen Länge 16 Bytes abbilden. Dies hat zur Folge, daß der Startwert eines auf diesen Routinen basierenden Pseudozufallszahlengenerators keinen Schlüsselanteil beinhaltet und somit eine Länge von 16 Bytes aufweist.

Dieser Wert wird zur Initialisierung der Iteration eingesetzt, in deren Verlauf innerhalb eines jeden Schrittes zwischen einem und 128 Bits des Iterationswertes extrahiert werden, welche die Ausgabe des Generators repräsentieren, wie Abbildung 3.5 verdeutlicht.

Nachdem mit diesem Abschnitt die Beschreibung der zu implementierenden Verfahren abgeschlossen wurde, widmet sich das folgende Kapitel der Realisierung der einzelnen Generatoren, wobei Betrachtungen zu statistischen Fragen im Anhang wiedergegeben werden.

[18]Diese *Message-Digest* Algorithmen wurden von *RSA Data Security, Inc.* entwickelt und sind unter der Voraussetzung, daß diese Quelle genannt wird, frei verwendbar.

Kapitel 4

Implementation der Verfahren

4.1 Vorbemerkungen

Zur Implementation der Verfahren ist grundsätzlich zu sagen, daß in den Fällen, die Langzahlarithmetik benötigen, das *GNU-MP*-Paket Verwendung fand. Hierbei handelt es sich um eine allgemeine Bibliothek von Funktionen zur Handhabung von Integer-Zahlen, deren Länge prinzipiell nur durch den zur Verfügung stehenden Speicherplatz eingeschränkt wird. Aufgrund der einheitlichen Schnittstelle der GNU-MP-Funktionen dürfte ein Austausch dieses Paketes (beispielsweise zur Verbesserung des Laufzeitverhaltens) gegen ein anderes kein unüberwindliches Problem darstellen.

Weiterhin verwendet die implementierte Bibliothek eine Reihe bekannter kryptographischer Routinen; dies sind:

- Die MD-Hashfunktionen der RSA Data Security, Inc., Created 1990. Dies sind im einzelnen: *MD2*, *MD4* und *MD5*.
- Die kryptographischen Algorithmen für das verwendete IDEA-System von

  Richard De Moliner (demoliner@isi.ethz.ch)
  Signal and Information Processing Laboratory
  Swiss Federal Institute of Technology CH-8092 Zürich, Switzerland

  in der Version vom 23. April 1992.
- Die Implementation des DES-, sowie D3DES-Systems von

  Richard Outerbridge.

Bereits an dieser Stelle sei zu den implementierten Funktionen folgendes bemerkt: Da einige Routinen die *rand*-Funktion der Laufzeitbibliothek des jeweiligen Systems verwenden, stellt sich die Frage nach einer geeigneten Initialisierung dieser Funktion vor ihrer eigentlichen Verwendung. Zur Kontrolle dieses Mechanismus dient die Konstante *DISABLE_INIT*. Wird sie vor der Übersetzung von *bitstream_lib.c*[1] (beispielsweise in *bitstream_lib.h* definiert, unterbleibt

[1] Eine Kurzbeschreibung der einzelnen Files, aus denen sich die gesamte Bitstrom-Bibliothek zusammensetzt, findet sich in Tabelle C.1 in Anhang C.

eine Initialisierung innerhalb der Bitstrombibliothek, so daß in diesem Fall das rufende Programm hierfür Sorge tragen muß. Ist diese Konstante nicht definiert, erfolgt eine Initialisierung mit Hilfe der Funktion *time()*, deren Rückgabewert allerdings lediglich in Sekundenintervallen inkrementiert wird, was für eine Reihe von Anwendungen sicherlich ungeeignet ist.

Die Entwicklung der Programme fand auf einer *VAX* unter *VMS 6.1* statt, ein hierfür portiertes GNU-MP-Paket steht zur Verfügung und kann bei Bedarf an Interessenten abgegeben werden. Besonderer Dank gebührt an dieser Stelle Herrn *Wolfgang J. Möller* von der *Gesellschaft für wissenschaftliche Datenverarbeitung, Göttingen*, ohne dessen Unterstützung dieses Langzahlarithmetik-Paket nur unter großem Aufwand unter VMS lauffähig geworden wäre.

Da alle implementierten Verfahren auf der Anwendung einer rekursiven Rechenvorschrift beruhen, sei an dieser Stelle darauf hingewiesen, daß die Implementation in allen Fällen kein Interleave der Rekursionswerte ermöglicht, was jedoch bei Bedarf leicht modifizierbar sein sollte. Das bedeutet, daß die eigentliche Funktion stets in Form von

$$x_{n+1} = f(x_{n-i}) \qquad \text{mit } i = 0 \tag{4.1}$$

angewandt wird.

Weiterhin ist zu bemerken, daß alle Programme, die auf der GNU-MP-Bibliothek basieren, die Include-Files *gmp.h*, *gmp-impl.h*, sowie *urandom.h* in ihrem Verzeichnis zur Compilation benötigen.

Alle im Rahmen dieser Diplomarbeit implementierten Pseudozufallsgeneratoren haben eine Gemeinsamkeit: Sie verwenden innerhalb der sie enthaltenden Bibliothek globale Variablen (zur Steuerung interner Abläufe und zur Sicherung einzelner Zwischenwerte, die über Funktionsaufrufe hinaus erhalten bleiben müssen), so daß sich die Einbindung der Verfahren in eigene Programme stets in einen *Initialisierungsschritt*, die eigentliche *Iteration* und einen abschließenden *Cleanup*-Schritt gliedert.

4.2 Der BBS-Generator

Das Verfahren nach Blum-Blum-Shub gliedert sich in die folgenden drei grundlegenden Einheiten:

- Schlüsselgenerierung,
- Verschlüsselung und
- Entschlüsselung von Daten,

die nun einzeln in ihrem Aufbau erläutert werden sollen:

4.2.1 Schlüsselerzeugung

Die hierzu gehörenden Bibliotheksroutinen dienen der Erzeugung eines öffentlichen sowie eines geheimen Schlüssels für den BBS-Generator, wobei der nichtöffentliche Schlüssel aus zwei Primzahlen P und Q besteht, für die gilt:

- P und Q sowie $P - Q$ sind von derselben Größenordnung,
- P und Q können bei Bedarf als spezielle Primzahlen nach Definition 8 erzeugt werden, so daß sie sich wie folgt zusammensetzen:

$$\begin{aligned} P &= 2P' + 1, && (4.2)\\ P' &= 2P'' + 1, && (4.3)\\ Q &= 2Q' + 1 \text{ und} && (4.4)\\ Q' &= 2Q'' + 1, && (4.5) \end{aligned}$$

  wobei P', Q', sowie P'' und Q'' wiederum prim sind und es gilt stets

- $P \equiv Q \equiv 3 \pmod 4$.

Der öffentliche Schlüssel N ist das Produkt dieser beiden Primzahlen P und Q, und somit eine Blum-Zahl. Durch obige Wahl von P und Q wird die Anwendung von *Pollards-ρ*-Methode unterbunden, die die effiziente Faktorisierung von Zahlen der Gestalt $N = PQ$ unter der Voraussetzung, daß entweder $P - 1$ oder $Q - 1$ das Produkt kleiner Primzahlen ist, ermöglicht.

4.2.2 Verschlüsselung

Zur Verschlüsselung eines Eingabedatenstromes wird lediglich der öffentliche Schlüssel N benötigt, der als Modul des Verfahrens dient. Das prinzipielle Vorgehen gestaltet sich wie folgt:

Zunächst wird ein *zufälliger* (hierfür kann ein beliebiger einfacher Zufallszahlengenerator verwendet werden, wie beispielsweise die *rand()*-Funktion der C-Laufzeitbibliotheken) Startwert x_0 ausgewählt, für den gilt:

$$x_0 \in [\sqrt{N} + 1 : N - 2]. \tag{4.6}$$

Dieser Startwert wird vor seiner eigentlichen Verwendung einmal modulo N quadriert, d.h. $x_1 := x_0^2 \bmod N$. Nun wird ausgehend hiervon die Folge

$$x_{i+1} = x_i^2 \bmod N \tag{4.7}$$

berechnet. Dabei wird von jedem Folgenglied eine bestimmte Anzahl von Bits extrahiert (beginnend mit dem niederwertigsten Bit), die bei Aufruf der entsprechenden Routinen spezifiziert werden kann. Jeweils 8 derart gewonnene Zufallsbits können nun der Verschlüsselung eines Zeichens des Klartextes dienen, indem sie zu jeweils einem Klartextzeichen binär modulo 2 addiert werden.

Um die nachfolgende Entschlüsselung des so gewonnen Chiffretextes zu ermöglichen, muß nach vollständiger Abarbeitung des gesamten Eingabedatenstroms das nächste Folgenelement der x_i durch Quadrierung wie oben bestimmt werden. Im vorliegenden Anwendungsfall wird es zusätzlich zu den Chiffretextzeichen dem Empfänger übermittelt, der mit seiner Hilfe und der Kenntnis über die Zerlegung von N in der Lage ist, den ursprünglich zur Verschlüsselung verwendeten Iterationsstartwert x_0 zu bestimmen, wie der folgende Abschnitt zeigt:

4.2.3 Entschlüsselung

Prinzipiell verläuft die Entschlüsselung analog zur Verschlüsselung, mit dem Unterschied, daß der verwandte Startwert nicht zufällig bestimmt, sondern ausgehend von dem wie oben bestimmten letzten Quadratrest (im folgenden x_{k+1} genannt), unter Zuhilfenahme des geheimen Schlüssels errechnet wird.

Zusätzlich müssen die Anzahl der pro Schritt extrahierten Bits, sowie die Anzahl der Klartext/Chiffretext-Zeichen bekannt sein, um ausgehend hiervon $k+1$ bestimmen zu können, mit dessen Hilfe das benötigte x_0 berechnet werden kann:

Nach Satz 8 gilt

$$\begin{aligned} x_{-i} \bmod P &= \pm x_0^{(\frac{P+1}{4})^i} \bmod P && (4.8)\\ &= \pm x_0^{(\frac{P+1}{4})^i \bmod (P-1)} \bmod P \qquad \text{und} && (4.9)\\ x_{-i} \bmod Q &= \pm x_0^{(\frac{Q+1}{4})^i} \bmod Q && (4.10)\\ &= \pm x_0^{(\frac{Q+1}{4})^i \bmod (Q-1)} \bmod Q. && (4.11) \end{aligned}$$

Wurden auf diese Art und Weise $x_0 \bmod P$ und $x_0 \bmod Q$ berechnet, kann mit Hilfe des chinesischen Restsatzes leicht $x_0 \bmod N$ errechnet werden.

Da $N = PQ$, $Q \neq Q$ eine Blum-Zahl ist, gilt $\mathrm{ggT}(P, Q) = 1$, d.h. es existiert genau ein $x \in [0 : N-1]$ mit

$$x \equiv a_1 \pmod P \equiv a_2 \pmod Q \qquad \text{und } a_i \in \mathbb{Z}. \tag{4.12}$$

Bestimme nun (beispielsweise mit Hilfe des *Berlekamp*-Algorithmus, siehe [7, S.292ff.]) x_1 und x_2, so daß gilt:

$$\begin{aligned} x_1 p \bmod q &= 1 \text{ und} && (4.13)\\ x_1 q \bmod p &= 1. && (4.14) \end{aligned}$$

Setzt man

$$\begin{aligned} r_1 &= x_1 p \text{ und} && (4.15)\\ r_2 &= x_2 q, && (4.16) \end{aligned}$$

so kann der gesuchte Startwert $x_{-i} \bmod N$ leicht bestimmt werden:

$$x_{-i} \bmod N = [(x_{-i} \bmod p) r_2 + (x_{-i} \bmod q) r_1)] \bmod N. \tag{4.17}$$

Ausgehend hiervon kann nun erneut die Folge der Werte $x_0, \ldots, x_k$ erzeugt werden, von denen jeweils dieselbe Anzahl Bits wie beim zugehörigen vorangegangenen Verschlüsselungsschritt extrahiert wird. Diese Ausgabebits werden nun wieder binär modulo 2 zu den korrespondierenden Chiffretextbits addiert, um so den Klartext zurückzugewinnen.

Beispiel 4:

Im folgenden seien $P = 2039$ und $Q = 4079$, d.h. $N = 8317081$. Der Ausgangswert x_{k+1} sei 12345, gesucht ist x_0 mit $k = 2$. Berechne nun x_1 und x_2 mit der Eigenschaft von (4.14). Man erhält $x_1 = 4077$ und $x_2 = 1$. Und hiermit nach (4.16) $r_1 = 8313003$ und entsprechend $r_2 = 4079$.

Hieraus ergibt sich durch Anwendung von (4.17) sofort das gesuchte x_0 zu $x_0 = 8299134$. Wie man leicht nachprüft, gilt wirklich $x_0^8 \bmod 8317081 = 12345$.

4.2.4 Die Bibliotheksfunktionen zu Blum-Blum-Shub:

Nach den vorangegangenen Erläuterungen werden nun die einzelnen Routinen, die das Verfahren nach Blum-Blum-Shub implementieren, eingehend besprochen:

Schlüsselerzeugung:

Zur Erzeugung eines neuen Schlüssels, d.h. zweier Primzahlen P und Q mit Eigenschaften wie in Abschnitt 4.2.1, dient die Funktion *bbs_new_key*:

```
int bbs_new_key (MP_INT *p, MP_INT *q, long key_len, int mode);
```

Die beiden ersten Übergabeparameter vom Typ *MP_INT* dienen der Aufnahme der zu erzeugenden Primzahlen P bzw. Q. Das Argument *key_len* enthält die Länge des sich aus der Multiplikation von P und Q ergebenden öffentlichen Schlüssels N in Bits, d.h. es gilt

$$|P| = \left\lfloor \frac{key\_len}{2} \right\rfloor \tag{4.18}$$

$$|Q| = \left\lceil \frac{key\_len}{2} \right\rceil. \tag{4.19}$$

P und Q sind hiermit von derselben Größenordnung.

Durch den in *mode* übergebenen Wert kann die Erzeugung einfacher oder spezieller Primzahlen gewählt werden. Die hierfür zulässigen Konstanten sind in *bitstream_lib.h* definitiert: *BBS$NORMAL* bzw. *BBS$SPECIAL*. Der Parameter *mode* wird auch in den beiden folgenden Funktionen verwandt, in denen ihm dieselbe Bedeutung wie im eben beschriebenen Fall zukommt.

Intern ruft *bbs_new_key* für jeden der beiden zu erzeugenden Werte die Routine *blum_random_prime* auf, die eine spezielle Primzahl von n Bits Länge zurückliefert:

```
int blum_random_prime (MP_INT *p, long n, int mode);
```

Hierzu wird zunächst mit Hilfe des Pseudozufallsgenerators der GNU-MP Bibliothek (ein linearer Kongruenzgenerator) eine Pseudozufallszahl x erzeugt, die eine Länge von n Bits aufweist. Mit ihr als Argument wird die im folgenden beschriebene Prozedur *blum_prime* aufgerufen:

```
int blum_prime (MP_INT *result, MP_INT *x, int mode);
```

Sie liefert in *result* die kleinste Primzahl $\geq x$ zurück, die die in 4.2.1 beschriebene Gestalt besitzt. Hierzu wird im Falle einer speziellen Primzahl (d.h. $mode = BBS\$SPECIAL$) folgender Algorithmus eingesetzt:

```
Setze result = x.
Falls result gerade, inkrementiere result um 1.
Berechne p1 = (result - 1) / 2.
Berechne p2 = (p1     - 1) / 2.
Falls p1 gerade,
      inkrementiere result um 2,
      berechne p1 und p2 wie eben neu.
Falls p2 gerade,
      inkrementiere result um 4,
      berechne wieder p1 und p2.

(Die beiden vorangegangenen Vergleiche haben zur
Folge, dass mod (result, 8) = 7 gilt und somit
p, p1 und p2 ungerade sind.)

Solange weder result noch p1 noch p2 prim sind,
      inkrementiere result um 8,
      berechne p1 und p2 neu.
```

Im Falle einer nicht-speziellen zu erzeugenden Primzahl vereinfacht sich das Vorgehen wie folgt:

```
Setze result = x.
Falls result gerade, inkrementiere result um 1.
Berechne p1 = (result - 1) / 2.
Falls p1 gerade,
      inkrementiere result um 2,
      berechne p1 neu.

Solange weder result noch p1 prim sind,
      inkrementiere result um 4,
      berechne p1 neu.
```

Falls der hiervon an *blum_random_prime* zurückgelieferte Wert $result \geq 2^n$ ist, wird er von *blum_random_prime* verworfen und durch das Ergebnis eines erneuten Aufrufs der eben beschriebenen Routine *blum_prime* mit $x = 2^{n-1}$ ersetzt. Das so erhaltene Resultat wird nun von *blum_random_prime* als P zurückgeliefert.

Die praktische Erzeugung eines neuen Schlüssels mit Hilfe der eben vorgestellten Routinen, dessen Primfaktoren einfache Primzahlen darstellen, mag folgendes Programmfragment verdeutlichen (im Falle zu erzeugender spezieller Primzahlen wäre *BBS$NORMAL* durch *BBS$SPECIAL* zu ersetzen):

```
MP_INT p, q, m; /* p und q als privater, m als */
                /* oeffentlicher Schluessel.   */
long key_len;
int status;

mpz_init (&p);
mpz_init (&q);

     /* Praeliminarien, setzen von key_len,... */

status = bbs_new_key (&p, &q, key_len, BBS$NORMAL);
switch (status)
{       case RC$NORMAL : /* Neuer Schluessel wurde erzeugt.      */
                         break;
        case RC$VALERR : /* Zu kleine Schluessellaenge           */
                         /* uebergeben, kein Schluessel erzeugt  */
                         /* Verzweigen zu Fehlerbehandlung.      */
        case RC$WRONG_MODE :
                         /* mode enthielt einen unzulaessigen    */
                         /* Wert.                                */
        case RC$KEY_NOT_GENERATED :
                         /* Fuer die angegebene Schluessellaenge */
                         /* konnte kein Schluessel mit p != q    */
                         /* erzeugt werden. Ein erneuter Versuch */
                         /* sollte eine groessere Schluessel-    */
                         /* laenge vorsehen.                     */
                         /* Eine andere Bedingung, die diesen    */
                         /* Return-Code zur Folge hat, ist die   */
                         /* Nichterfuellung der Groessenordnungs-*/
                         /* bedingung |p|=|q|=|p-q|.             */
}

mpz_mul (&m, &p, &q);
```

Nach diesen Anweisungen steht nun in p und q der private Schlüssel, sowie in m der öffentliche Schlüssel als Produkt aus p und q zur weiteren Verfügung.

Verschlüsselung:

Zur Verschlüsselung von Nutzdaten muß zunächst ein *zufälliger* Startwert x_0 erzeugt werden, was unter Zuhilfenahme der Routine *bbs_gen_seed* vollzogen werden kann, deren Aufruf folgendes Programmfragment darstellt[2]:

```
MP_INT m, seed; /* m enthaelt den Modul des Verfahrens,  */
```

[2]Dieses Programmfragment wird im weiteren Verlauf fortgesetzt und erweitert.

```
                /* seed wird den zu erzeugenen Startwert */
                /* aufnehmen.                            */

bbs_gen_seed (&m, &seed);
```

Der solchermaßen erzeugte Wert *seed* erfüllt folgende Bedingung: $\sqrt{m} < seed < m - 1$. Nachdem nun also ein entsprechender[3] Startwert zur Verfügung steht, kann der Initialisierungsschritt der Blum-Blum-Shub–Routinen aufgerufen werden, wie dies in Fortsetzung des obigen Programmausschnittes erfolgt:

```
int status, bits_per_step;

     /* Praeliminarien, die das Setzen von */
     /* bits_per_step umfassen...          */

status = bbs_init (&m, bits_per_step, &seed);
switch (status)
{       case RC$NORMAL : /* Initialisierung wurde erfolgreich */
                         /* abgeschlossen.                    */
                         break;
        case RC$VALERR : /* Der uebergebene Wert fuer die An- */
                         /* zahl der pro Iterationsschritt zu */
                         /* extrahierenden Bits war unzulaes- */
                         /* sig. Verzweigen zu Fehlerbehand-  */
                         /* lung.                             */
}
```

Dieser Initialisierungsaufruf setzt eine Reihe bibliotheksglobaler Variablen, in denen die Anzahl der in jedem Iterationsschritt zu extrahierenden Bits des Verfahrens, der initiale Startwert, sowie der zu verwendende Modul abgelegt werden. Weiterhin existieren globale Variablen zur Sicherung von Zwischenergebnissen, die außerhalb der Bibliotheksroutinen nicht zugänglich sind, jedoch mit Hilfe von *bbs_init* ebenfalls initialisiert werden.

Nach diesen vorbereitenden Schritten kann nun mit der Erzeugung von Pseudozufallszahlen durch fortgesetzte Quadrierung des Startwertes *seed* begonnen werden. Hierfür wird die Routine *bbs_prg_iterate* zur Verfügung gestellt, die im folgenden für jedes zu verschlüsselnde Klartextzeichen einmal aufgerufen werden muß. Jeder solche Funktionsaufruf liefert ein pseudozufälliges Byte zurück, mit dessen Hilfe durch eine Exklusiv-Oder-Verknüpfung mit jeweils einem Klartextbyte das entsprechende Chiffretextzeichen erzeugt werden kann[4]:

[3] Der Fall $seed = \sqrt{m}$ hätte nach Quadrierung modulo m das Resultat 0, $seed = m - 1$ entsprechend 1 zur Folge.

[4] Der Aufruf dieser Funktion setzt eine korrekte Initialisierung des Verfahrens durch einen vorangegangenen *bbs_init*-Aufruf zwingend voraus! Ist diese Bedingung nicht erfüllt, wird das Programm mit einer entsprechenden Fehlermeldung abgebrochen!

```
unsigned char *klartext, *chiffre;

/* Beliebige Schleife, um alle Klartextzeichen zu */
/* verschluesseln...                              */

{
        *chiffre++ = *klartext++ ^ bbs_prg_iterate ();
}
```

Sind nun auf diese Art und Weise alle Zeichen des Klartextes abgearbeitet, muß noch der nächste Seed bestimmt werden, um es dem Empfänger zu ermöglichen, mit dessen Hilfe und seiner Kenntnis der Primfaktorzerlegung des Moduls den ursprünglichen Startwert nach 4.2.3 zu bestimmen. Dies kann durch Aufruf der Routine *bbs_next_seed* erreicht werden:

```
/* Nach Schleife zur Verschluesselung des Klartextes... */

bbs_next_seed (&seed);
```

Dieser Wert kann nun zusammen mit dem Chiffretext an den Empfänger übermittelt werden. Nachdem nun alle Schritte zur Verschlüsselung eines gegebenen Klartextes abgearbeitet sind, bleibt zum Schluß noch der Aufruf von *bbs_cleanup*[5], mit dessen Hilfe die im Verlauf des Verfahrens allokierten Speicherbereiche für das Langzahlarithmetikpaket wieder freigegeben werden:

```
/* Nun Speicher wieder freigeben... */

status = bbs_cleanup ();
```

4.2.5 Entschlüsselung:

Das prinzipielle Vorgehen bei der Entschlüsselung eines gegeben Chiffretextes gestaltet sich fast analog zur Vorgehensweise bei der Verschlüsselung. Einziger Unterschied ist die Erzeugung des initialen Startwertes, der aus dem mit dem Chiffretext mitgelieferten Seed-Wert ermittelt wird, wie dies in Abschnitt 4.2.3 beschrieben wurde. In der praktischen Anwendung implementiert die Routine *bbs_backstep* diesen Algorithmus:

```
long steps;
MP_INT p, q, seed;
```

[5]Diese Funktion ist in der Lage, einen Fehlercode zurückzuliefern, der im vorliegenden Beispiel jedoch nicht näher verwertet wird. Für eine eingehendere Beschreibung siehe Anhang A.

```
/* Bestimme die Anzahl von Schritten, die zurueck-    */
/* gerechnet werden muss, aus der Anzahl der Bits,    */
/* die pro Iterationsschritt bei der Verschluesselung */
/* extrahiert wurden und der Anzahl der Zeichen       */
/* der Nachricht. Lege diesen Wert in steps ab.       */

... ... ... ... ... ... ... ... ... ... ... ... ...

/* seed enthaelt den zusammen mit dem Chiffretext  */
/* uebermittelten naechsten Seed aus der Ver-      */
/* schluesselung.                                  */

status = bbs_backstep (steps, &p, &q, &seed);
if (status != RC$NORMAL)
        /* Wird RC$VALERR zurueckgeliefert, wurde  */
        /* in steps ein Wert kleiner 1 uebergeben. */

/* Nach erfolgtem Prozeduraufruf enthaelt seed den */
/* gesuchten initialen Startwert.                  */
```

Als Argumente erwartet die Routine die Anzahl der Schritte, die zurückgerechnet werden soll, um x_0 zu erhalten, die beiden Teile des privaten Schlüssels p und q, sowie den Wert von x_{k+1}, der nach vollzogener Verschlüsselung gewonnen wurde.

Der gesuchte Startwert x_0 wird nach Abarbeitung der Routine in ihrem letzten Argument zurückgegeben, das schon der Übergabe von x_{k+1} diente.

Alle sich anschließenden Schritte entsprechen vollkommen dem bereits bei der Verschlüsselung Gesagten; nach Aufruf von *bbs_init* können durch wiederholte Aufrufe von *bbs_prg_iterate* wieder die entsprechenden Pseudozufallsbytes gewonnen werden, die schon bei der Verschlüsselung zum Einsatz kamen. Nach Exklusiv-Oder-Verknüpfung dieser Werte mit den entsprechenden Zeichen des Chiffretextes steht der Klartext wieder zur Verfügung.

Nachdem nun alle wesentlichen Routinen der Implementation des Blum-Blum-Shub–Generators behandelt wurden, kann mit der Beschreibung der nächsten Funktionsgruppe fortgefahren werden:

4.3 Der Generator nach Micali-Schnorr

Der folgende Abschnitt betrachtet vorab die notwendigen Voraussetzungen, die an die Parameter des Verfahrens gestellt werden, um seine Sicherheit zu gewährleisten.

4.3.1 Parameterwahl

Für praktische Anwendungen des Polynomial-Generators nach Abschnitt 3.2.1 ist es zunächst wichtig, N genügend groß zu wählen, um seiner Faktorisierung vorzubeugen. In [11] wird folgendes Verfahren vorgeschlagen, das jedoch in der

vorliegenden Implementation nicht weiter verfolgt wurde:

N sei das Produkt zweier zufälliger Primzahlen P und Q, die jeweils eine Länge von mindestens 256 Bits aufweisen. Desweiteren muß gewährleistet sein, daß $P-1$, $P+1$, $Q-1$ und $Q+1$ mindestens einen Primfaktor aufweisen, der größer als 2^{80} ist.

Nicht alle diese Bedingungen werden von der vorliegenden Implementation berücksichtigt. Die Erzeugung eines geeigneten Moduls wird der Einfachheit halber mit den Routinen des Blum-Blum-Shub–Generators vollzogen, wobei die erzeugten Primzahlen P und Q (wie bereits erwähnt) folgende Eigenschaften besitzen, was letztlich ausreichend sein sollte:

- P und Q, sowie $P-Q$ sind von derselben Größenordnung,
- P und Q setzen sich wie folgt zusammen:

$$P = 2P' + 1, \tag{4.20}$$
$$Q = 2Q' + 1, \tag{4.21}$$

  wobei P' und Q' wiederum prim sind.

- $P \equiv Q \equiv 3 \pmod 4$.

Soll der zu generierende Modul N, der die Gestalt $N = PQ$ aufweist, eine Länge von n Bits besitzen, so gilt für die Längen seiner beiden Primfaktoren:

$$|P| = \left\lfloor \frac{n}{2} \right\rfloor \tag{4.22}$$
$$|Q| = \left\lceil \frac{n}{2} \right\rceil \tag{4.23}$$

Zur Erzeugung dieser beiden Primfaktoren eines neuen Moduls steht die Funktion *mschnorr_new_mod* zur Verfügung, deren Aufruf folgendes Programmfragment verdeutlichen mag:

```
int status;
long mod_len;
MP_INT p, q, modul;

mpz_init (&p);
mpz_init (&q);
mpz_init (&modul);

/* Vor Aufruf der eigentlichen Funktion zur Generierung */
/* eines neuen Moduls muss mod_len mit der gewuenschten */
/* Laenge des Moduls in Bit belegt werden...           */
```

```
status = mschnorr_new_mod (&p, &q, mod_len);
switch (status)
{       case RC$NORMAL : break;
case RC$VALERR : /* mod_len enthielt einen */
                         /* unzulaessigen Wert!    */
        case RC$KEY_NOT_GENERATED :
                         /* Es konnten keine zwei    */
                         /* Primzahlen p und q mit   */
                         /* p != q und |p|=|q|=|p-q| */
                         /* zur Basis 10 gefunden    */
                         /* werden.                  */
}

/* Nun neuen Modul erzeugen: */

mpz_mul (&modul, &p, &q);
```

Nach der Erzeugung eines solchen Moduls muß ein passender Exponent d mit $d > 3,\ d \bmod 2 = 1$ für das Verfahren ausgewählt werden, der zusätzlich

$$\mathrm{ggT} = (d, \varphi(N)) = 1 \tag{4.24}$$

erfüllt. Die Auswahl eines solchen d wird mit Hilfe der Funktion *mschnorr_gen_d* vollzogen (der folgende Programmausschnitt setzt den vorangegangenen fort):

```
long d;

d = mschnorr (&p, &q);
if (d < 3)
        /* Es konnte kein den gestellten  */
        /* Bedingungen Genuege leistender */
        /* Exponent d aufgefunden werden. */
```

Diese Funktion testet in aufsteigender Reihenfolge alle ungeraden d mit $2 < d < \log_2 N$ auf Erfüllung der Bedingung 4.24. Kann innerhalb dieser Grenzen kein solches d gefunden werden, liefert die Funktion den Wert 0 zurück[6].

Nachdem jetzt sowohl ein Modul N mit einer Länge von n Bits, als auch ein zugehöriger Exponent d erzeugt wurden, muß lediglich noch ein passender Startwert s mit

$$2^{\frac{2n}{d}-1} \leq s < 2^{\frac{2n}{d}} \tag{4.25}$$

generiert werden, was durch Aufruf der Funktion *mschnorr_gen_seed* geschieht:

[6]Da durch P und Q die Primfaktorzerlegung des Moduls zur Verfügung steht, ist man für die Erzeugung eines solchen d nicht ausschließlich auf diese Funktion angewiesen. Über $\varphi(N) = (P-1)(Q-1)$ kann für ein benutzergewähltes d Bedingung 4.24 leicht nachgeprüft werden, so daß auf diese Art und Weise leicht ein eigener Exponent d Verwendung finden kann.

```
MP_INT seed;

mpz_init (&seed);

status = mschnorr_gen_seed (&seed, &modul, d);
switch (status)
{       case RC$NORMAL : break;

}
```

Um nun das Verfahren von Micali-Schnorr mit den so generierten Parametern anwenden zu können, müssen noch die von ihm benötigten, bibliotheksinternen globalen Variablen initialisiert werden:

```
status = mschnorr_init (&modul, d, &seed);
switch (status)
{       case RC$NORMAL : break;
        case RC$VALERR : /* Der in d uebergebene Exponent */
                         /* ist kleiner 3.               */
{
```

Selbstverständlich muß nicht für jede Anwendung dieses Verfahrens ein vollständiger neuer Parametersatz – bestehend aus N und d – erzeugt werden. Er kann vielmehr des öfteren Anwendung finden, und entsprechend durch das Rahmenprogramm in Files verwaltet werden (lediglich die Erzeugung eines neuen Startwertes bleibt zu vollziehen).

Nachdem die Initialisierung des Verfahrens durch obigen Aufruf von *mschnorr_init* durchgeführt wurde, kann mit der Erzeugung pseudozufälliger Bytes zur Ver- bzw. Entschlüsselung von Daten fortgefahren werden:

```
unsigned char *chiffre, *klartext;

/* Eine Schleife, um alle zur Verfuegung stehenden */
/* Zeichen abzuarbeiten...                         */
{
        *chiffre++ = *klartext++ ^ mschnorr_prg_iterate ();
}
```

Jeder Aufruf der Funktion *mschnorr_prg_iterate* liefert ein neues, pseudozufälliges Byte nach dem Verfahren von Micali und Schnorr zurück. Zu beachten ist an dieser Stelle noch, daß ein erfolgreicher Aufruf dieser Routine eine korrekte Initialisierung des Verfahrens durch *mschnorr_init* zwingend voraussetzt! Ist dies nicht gegeben, wird der Programmlauf unter Ausgabe einer entsprechenden Fehlermeldung abgebrochen!

Wurden alle Eingabedaten vollständig verschlüsselt, kann der während des Programmlaufs belegte Speicherplatz der globalen Variablen durch einmaligen Aufruf der Funktion *mschnorr_cleanup* wieder freigegeben werden:

```
status = mschnorr_cleanup ();
switch (status)
{       case RC$NORMAL          : break;
        case RC$NOT_INITIALIZED : /* Das Verfahren wurde */
                                  /* nicht korrekt ini-  */
                                  /* tialisiert!         */
}
```

4.4 Das Impagliazzo-Naor-Verfahren

Da es sich bei diesem Verfahren um eine rein symmetrische Verschlüsselungstechnik handelt, entfällt zunächst die (beispielsweise bei *bbs* notwendige) Unterteilung in Ver- bzw. Entschlüsselung von Eingabedaten, so daß lediglich die folgenden grundlegenden Gruppen von Routinen näher zu beleuchten bleiben:

- Schlüsselerzeugung und
- Ver-/Entschlüsselung.

4.4.1 Funktionsweise

Da das Verfahren nach Impagliazzo-Naor auf Anwendung des Knapsack-Problems basiert (siehe Kapitel 3.3), geht die Schlüsselerzeugung in gewisser Hinsicht über das bisherige Maß hinaus, weil nicht nur ein geeigneter Startwert für das Verfahren an sich ausgewählt, sondern zudem die benötigten a_i für den Parametersatz erzeugt werden müssen.

Zwei Werte sind für einen solchen Parametersatz kennzeichnend: Einerseits die Anzahl n der Vektorelemente, andererseits die Länge der einzelnen Elemente in Bits, $l(n)$. Ausgehend von einem Startwert $\vec{x}$, der als Bitvektor der Länge n aufgefaßt wird, beruht das Verfahren auf Summationen folgender Art:

$$T = \sum_{i=1}^{n} a_i x_i \bmod 2^{l(n)}. \tag{4.26}$$

Das Resultat T dieser Addition besitzt eine Länge von $l(n)$ Bits[7], von denen die unteren (siehe Abbildung 3.2) als neuer Startwert $\vec{x}$ für den nächsten Iterationsschritt verwandt werden, so daß pro Iteration $l(n) - n$ Bits ausgegeben werden können. Durch geeignete Wahl von n bzw. $l(n)$ bei Erzeugung des Parametersatzes kann die Effizienz des Verfahrens also in hohem Maße beeinflußt werden (wobei prinzipiell $l(n) > n$ zu wählen ist).

Da die Wahl der Parameter als *der* sicherheitsbestimmende Faktor bei diesem Verfahren angesehen werden kann, ist es eventuell sinnvoll, nicht lediglich eine Folge konsekutiv aufeinanderfolgender Zufallszahlen über die C-eigene Runtime-Bibliothek zu generieren, um nicht auf eventueller Kenntnis des Pseudozufallszahlengenerators der Bibliothek (ein sicherlich in der Mehrzahl der

[7]Die auflaufenden Überträge, die eine Länge $\lfloor \log_2 n \rfloor$ Bits besitzen, werden durch den Modul $2^{l(n)}$ unterdrückt.

Fälle auf einem linearen Kongruenzgenerator beruhendes Verfahren) basierenden Attacken unnötig Vorschub zu leisten. Da eine stete Neuinitialisierung mit Zeitwerten o.ä. aus Effizienzgründen nicht unbedingt wünschenswert ist, wurde in der vorliegenden Implementation auf eine Liste zur Laufzeit generierter Pseudozufallszahlen der rand()-Routine zurückgegriffen, die in ihrer Länge der Anzahl gewünschter Parameter entspricht. Mit ihrer Hilfe wird nun nach der Erzeugung eines jeden Parameters der Zufallsgenerator des Systems erneut initialisiert. Diese Vorgehensweise ist sicherlich nicht perfekt, denkbar wäre beispielsweise eine Verfeinerung im Sinne Knuths (siehe 2.5.1).

Zur Anzahl der zu wählenden Parameter sei angemerkt, daß der beste bekannte Algorithmus (siehe [6, S.195f.]) zur Lösung eines 0-1-Knapsackproblems (nach 3.3.1) $O(2^{\frac{2}{n}})$ Schritte benötigt, so daß $n > 100$ als durchaus sicher zu betrachten ist.

Für die eigentliche Ver- bzw. Entschlüsselung von Daten muß lediglich noch ein geeigneter Startwert erzeugt werden, der als einzige Bedingung eine Länge von n Bits aufweisen muß.

Wie stets werden die mit Hilfe dieses Generators erzeugten Pseudozufallsbits modulo 2 zu den einzelnen Bits des Klartextes addiert, so daß Ver- und Entschlüsselung identische Operationen sind. Verfügen beide Kommunikationspartner über denselben Parametersatz, ist im Betriebsfalle lediglich der jeweils verwandte Startwert $\vec{x}$ zu übermitteln.

4.4.2 Effizienz

Wie man anhand von (3.101) sieht, eignet sich das vorliegende Verfahren ausgesprochen gut für die Implementation auf einer Mehrprozessor-Anlage, da die verwandte Summation leicht in unabhängige Teile aufgespalten werden kann, so daß jedem einzelnen Prozessor ein Teil des Parametersatzes zur Verarbeitung zukommt. Da hierbei zwischen den einzelnen Prozessoren keinerlei Kommunikationsaufkommen erforderlich ist – vom Sammeln der Teilsummen am Ende eines Programmlaufes einmal abgesehen – ist hiermit prinzipiell ein linearer Speedup zu verwirklichen.

Können derartige Mehrprozessor-Architekturen nicht genutzt werden, wie dies in der vorliegenden Implementation der Fall ist, weist das Verfahren naturgemäß das schlechteste Laufzeitverhalten aller betrachteten Pseudozufallsgeneratoren auf. Laufzeiten im Bereich von etlichen CPU-Minuten auf einer VAXstation 4000 Model 90 für die Generierung nur weniger tausend Pseudozufallsbytes sind nicht ungewöhnlich.

4.4.3 Die Impagliazzo-Naor–Routinen

Im Vordergrund eines jeden Ver- oder Entschlüsselungsvorhabens steht die Generierung eines geeigneten Schlüssels, die im vorliegenden Verfahren deutlich komplexer als gewohnt ausfällt, da nicht nur ein eigentlicher Startwert gewählt, sondern zuallererst ein Parametersatz für die Summationen erzeugt werden muß. Dieser Parametersatz allerdings kann für viele Läufe des Ver-

fahrens Anwendung finden – er stellt das *Geheimnis* des Impagliazzo-Naor–Pseudozufallsgenerators dar.

Zunächst bleibt noch anzumerken, daß auch das vorliegende Verfahren bibliotheksintern auf einer Reihe globaler Variablen und Felder beruht, die vor Anwendung des Systems entsprechend initialisiert werden müssen. Auch arbeitet die Funktion zur Erzeugung eines Parametersatzes selbst nur auf diesen globalen Speicherbereichen, die Programmteilen außerhalb der eigentlichen Bibliothek selbst nicht direkt zugänglich sind, so daß für die Sicherung eines solchen Parametersatzes im Bedarfsfalle weitere Bibliotheksroutinen bemüht werden müssen.

Die Erzeugung eines neuen Parametersatzes:

Im Mittelpunkt dieses Aufgabenkomplexes steht die Funktion *inaor_gen_par_set*, deren praktischen Aufruf folgendes Programmfragment verdeutlicht:

```
int status, par_anz, par_len, i, j, seed_len;
char *file_name;
unsigned char *klartext, *chiffre;
MP_INT *parameter [], seed;

/* Vor dem eigentlichen Aufruf muessen par_anz bzw.  */
/* par_len auf die gewuenschte Anzahl und Laenge der */
/* zu erzeugenden Parameter gesetzt werden.          */

status = inaor_gen_par_set (par_anz, par_len);
switch (status)
{       case RC$NORMAL    : break;
case RC$VALERR    : /* par_len < par_anz oder      */
                            /* par_len > par_anz * INAOR_C */
                            /* Sprung zu Fehlerbehandlung. */
case RC$PAROVL    : /* par_num < 1 ||              */
                            /* par_anz > INAOR$MAXPAR      */
                            /* Sprung zu Errorhandler.     */
case RC$NOT_CLEANED_UP :
                            /* Das Verfahren war bereits   */
                            /* initialisiert, vor neuem    */
                            /* Aufruf inaor_cleanup ()     */
                            /* aufrufen.                   */
}
```

Diese Routine vollzieht neben der Generierung eines Parametersatzes gleichzeitig eine vollständige Initialisierung des Impagliazzo-Naor–Verfahrens, die lediglich um einen Startwert ergänzt werden muß, um mit den eigentlichen Iterationen zu beginnen. Hierbei werden die globalen Variablen für Anzahl und Länge der Parameter, sowie das eigentliche Parameterfeld vorbesetzt.

Das bibliotheksinterne globale Parameterfeld, das die zu verwendenden Parameterwerte aufnimmt, kann *nicht* dynamisch erweitert werden! Die in der Datei *bitstream_lib.h* definierte Konstante *INAOR$MAXPAR* legt die maximale Größe dieses Arrays fest, die (außer durch Änderung des betreffenden Wertes und Neuübersetzung) nicht verändert werden kann. Dieser Weg wurde gewählt, obwohl die GNU-MP Bibliothek die Funktion *mpz_array_init* enthält, die die Generierung beliebiger Felder des Typs *MP_INT* gestattet. Einerseits sind die einzelnen Feldelemente eines durch Aufruf von *mpz_array_init* generierten Arrays nicht in ihrer Länge veränderlich, andererseits kann der durch ein solches Feld belegte Speicherplatz nicht mehr freigegeben werden, was letztlich den Ausschlag zugunsten der vorliegenden Implementation mit nicht-dynamischen Feldern lieferte.

Zur Sicherung eines solchermaßen erzeugten Parametersatzes steht die folgende Funktion zur Verfügung (das Programmfragment knüpft an die obigen Zeilen an):

```
/* Der Pointer file_name zeigt auf einen String, der den */
/* Namen des zu schreibenden Files enthaelt.             */

status = inaor_save_par_set (&file_name);
switch (status)
{       case RC$NORMAL     : break;
        case RC$FOWERR     : /* Das File konnte nicht fuer */
                             /* schreibenden Zugriff ge-   */
                             /* oeffnet werden!            */
                             ... ... ... ... ... ... ... ...
        case RC$NOT_INITIALIZED :
                             /* Es exisitiert kein globaler*/
                             /* Parametersatz!             */
                             ... ... ... ... ... ... ... ...
}
```

Neben dieser Möglichkeit, einen Parametersatz auf ein File zu sichern, können auch mit Hilfe der beiden folgenden Routinen die Parameterwerte des globalen Parameterfeldes ermittelt und kopiert werden. Folgende Programmzeilen ermöglichen die Bestimmung der Parameterarray-Charakteristika Anzahl und Länge:

```
status = inaor_characteristics (&i, &j);
switch (status)
{       case RC$NORMAL     : break;
        case RC$NOT_INITIALIZED :
                             /* Das System ist nicht korrekt */
                             /* initialisiert, es existiert  */
                             /* kein Parametersatz.          */
```

```
                          ... ... ... ... ... ... ... ...
}

/* Nach diesen Zeilen enthalten i und j Anzahl sowie Laenge  */
/* der globalen Parameterelemente.                           */
```

Nachdem nun auf diese Weise die den Parametersatz kennzeichnenden Eigenschaften bestimmt werden konnten, ermöglicht die folgende Funktion ihre Kopie auf Feldelemente des rufenden Programmes:

```
status = inaor_get_par_set (i, &parameter);
switch (status)
{       case RC$NORMAL     : break;
        case RC$NOT_INITIALIZED :
                             /* Das System ist nicht korrekt */
                             /* initialisiert, es existiert  */
                             /* kein Parametersatz.          */
                             ... ... ... ... ... ... ... ...
}

/* Nun enthaelt das lokale Feld parameter eine Kopie des  */
/* globalen Parametersatzes. Die benoetigten Feldelemente */
/* werden innerhalb der Funktion entsprechend allokiert.  */
```

Außer den genannten Methoden zur Erzeugung und Speicherung eines Parametersatzes existiert eine Reihe von Funktionen, mit deren Hilfe ein Parametersatz mit vorgegebenen Werten belegt werden kann. Zunächst ist die Möglichkeit des Ladens eines Parametersatzes von einem spezifizierten File zu nennen:

```
/* Vor dem Aufruf der folgenden Funktion muss sicher- */
/* gestellt sein, dass file_name auf den Namen eines  */
/* geeigneten Parameterfiles zeigt...                 */

status = inaor_load_par_set (&file_name);
switch(status)
{ case RC$NORMAL         : break;
        case RC$FORERR         : /* Das File konnte nicht */
                                 /* geoeffnet werden.     */
        case RC$PAROVL         : /* number < 1 ||         */
                                 /* number > INAOR$MAXPAR */
        case RC$VALERR         : /* len < 1 ||            */
                                 /* len > number * INAOR_C*/
        case RC$EOFENC         : /* Waehrend des Lesens   */
                                 /* wurde unerwartet das  */
                                 /* Fileende erreicht!    */
        case RC$NOT_CLEANED_UP : /* Das Verfahren ist     */
```

```
                                    /* bereits initialisiert.*/
                                    /* Der alte Parametersatz*/
                                    /* bleibt erhalten.      */
}
```

Eine weitere Methode, den globalen Parametersatz mit bestimmten Werten zu belegen, besteht in einem Kopiervorgang eines Parameterfeldes des rufenden Progammabschnitts in das globale Feld:

```
/* Der folgende Funktionsaufruf setzt voraus, dass zum  */
/* einen Anzahl und Laenge der Parameter in i resp. j   */
/* abgelegt sind, und zum anderen das lokale Feld       */
/* parameter die entsprechenden Parameterwerte enthaelt.*/

status = inaor_set_par_set (par_num, par_len, &parameter);
switch(status)
{       case RC$NORMAL         : break;
        case RC$PAROVL         : /* number < 1 ||         */
                                 /* number > INAOR$MAXPAR */
        case RC$VALERR         : /* len < 1 ||            */
                                 /* len > number * INAOR_C*/
        case RC$NOT_CLEANED_UP : /* Das Verfahren ist     */
                                 /* bereits initialisiert.*/
                                 /* Der alte Parametersatz*/
                                 /* bleibt erhalten.      */
}
```

Wie dies bereits bei *inaor_gen_par_set* der Fall war, initialisieren auch die beiden Routinen *inaor_load_par_set* bzw. *inaor_set_par_set* das Verfahren in all seinen globalen Variablen, den Startwert ausgenommen, auf dessen Erzeugung bzw. Belegung nun eingegangen wird:

Zur Generierung eines neuen Startwertes für das vorliegende Verfahren dient die Routine *inaor_gen_seed*:

```
/* Der erfolgreiche Aufruf der folgenden Routine */
/* erwartet die Laenge des zu erzeugenden Seeds  */
/* in l, der Startwert selbst wird ueber seed    */
/* zurueckgeliefert.                             */

status = inaor_gen_seed (seed_len, &seed);
switch (status)
{       case RC$NORMAL : break;
        case RC$VALERR : /* l < 1. */
                         ... ... ...
}
```

Weiterhin kann ein bereits gegebener Startwert an das Verfahren übergeben werden:

```
/* seed ist ein MP_INT-Objekt, das den zu */
/* verwendenden Startwert enthaelt.       */

status = inaor_set_seed (&seed);
switch (status)
{       case RC$NORMAL          : break;
        case RC$NOT_INITIALIZED : /* Das Verfahren besitzt  */
                                  /* noch keinen Parameter- */
                                  /* satz! Der Startwert    */
                                  /* wurde nicht kopiert!   */
}
```

Die umgekehrte Funktion zur Ermittlung des aktuellen Seeds des Verfahrens hat folgende Aufrufgestalt:

```
status = inaor_get_seed (&seed);
switch (status)
{       case RC$NORMAL          : break;
        case RC$NOT_INITIALIZED : /* Das Verfahren besitzt  */
                                  /* noch keinen Parameter- */
                                  /* satz! Der Startwert    */
                                  /* wurde nicht kopiert!   */
}

/* Nach diesem Aufruf enthaelt seed den Wert des */
/* bibliotheksglobalen Startwertes.              */
```

Wurde ein Startwert mittels einer der beiden eben genannten Methoden erzeugt, ist das Verfahren, wenn mit Hilfe einer der im vorangegangenen beschriebenen Funktionen auch der Parametersatz gesetzt resp. generiert wurde, vollständig initialisiert! In diesem Falle kann mit der Erzeugung pseudozufälliger Bytes nach dem Verfahren von Impagliazzo-Naor fortgefahren werden, mit deren Hilfe ein Strom gegebener Klartextzeichen chiffriert werden kann:

```
/* Beliebige Schleife, um alle Klartextzeichen zu */
/* verschluesseln...                              */
{
        *chiffre++ = *klartext++ ^ inaor_prg_iterate ();
}
```

Auch diese Funktion setzt (wie alle Iterationsfunktionen dieser Bibliothek) eine korrekte Initialisierung des ihr zugrundeliegenden Verfahrens durch einen vorangegangenen Aufruf von *inaor_init* zwingend voraus!

Beiinhalteten bereits die Implementationen der Verfahren nach Blum-Blum-Shub bzw. Micali-Schnorr jeweils eine spezielle Funktion zur Freigabe nicht

Verfahren	Länge von $\vec{k}$ in Bits
DES	64
D3DES	192
IDEA	128

Tabelle 4.1: DES-, D3DES- und IDEA-Schlüssellängen für $\vec{k}$

mehr benötigten Speicherplatzes, so ist dies auch und vor allem bei dem vorliegenden Pseudozufallsgenerator notwendig, da er für die Speicherung des Parametersatzes mehr Speicherplatz als alle anderen Systeme benötigt, dessen Freigabe nach seiner Verwendung nicht vergessen werden sollte:

```
status = inaor_cleanup ();
switch (status)
{       case RC$NORMAL          : break;
        case RC$NOT_INITIALIZED : /* Das System ist nicht */
                                  /* initialisiert, es    */
                                  /* kann kein Speicher   */
                                  /* deallokiert werden.  */
}
```

4.5 DES, D3DES, IDEA und MDx im OFB-Mode – Implementation

Analog zu den drei bisher behandelten Verfahren soll nun zunächst die Erzeugung und Verwendung geeigneter Schlüssel für die vier genannten Pseudozufallsgeneratoren betrachtet werden:

Die auf DES, D3DES, sowie IDEA beruhenden Systeme verwenden in der OFB-Betriebsart Schlüssel prinzipiell gleicher Struktur. Im folgenden sei $\vec{s} = (s_0, \ldots, s_n)$ ein den Schlüssel repräsentierender Bitvektor der Länge n. Dieser wird in zwei Teilvektoren $\vec{k}$ und $\vec{k'}$ aufgespalten. Hierbei wird $\vec{k}$ als eigentlicher Schlüssel des zugrundeliegenden Blockchiffreverfahrens DES oder IDEA verwendet, während $\vec{k'}$ als Initialisierungsvektor für den Eingabeblock des Verfahrens dient.

Sowohl DES und D3DES, als auch IDEA arbeiten auf 8-Byte Datenblöcken, so daß in diesen drei Fällen $\vec{k'}$ eine Länge von 64 Bits aufweist. Unterschiedlich sind hingegen die Längen der eigentlichen Schlüssel für die verschiedenen Verfahren, wie Tabelle 4.1 zeigt.

Hieraus ergeben sich für $\vec{s}$ Längen von 128, 256 bzw. 192 Bits für die genannten drei Blockchiffreverfahren im OFB-Mode[8].

Die Verwendung der MDx-Hashfunktionen zur Implementation eines Pseudozufallszahlengenerators unterscheidet sich von den bereits genannten drei

[8]Zu beachten ist bei Anwendung des DES- bzw. D3DES-Systems, daß der eigentliche Schlüssel byteweise ungerade Parität aufweisen muß.

Zu selektierendes Verfahren	*mode*	*sub_mode*
DES	*OFB$DES*	*DES$MODE_DES*
D3DES	*OFB$D3DES*	*DES$MODE_D3DES*
IDEA	*OFB$IDRAN*	
MD2	*OFB$MDRAN*	*MDRAN$MODE_MD2*
MD4	*OFB$MDRAN*	*MDRAN$MODE_MD4*
MD5	*OFB$MDRAN*	*MDRAN$MODE_MD5*

Tabelle 4.2: Vordefinierte Konstanten für *mode* bzw. *sub_mode*

Verfahren lediglich dadurch, daß der Schlüssel $\vec{s}$ in seiner Gesamtlänge von hierbei 128 Bits als Iterationsstartwert Anwendung findet, so daß eine Unterteilung in einen eigentlichen Schlüssel $\vec{k}$ des Verfahrens, sowie einen Startwert $\vec{k'}$ entfällt.

Bevor nun die Erzeugung eines neuen Schlüssels für ein beliebiges der oben genannten Verfahren demonstriert wird, muß zuvor die Organisation der Bibliotheksroutinen, die diese vier unterschiedlichen Generatoren zusammenfassen, kurz erläutert werden:

Die Gleichartigkeit ihrer Parameter sowie ihrer Funktionsweise legte den Gedanken nahe, die auf DES, D3DES, IDEA bzw. MDx basierenden Generatoren in einer Gruppe von Routinen zusammenzufassen, deren einzelne Funktionen stets mit dem Kürzel *ofb_* beginnen. Fast alle Routinen dieser Klasse besitzen zwei Parameter *mode* bzw. *sub_mode* des Typs *int*, durch die der gewünschte Generator angewählt wird. Durch geeignetes Belegen von *mode* wird zwischen DES-, IDEA- und MDx-basierten Generatoren unterschieden, während *sub_mode* innerhalb dieser Teilgruppierungen eine Unterscheidung zwischen DES/D3DES bzw. MD2/MD4/MD5 ermöglicht.

Für diese Form der Auswahl finden sich entsprechend definierte Konstanten in *bitstream_lib.h*, die in Tabelle 4.2 aufgelistet sind.

Die eigentliche Schlüsselerzeugung für eines der genannten vier Verfahren (wobei die drei unterschiedlichen Hashfunktionen MD2, MD4, sowie MD5 in dieser Hinsicht als ein einziges Verfahren gezählt werden, da sie sich hinsichtlich ihres Startwertaufbaus nicht unterscheiden), kann mit Hilfe der Bibliotheksroutinen *ofb_new_key* bzw. *ofb_new_hex_key* vollzogen werden, deren Aufruf der folgende Programmausschnitt beispielhaft für die erstgenannte Funktion darstellt:

```
int status, mode, sub_mode;
unsigned char *key;

/* Vor dem eigentlichen Aufruf der Schluesselerzeugung */
/* muessen mode, sowie sub_mode mit den gewuenschten   */
/* Werten fuer die Anwahl des betreffenden Verfahrens  */
/* belegt werden. Weiterhin muss eine entsprechende    */
```

```
/* Menge an Speicherplatz fuer key allokiert werden.   */
/* Im folgenden soll ein Schluessel fuer den auf dem   */
/* im OFB-Mode betriebenen D3DES-Verfahren erzeugt     */
/* werden:                                             */

mode = OFB$DES;
sub_mode = DES$MODE_D3DES;

status = ofb_new_key (key, mode, sub_mode);
if (status != RC$NORMAL)
        /* Entweder mode oder sub_mode enthielten */
        /* einen unzulaessigen Wert.              */
```

Nach diesem Aufruf enthält der String, der durch *key* adressiert wird, einen zulässigen Binärschlüssel für den gewünschten Pseudozufallsgenerator. Soll im Unterschied hierzu ein Schlüssel in hexadezimaler Darstellung generiert werden, kann die Routine *ofb_new_hex_key* eingesetzt werden. Zu beachten ist in diesem Falle lediglich, daß das erste Argument dieser Funktion ein Pointer des Typs *char* ist, der auf einen Speicherbereich doppelter Länge des für *ofb_new_key* benötigten zeigt, da zur hexadezimalen Darstellung eines jeden Bytes des zugrundeliegenden eigentlichen Binärschlüssels jeweils zwei Zeichen vorgesehen werden müssen.

Außer den beiden genannten Routinen steht eine weitere Funktion zur Generierung eines Schlüssels für die OFB-Pseudozufallsgeneratoren zur Verfügung, die es ermöglicht, aus einer gegebenen Passwortzeichenkette einen gültigen, reproduzierbaren Binärschlüssel zu erzeugen: *ofb_pwd_to_key*.

Diese Funktion hasht zunächst das übergebene Passwort unter Zuhilfenahme einer der MDx-Funktionen, um so einen passwortabhängigen 64-Bit-Wert zu generieren, der im nachfolgenden Schritt als Startwert für einen DES-basierten Pseudozufallsgenerator verwandt wird, aus dessen Ausgabebits der gewünschte Binärschlüssel aufgebaut wird. Der Funktionskopf von *ofb_pwd_to_key* hat folgende Gestalt:

```
int ofb_pwd_to_key (unsigned char *key, char *pwd,
                    int mode, int sub_mode,
                    int mdx, int iterations,
                    int bits_per_step, int enc_dec);
```

Der erste Übergabeparameter ist ein Pointer auf eine Kette vorzeichenloser Bytes, die der Aufnahme des zu erzeugenden Binärschlüssels dienen, wohingegen das zweite Argument einen Pointer auf den Passwortstring darstellt.

Durch entsprechendes Setzen der beiden folgenden Werte *mode* bzw. *sub_mode* wird festgelegt, für welches der OFB-Verfahren ein neuer Schlüssel zu erzeugen ist (hierfür sind alle Werte aus Tabelle 4.2 zulässig.).

Die Parameter *mdx* und *iterations* dienen der Steuerung des Passworthashings; der in *mdx* übergebene Wert legt fest, welches der drei MDx-Hashverfahren verwendet werden soll, wohingegen die in *iterations* vorgegebene Ganzzahl festlegt,

wieviele Hashiterationen zur Erzeugung des 64-Bit Startwertes für das eingebettete DES-Verfahren vollzogen werden sollen.

Die beiden letzten Argumente *bits_per_step* bzw. *enc_dec* dienen der Kontrolle des verwendeten Pseudozufallsgenerators auf Basis des im OFB-Modus betriebenen DES-Systems. Durch *bits_per_step* wird festgelegt, wieviele Bits pro Iterationsschritt des Generators zur Erzeugung des endgültigen Schlüssels extrahiert werden sollen. *enc_dec*, für das die beiden (ebenfalls in *bistream_lib.h* definierten) Konstanten *DES$ENCRYPT* bzw. *DES$DECRYPT* vorgesehen sind, legt fest, ob das dem Pseudozufallsgenerator zugrundeliegende DES-System im Ver- bzw. Entschlüsselungsmodus betrieben werden soll.

Zu Demonstration dieser Funktion soll nun aus einem gegebenen Passwort ein Binärschlüssel für den MD5-basierten Pseudozufallsgenerator abgeleitet werden:

```
int status, mdx, iterations, bps, enc_dec;
unsigned char *key;
char *pwd;

/* Allokieren des fuer den zu erzeugenden    */
/* Schluessel benoetigten Speicherplatzes,   */
/* vor dem eigentlichen Aufruf der Funktion  */
/* muss weiterhin gewaehrleistet sein, dass  */
/* pwd auf das gewuenschte Passwort zeigt... */
/* Zudem muessen mdx, iterations, bps, sowie */
/* enc_dec entsprechend belegt werden.       */

status = ofb_pwd_to_key (key, pwd, OFB$MDRAN, MDRAN$MODE_MD5,
                         mdx, iterations, bps, enc_dec);
switch (status)
{       case RC$NORMAL : break;
        case RC$WRONG_MODE : /* mode, submode oder mdx    */
                             /* enthielten einen unzu-    */
                             /* laessigen Wert.           */
        case RC$VALERR :     /* iterations < 1         || */
                             /* bps < 1 || bps > 64    || */
                             /*(enc_dec != DES$ENCRYPT && */
                             /* enc_dec != DES$DECRYPT)   */
}
```

Nachdem auf eine der beschriebenen drei Arten ein Schlüssel für einen der DES-, IDEA- oder MDx-basierten Pseudozufallsgeneratoren der OFB-Teilbibliothek erzeugt wurde, muß das betreffende System vor seiner eigentlichen Verwendung initialisiert werden. Hierfür steht die Funktion *ofb_init* zur Verfügung, die alle zur Laufzeit benötigten bibliotheksglobalen Variablen des betreffenden Generators setzt.

Wurde beispielsweise ein Schlüssel für den IDEA-basierten Generator erzeugt, der im folgenden für die Generierung pseudozufälliger Bytes zur Ver-

bzw. Entschlüsselung von Nutzdaten eingesetzt werden soll, kann dieser mit folgendem Funktionsaufruf initialisiert werden:

```
int status, bits_per_step, mode, sub_mode = 0;
unsigned char *key;

/* key zeigt auf den zu verwendenden Binaerschluessel, */
/* bits_per_step sei mit dem gewuenschten Wert ini-    */
/* tialisiert.                                         */

mode = OFB$IDRAN;

status = ofb_init (mode, sub_mode, key, bits_per_step);

/* In diesem Beispiel wird der Wert von sub_mode */
/* nicht ausgewertet, da durch mode = OFB$IDRAN  */
/* ein Verfahren ausgewaehlt wurde, das keine    */
/* weitere Untergruppierung besitzt.             */

switch (status)
{       case RC$NORMAL      : break;
        case RC$WRONG_MODE  : /* mode oder sub_mode    */
                              /* enthielen einen un-   */
                              /* zulaessigen Wert.     */
        case RC$VALERR      : /* Der in bits_per_step  */
                              /* uebergebene Wert lag  */
                              /* ausserhalb der fuer   */
                              /* das ausgewaehlte Ver- */
                              /* fahren zulaessigen    */
                              /* Grenzen.              */
        case RC$ILLEGAL_KEY : /* Wurde der DES- bzw.   */
                              /* D3DES-basierte Gene-  */
                              /* rator angewaehlt,     */
                              /* wird dieser Rueck-    */
                              /* gabewert erhalten,    */
                              /* falls eines der ersten*/
                              /* 8 Schluesselbytes     */
                              /* gerade Paritaet be-   */
                              /* sitzt.                */
```

Nach erfolgreicher Initialisierung des gewünschten Verfahrens kann mit der eigentlichen Erzeugung pseudozufälliger Bytes begonnen werden:

```
unsigned char *chiffre, *klartext;

/* Im folgenden sei vorausgesetzt, dass sowohl   */
```

mode	Das selektierte Verfahren ist	
	initialisiert	nicht initialisiert
OFB$DES	*DES$SETUP_CORRECTLY*	*DES$NOT_YET_SETUP*
OFB$IDRAN	*IDRAN$SETUP_CORRECTLY*	*IDRAN$NOT_YET_SETUP*
OFB$MDRAN	*MDRAN$SETUP_CORRECTLY*	*MDRAN$NOT_YET_SETUP*
OFB$BBS	*BBS$SETUP_CORRECTLY*	*BBS$NOT_YET_SETUP*
OFB$INAOR	*INROA$SETUP_CORRECTLY*	*INAOR$NOT_YET_SETUP*
OFB$MSCHNORR	*MSCHNORR$SETUP_CORRECTLY*	*MSCHNORR$NOT_YET_SETUP*

Tabelle 4.3: Aufruf- und Rückgabewerte für *ofb_get_status*

```
/* chiffre, als auch klartext auf einen ent-     */
/* sprechenden Ziel- bzw. Quellstring verweisen. */

/* Schleife zur Verarbeitung aller zu */
/* verschluesselnden Zeichen...       */
{
        *chiffre++ = *klartext++ ^ ofb_iterate ();
}
```

Mit Hilfe der vorgestellten Bibliotheksroutinen kann die gesamte Funktionalität der auf den DES-, D3DES- bzw. IDEA-Blockchiffren und den MD2-, MD4- bzw. MD5-Hashfunktionen beruhenden Pseudozufallszahlengeneratoren genutzt werden.

Die beiden Routinen *ofb_init* bzw. *ofb_iterate* verwenden ihrerseits folgende Routinen (deren Aufrufkonventionen in Anhang A beschrieben werden), die an dieser Stelle kurz erwähnt seien, falls sie alleinstehend und ohne den Überbau der OFB-Routinen genutzt werden sollen:

DES-/D3DES-basiert: *des_init* bzw, *des_prg_iterate*.

IDEA als Grundlage: *idran_init* und *idran_prg_iterate*.

Auf MDx beruhend: *mdran_init*, *mdran_prg_iterate*.

Abschliessend sei die letzte verbliebene OFB-Funktion beschrieben: Sie greift über die eigentlichen OFB-Routinen hinaus und gestattet es, den Zustand eines *jeden* implementierten Pseudozufallsgenerators zu betrachten:

```
int ofb_get_status (int mode);
```

Die für *mode* zulässigen Übergabewerte, sowie die zu erwartenden Rückgabewerte der Funktion listet Tabelle 4.3 auf[9].

[9]Ein unzulässiger Wert für *mode* hat als Returncode *RC$WRONG_MODE* zur Folge!

Kapitel 5

Ausblick

Im folgenden seien einige Gedanken zu Erweiterungsmöglichkeiten der vorgestellten Verfahren geäußert, die in der vorliegenden Implementation nicht berücksichtigt oder verfolgt wurden. Sie beziehen sich ausschließlich auf die Verfahren von Blum-Blum-Shub, Micali-Schnorr bzw. Impagliazzo-Naor – den bekannten Blockchiffre-Verfahren in der OFB-Betriebsart sei nichts weiter hinzugefügt.

Problematisch bei jedem dieser drei Verfahren ist in erster Linie das Laufzeitverhalten, das auf die zur Gewährleistung der kryptographischen Sicherheit benötigten großen Wertebereiche zurückzuführen ist, so daß in erster Linie Methoden zur Verringerung des Zeitaufwandes für die Erzeugung einer bestimmten Anzahl pseudozufälliger Werte von Interesse sind.

Sofern ein geeigneter Parallelrechner zur Verfügung steht, ist sicherlich die Parallelisierung solcher Pseudozufallszahlengeneratoren die naheliegendste Variante zur Erzielung eines verbesserten Laufzeitverhaltens. Zumindest eines der beschriebenen Verfahren – der Generator nach Impagliazzo-Naor – legt eine derartige Implementationsvariante, wie bereits in Abschnitt 4.4.2 bemerkt wurde, direkt nahe.

Für andere Verfahren, die nicht direkt einer Parallelisierung zugänglich scheinen, sei auf ein interessantes, allgemeines Verfahren zur Parallelisierung von Pseudozufallszahlengeneratoren verwiesen, das in Anhang E kurz angerissen wird. Das dort vorgestellte Prinzip ermöglicht die Parallelisierung aller perfekten Pseudozufallsgeneratoren mit einem linearen Geschwindigkeitszuwachs, da das benötigte Kommunikationsaufkommen zwischen den einzelnen Prozessoren minimal ist und keine bremsenden Datenabhängigkeiten im Verlauf einer Berechnung auftreten.

Mit dem Vorhandensein eines Parallelrechners jedoch stehen und fallen derartige Ansätze zur Leistungssteigerung derartiger Verfahren zur Erzeugung pseudozufälliger Zahlen. Ist man auf die Verwendung eines Einprozessorsystems angewiesen, halte ich es für denkbar, verschiedene Verfahren miteinander zu kombinieren, wobei auch die Anwendung kryptographisch nicht sicherer Verfahren zu überlegen ist. Beispielsweise könnte ein einfacher linearer Kongruenzgenerator zur *Verstärkung* der Anzahl von Ausgabewerten eines Blum-Blum-

Shub–Generators o.ä. verwandt werden.

Denkbar wäre eine häufige Neuinitialisierung eines solchen einfachen Generators (LKG, etc.) durch ein kryptographisch sicheres Verfahren. Werden von diesem nachgeschalteten Generator nur vergleichsweise kurze Zahlenfolgen zwischen zwei von dem übergeordneten Verfahren ausgehenden Initialisierungsschritten erzeugt, von denen zudem unter Umständen nur wenige Bits verwandt werden, könnte auf diese Art und Weise auch auf einfachen Rechnern ein nicht unerheblicher Geschwindigkeitszuwachs erzielt werden[1]. Dennoch muß der kryptographischen Sicherheit eines derartigen Mischverfahrens im Vorfeld einer realen Implementation viel Aufmersamkeit geschenkt werden.

Ebenso wäre die Frage nach der Verwendbarkeit der in jedem Schritt des Verfahrens nach Impagliazzo-Naor entstehenden $\lfloor \log_2 n \rfloor$ Übertragsbits zu untersuchen, die in der vorliegenden Implementation durch Ausführung der Additionen Modulo $2^{l(n)}$ unterdrückt werden[2].

[1] Auch in Verbindung mit dem in Anhang E erläuterten Verfahren zur Parallelisierung perfekter Generatoren ist der obige Vorschlag denkbar. Ausgehend von einem Startwert (erzeugt durch ein perfektes Verfahren), könnten einige Ebenen des entsprechenden Berechnungsbaumes von einfacheren und effizienteren Verfahren mit Werten belegt werden, bis wieder ein kryptographisch sicherer Algorithmus zum Einsatz kommt, usf.

[2] Zu klären ist, ob die Ausgabe dieser zusätzlichen Bits auf Kosten der kryptographischen Sicherheit zuviel Information über den inneren Zustand des Generators preisgegeben würde.

Anhang A

Kurzreferenz zu den Bibliotheksfunktionen

A.1 Rückgabewerte

Der folgende Abschnitt listet die möglichen Rückgabewerte der einzelnen Bibliotheksfunktionen auf, die in *bitstream_lib.h* definiert werden und bei Verwendung der Funktionen abgefangen werden müssen:

A.1.1 Allgemeine Rückgabewerte

RC$NORMAL Die Funktion wurde korrekt abgeschlossen.

RC$FOWERR Ein angefordertes File konnte nicht für schreibenden Zugriff geöffnet werden, die Funktion wurde nicht vollständig ausgeführt.

RC$FORERR Das Öffnen eines benötigten Files für Lesezugriff konnte nicht korrekt vollzogen werden.

RC$EOFENC Beim Lesen eines Files wurde ein unerwartetes EOF angetroffen. Die Funktion wurde abgebrochen.

RC$VALERR Überschreitung des Wertebereichs eines Übergabeparameters.

RC$PAROVL Es wurde der Versuch unternommen, für das Verfahren von Impagliazzo-Naor einen zu kleinen bzw. zu großen Parametersatz zu generieren.

RC$WRONG_MODE Die Funktion wurde mit einem unzulässigen Modus aufgerufen.

RC$ILLEGAL_KEY Der übergebene Schlüssel war unzulässig (d.h. speziell im Fall von DES, er wies Paritätsfehler auf).

RC$NOT_CLEANED_UP Ein Verfahren sollte initialisiert werden, das bereits initialisiert ist. Die Neuinitialisierung wurde nicht vollzogen.

RC$NOT_INITIALIZED Eine Routine, die eine korrekt initialisierte Verfahrensumgebung zwingend voraussetzt, wurde ohne vorangegangenen Initialisierungsschritt aufgerufen.

RC$KEY_NOT_GENERATED Für die Verfahren nach Blum-Blum-Shub bzw. Micali-Schnorr konnte kein geeigneter Schlüssel N als Produkt zweier Primzahlen P und Q mit $P \neq Q$ erzeugt werden. Vor einem erneuten Aufruf der betreffenden Routine sollte eine größere Schlüssellänge gewählt werden.

A.1.2 Spezielle Rückgabewerte

Zu allen sechs implementierten Verfahren gehört eine gemeinsame Routine zur Bestimmung des Zustandes eines jeden (*ofb_get_status*). Hiermit kann leicht überprüft werden, ob bereits eine Initialisierung für den betreffenden Pseudozufallsgenerator vorgenommen wurde, oder nicht. Hierbei können grundsätzlich die beiden folgenden Werte zurückgeliefert werden:

xxx$NOT_YET_SETUP - Das ausgewählte Verfahren wurde noch nicht korrekt initialisiert. (Der Blum-Blum-Shub–Generator beeinhaltet eine Routine, *bbs_next_seed*, die bei fehlerhaftem Aufruf ebenfalls dieses Resultat zurückliefert. Im Erfolgsfalle erhält man entsprechend **RC$NORMAL**.)

xxx$CORRECTLY_SETUP - Der betrachtete Pseudozufallsgenerator wurde korrekt initialisiert.

Für **xxx** in obigen Konstantennamen kann folgendes eingesetzt werden, um allen zur Verfügung stehenden Verfahren gerecht zu werden:

- BBS
- INAOR
- MSCHNORR
- DES
- IDRAN
- MDRAN
- OFB

A.2 Initialisierungsmodi

Drei der implementierten Pseudozufallsgeneratoren besitzen eine gemeinsame Initialisierungsroutine (*ofb_init*), sowie eine gemeinsame Schlüsselerzeugung, die sich aus den Funktionen *ofb_new_hex_key*, *ofb_new_key*, sowie *ofb_pwd_to_key* zusammensetzt. Außer diesen Funktionen existiert für alle sechs Verfahren eine gemeinsame Routine zur Bestimmung ihres Initialisierungszustandes, *ofb_get_status*,

so daß entsprechend sechs verschiedene Modi unterschieden werden. Zur Steuerung der jeweiligen Teilfunktionen können folgende (ebenfalls in *bitstream_lib.h* definierte) Konstanten dienen:

OFB$DES - Wählt den auf dem DES-System im OFB-mode basierenden Pseudozufallsgenerator aus. Hierbei sind zwei Sub-Modi zu beachten, die den betreffenden Routinen in einem weiteren Parameter übergeben werden:

DES$MODE_DES - wählt das einfache DES-Verfahren aus,

DES$MODE_D3DES - D3DES stattdessen.

OFB$IDRAN - Hiermit wird der auf dem im OFB-mode betriebenen IDEA-Verfahren beruhende Pseudozufallsgenerator angesprochen.

OFB$MDRAN - Dieser Wert selektiert den auf den MDx-Hashfunktionen beruhenden Pseudozufallsgenerator, für den ebenfalls weitere Sub-Modi unterschieden werden müssen:

MDRAN$MODE_MD2 - Wählt das MD2-Verfahren als Grundlage des Systems aus,

MDRAN$MODE_MD4 - entsprechend MD4,

MDRAN$MODE_MD5 - MD5.

OFB$BBS - Sowohl diese, als auch die nachfolgenden Konstanten sind nur bei Aufruf der Funktion *ofb_get_status* zulässig. Hiermit wird das BBS-System als Grundlage gewählt.

OFB$INAOR - Setzt entsprechend den auf dem Verfahren von Impagliazzo-Naor beruhenden Pseudozufallsgenerator als Bezugspunkt.

OFB$MSCHNORR - Das Verfahren von Micali-Schnorr wird selektiert.

Zur besseren Übersicht seien die verschiedenen Modi mit ihren zugehörigen Submodi im folgenden kurz tabellarisch aufgelistet:

mode	sub_mode
OFB$BBS	
OFB$INAOR	
OFB$MSCHNORR	
OFB$DES	DES$MODE_DES DES$MODE_D3DES
OFB$IDRAN	
OFB$MDRAN	MDRAN$MODE_MD2 MDRAN$MODE_MD4 MDRAN$MODE_MD5

Da einige der nun aufgelisteten Routinen mehr als nur einen Pseudozufallszahlengenerator beeinflussen können, muß das gewünschte Zielverfahren duch Angabe der betreffenden Modus- und eventuell Submodus-Werte selektiert werden, wie sie sich in obiger Tabelle abgebildet finden.

A.3 Die Routinen im einzelnen

Zur Notation der folgenden Ausführungen sei bemerkt, daß mit *Rückgabewert* stets die Rückgabe eines *Funktionsaufrufs* benannt wird. In vielen Fällen stellt sie lediglich eine Statusinformation dar, während die eigentlichen Ausgaben der Routine über direkt referenzierte Aufrufargumente zurückgeliefert werden. Diese werden mit *Ausgabe* kenntlich gemacht.

A.3.1 DES, IDEA und MDx im OFB-Mode

Die im folgenden Abschnitt enthaltenen Routinen werden in ihrer Gesamtheit als *OFB*-Routinen bezeichnet, sie stellen die eben erwähnten Funktionen dar, deren Zielverfahren durch Mode- und Submode-Argumente näher spezifiziert werden müssen.

ofb_init:

Die vorliegende Funktion ist die vielleicht allgemeinst gehaltene der gesamten Bibliothek, ermöglicht sie doch ein korrektes Setup der sechs auf dem DES-, D3DES- und dem IDEA-Verfahren, sowie auf den MD2,4,5-Hashfunktionen beruhenden Pseudozufallszahlengeneratoren.

```
int ofb_init (int mode, ind sub_mode,
              unsigned char *key, int bits_per_step)
```

Eingabe: In *mode* bzw. *sub_mode* werden gemäß Abschnitt A.2 Steuerinformationen zur Anwahl des gewünschten Verfahrens übergeben, das mit dem in *key* übergebenen Schlüssel, sowie dem Wert von *bits_per_step* initialisiert wird. Hierbei ist zu beachten, daß die Länge des durch *key* referenzierten Schlüssels (der in binärer Form vorliegen muß), nicht überprüft wird! Für die einzelnen Verfahren gelten folgende Schlüssellängen, die beachtet werden müssen:

Verfahren	Schlüssellänge in Bytes
DES	16
D3DES	32
IDEA	24
MDx	16

Ausgabe: Die Routine setzt alle benötigten bibliotheksinternen globalen Variablen für das selektierte Verfahren.

Rückgabewerte: **RC$NORMAL** Die Initialisierung wurde erfolgreich abgeschlossen, das betreffende Verfahren kann im folgenden zur Erzeugung pseudozufälliger Bytes eingesetzt werden.

RC$WRONG_MODE Die in *mode* bzw. *sub_mode* übergebenen Werte sind nicht zulässig.

RC$VALERR Der Wert von *bits_per_step* ist zu klein bzw. zu groß.

RC$ILLEGAL_KEY Dieser Rückgabewert kann nur bei Initialisierung des DES- resp. D3DES-basierten Generators auftreten. Er wird generiert bei falscher Parität des verwendeten Schlüssels.

ofb_new_hex_key:

Mit Hilfe der vorliegenden Funktion kann ein neuer Schlüssel für eines der sechs in dieser Gruppe zusammengefassten Verfahren erzeugt werden, der in Form eines hexadezimalen Strings zurückgeliefert wird.

```
int ofb_new_hex_key (char *hex_key, int mode, int sub_mode)
```

Eingabe: Die Auswahl des betreffenden Zielverfahrens, für das ein neuer Schlüssel zu erzeugen ist, geschieht durch Übergabe geeigneter Werte für *mode* bzw. *sub_mode*. Hierdurch wird funktionsintern die Länge des zu erzeugenden Schlüssels bestimmt, der in dem durch *hex_key* referenzierten String übergeben wird. Wie bereits in der vorangegangenen Funktion wird dieser String nicht auf seine Länge überprüft – hierfür muß das rufende Programm Sorge tragen.

Ausgabe: *hex_key* – dieser String muß die folgenden Längen aufweisen:

Verfahren	Schlüssellänge in Bytes
DES	32
D3DES	64
IDEA	32
MDx	48

Rückgabewerte: **RC$NORMAL** Es konnte ein neuer Schlüssel erzeugt werden.

RC$WRONG_MODE Die Werte für *mode* beziehungsweise *sub_mode* sind nicht zulässig.

ofb_new_key:

Diese Routine gestattet – analog zu der eben erläuterten Funktion *ofb_new_key* – die Erzeugung eines neuen Schlüssels für die OFB-Verfahren. Im Unterschied zu dieser wird jedoch direkt ein binärer Schlüssel statt eines hexadezimalen erzeugt, der ohne weitere Konversion zur Initialisierung über *ofb_init* dienen kann.

```
int ofb_new_key (unsigned char *key,
                 int mode, int sub_mode)
```

Eingabe: Siehe oben.

Ausgabe: Über *key* wird der generierte Schlüssel zurückgeliefert, der denselben Längenbedingungen, wie bereits bei *ofb_init* besprochen, unterliegt.

Rückgabewerte: Siehe *ofb_new_hex_key*.

ofb_pwd_to_key:

Diese Funktion ermöglicht die Generierung eines Binärschlüssels aus einem gegebenen Passwort. Dieses wird mit Hilfe einer MDx-Hashfunktion auf einen 16 Byte Wert abgebildet, der zur Initialisierung eines Pseudozufallszahlengenerators auf Basis des DES-Verfahrens im OFB-Mode verwendet wird. In den jeweiligen Iterationsschritten dieses Generators wird nun jeweils eine – innerhalb gewisser Grenzen – frei wählbare Anzahl von Bits extrahiert, aus denen sich der zu erzeugende Schlüssel zusammensetzt.

```
int ofb_pwd_to_key (unsigned char *key, char *pwd,
                    int mode, int sub_mode,
                    int mdx, int bits_per_step, int enc_dec)
```

Eingabe: Die Auswahl des Zielverfahrens, für das ein neuer Schlüssel generiert werden soll, geschieht wie stets über *mode* resp. *sub_mode*. Das Ausgangspasswort wird dem über *pwd* referenzierten String entnommen, der neue Binärschlüssel wird mit Hilfe von *key* zurückgeliefert.

Der in *mdx* übergebene Wert dient der Auswahl des gewünschten Hash-Verfahrens zur Initialisierung des eingebetteten DES-Generators. Durch *iterations* kann die Anzahl der Hash-Iterationen gesteuert werden – ein Hash-Vorgang wird mindestens ausgeführt.

Die Parameter *bits_per_step*, sowie *enc_dec* dienen der Steuerung des eigentlichen Pseudozufallszahlengenerators. Hierbei gibt *bits_per_step* die Anzahl der in jedem einzelnen Iterationsschritt zu extrahierenden Bits an, während über *enc_dec* bestimmt werden kann, ob das zugrundeliegende DES-Verfahren im Ver- bzw. Entschlüsselungsmodus betrieben werden soll.

Ausgabe: Der neue Schlüssel wird in binärer Form über *key* zurückgeliefert.

Rückgabewerte: **RC$NORMAL** Es wurde ein neuer Schlüssel generiert.

RC$WRONG_MODE *mode* und/oder *sub_mode* enthielten unzulässige Werte.

RC$VALERR Eine der folgenden Beziehungen ist nicht erfüllt:

$$0 < bits\_per\_step < 65 \tag{A.1}$$

$$0 < iterations \tag{A.2}$$

Außer diesen Fehlerquellen besteht die Möglichkeit, daß *enc_dec* einen unzulässigen Wert enthält. (Es sind die in *bitstream_lib.h* definierten Konstanten *DES$ENCRYPT* bzw. *DES$DECRYPT* verwendbar.

Ist der übergebene Passwortstring der Leerstring, wird die Funktion ebenfalls unter Rückgabe dieses Returncodes abgebrochen.

ofb_iterate:

Nach korrektem Setup des betreffenden OFB-Verfahrens durch Aufruf von *ofb_init* kann die vorliegende Funktion zur Erzeugung pseudozufälliger Bytes mit Hilfe des gewünschten Generators verwendet werden.

```
unsigned char ofb_iterate ()
```

Diese Funktion *muß* durch einen vorangegangenen *ofb_init*-Aufruf eine korrekt initialisierte Umgebung vorfinden! In jedem anderen Fall bricht sie mit einer bibliotheksinternen Fehlermeldung ab[1].

Rückgabewerte: Jeder Funktionsaufruf liefert ein pseudozufälliges Byte zurück.

ofb_get_status:

Quasi in Umkehrung der eben beschriebenen Routine gestattet es diese Funktion, den Status eines jeden der sechs verschiedenen Pseudozufallsgeneratoren zurückzuliefern (initialisiert/nicht initialisiert):

```
int ofb_get_status (int mode);
```

Eingabe: mode

Rückgabewerte: **RC$WRONG_MODE** - Die Funktion wurde mit einem unzulässigen Wert für den Parameter *mode* aufgerufen.

RC$xxx_SETUP_CORRECTLY - Das selektierte Verfahren ist vollständig initialisiert.

RC$xxx_NOT_YET_SETUP - Vor weiterer Verwendung muß das betreffende Verfahren initialisiert werden.

ofb_get_keylen:

Diese Funktion ermöglicht es, die für ein bestimmtes Verfahren benötigte Länge des Binärschlüssels zu ermitteln.

```
int ofb_get_keylen (int *key_len, int mode, int sub_mode)
```

Eingabe: Die Anwahl des gewünschten Verfahrens geschieht durch geeignete Wertübergabe in *mode* bzw. *sub_mode*.

Ausgabe: Die Länge des benötigten Binärschlüssels wird über den Pointer *key_len* zurückgeliefert.

Rückgabewerte: **RC$NORMAL** - Die Funktion wurde korrekt beendet.

RC$WRONG_MODE - *mode* oder *dub_mode* enthielten unzulässige Werte.

[1] Dies gilt im übrigen für alle der implementierten Iterationsfunktionen, was manchem unelegant erscheinen mag. Dennoch kann mit dieser Konvention auf weitere Abfragen innerhalb einer Verschlüsselungsroutine, die lediglich eine Reihe von *ofb_iterate*-Aufrufen tätigt, verzichtet werden. Die betreffende Stelle ist im Quellcode leicht kenntlich und kann auf andere Bedürfnisse ohne großen Aufwand abgestimmt werden.

key_to_hex:

Hiermit wird die Umwandlung eines Binärschlüssels in seine hexadezimale Darstellung ermöglicht.

```
void key_to_key (char *hex_key, unsigned char *key, int len)
```

Eingabe: *key* stellt einen Pointer auf den zu konvertierenden Schlüssel dar, dessen Länge in *len* übergeben wird.

Ausgabe: Die Hexadezimaldarstellung des Ausgangsschlüssels wird über *hex_key* in einem String abgelegt, dessen Länge nicht überprüft wird. Sie muß dem doppelten des in *len* übergebenen Wertes entsprechen!

hex_to_key:

Diese Funktion stellt das Gegenstück zu der eben beschriebenen Routine dar und gestattet die Konversion eines in Hexadezimalform vorhandenen Schlüssels in einen entsprechenden binären Schlüssel.

```
int hex_to_key (unsigned char *key, char *hex_key, int len)
```

Eingabe: *hex_key* stellt einen Zeiger auf den umzuwandelnden Schlüsseltext dar. *len* enthält die Länge des *Zielschlüssels* in Bytes. Dieser Wert entspricht folglich der halben Länge des Strings *hex_key*.

Ausgabe: über *key* wird die Binärdarstellung des Ausgangsstrings in einen String kopiert.

Rückgabewerte: **RC$NORMAL** Der übergebene Schlüssel konnte korrekt konvertiert werden.

RC$ILLEGAL_KEY Die Länge des über *hex_key* adressierten Quellstrings betrug nicht das doppelte des in *len* übergebenen Wertes.

Die folgenden Routinen implementierten die einzelnen Pseudozufallszahlengeneratoren, die durch die OFB-Funktionen zusammengefaßt werden. In den meisten Fällen ist ihre direkte Verwendung nicht notwendig, da die zuvor beschriebenen Bibliotheksfunktionen für die meisten Zwecke hinreichend sind. Der Vollständigkeit halber seien sie jedoch auch kurz erläutert:

idran_init:

Diese Funktion dient der Initialisierung des auf dem im OFB-Modus betriebenen IDEA-Verfahren beruhenden Generators. (Im Normalfall wird diese Routine indirekt über *ofb_init* aufgerufen.)

```
int idran_init (int bits_per_step, unsigned char *key)
```

Eingabe: *bits_per_step* enthält die Anzahl der in jedem Iterationsschritt des Verfahrens zu extrahierenden pseudozufälligen Bits, während *key* einen Pointer darstellt, der auf den zu verwendenden Binärschlüssel zeigt. Dieser Schlüssel muß eine Länge von 24 Bytes aufweisen, wird von der Funktion selbst jedoch nicht auf korrekte Länge hin überprüft!

Von diesen 24 Schlüsselbytes dienen die ersten 16 als direkter Schlüssel des verwendeten IDEA-Verfahrens, wohingegen die verbleibenden 8 Bytes als Initialstartwert der eigentlichen Iteration Anwendung finden.

Ausgabe: Die für das Verfahren benötigten, bibliotheksinternen globalen Variablen werden entsprechend den Funktionsparametern initialisiert.

Rückgabewerte: **RC$NORMAL** Das Verfahren wurde korrekt initialisiert.

RC$VALERR *bits_per_step* war entweder kleiner 1 oder größer 64.

idran_prg_iterate:

Dies ist die eigentliche Iterationsfunktion des auf dem IDEA-Verfahren beruhenden Pseudozufallszahlengenerators. (Sie wird in den meisten Fällen sicherlich über *ofb_iterate* angesprochen.)

```
unsigned char idran_prg_iterate ()
```

Analog zu dem bereits bei *ofb_iterate* Gesagten, setzt auch diese Funktion eine korrekte Initialisierung (durch *idran_init*) voraus!

Rückgabewerte: Jeder Funktionsaufruf liefert ein pseudozufälliges Byte zurück.

mdran_init:

Diese Funktion erlaubt die Initialisierung des auf den MDx-Hashfunktionen beruhenden Generators.

```
int mdran_init (int mode, int bits_per_step, unsigned char *key)
```

Eingabe: Über *mode* wird die gewünschte MD-Funktion ausgewählt – hierbei stehen folgende Konstanten aus *bitstream_lib.h* zur Verfügung:

- *MDRAN$MODE_MD2*,
- *MDRAN$MODE_MD4* bzw.
- *MDRAN$MODE_MD5*.

bits_per_step enthält die Anzahl der bei jedem Iterationsschritt zu extrahierenden Bits, während *key* einen Pointer auf den zu verwendenden Binärschlüssel darstellt.

Rückgabewerte: **RC$NORMAL** Das Verfahren konnte vollständig und korrekt initialisiert werden.

RC$VALERR Der in *bits_per_step* übergebene Wert war entweder kleiner 1 oder größer 128.

RC$WRONG_MODE

mdran_prg_iterate:

Diese Funktion implementiert das Herzstück des auf den MDx-Hashfunktionen basierenden Generators.

```
unsigned char mdran_prg_iterate ()
```

Analog zu dem bereits bei *ofb_iterate* Gesagten, setzt auch diese Funktion eine korrekte Initialisierung durch in diesem Fall *mdran_init* voraus!

Rückgabewerte: Jeder Aufruf dieser Funktion liefert ein neues pseudozufälliges Byte zurück.

des_init:

Initialisiert den DES/D3DES-basierten Pseudozufallszahlengenerator.

```
int des_init (int mode, int enc_dec, int bits_per_step,
              unsigned char *key)
```

Eingabe: Der in *mode* übergebene Wert erlaubt die Unterscheidung zwischen DES bzw. D3DES, was sich nicht zuletzt in der Länge des benötigten Binärschlüssels, der über den Pointer *key* angesprochen wird, niederschlägt. Hierfür sind die beiden Konstanten

- *DES$MODE_DES* und
- *DES$MODE_D3DES*

als Aufrufwerte vorgesehen.

Über *enc_dec* wird selektiert, ob das gewählte Verfahren im Ver- bzw. Entschlüsselungsmodus betrieben werden soll:

- *DES$ENCRYPT* bzw.
- *DES$DECRYPT*.

Ausgabe: Setzen der bibliotheksinternen globalen Variablen des Verfahrens.

Rückgabewerte: **RC$NORMAL** Die Initialisierung konnte korrekt durchgeführt werden.

RC$VALERR Der in *bits_per_step* übergebene Wert ist entweder kleiner 1 oder größer 64.

RC$WRONG_MODE *mode* enthielt einen unzulässigen Wert.

RC$ILLEGAL_KEY Der Schlüssel weist einen Paritätsfehler auf.

des_prg_iterate:

Die vorliegenden Funktion stellt das eigentliche Generatorverfahren dar.

```
unsigned char des_prg_iterate ()
```

Auch diese Routine bedarf einer korrekten Initialisierung der Generator-Umgebung über *des_init*!

Rückgabewerte: Pro Aufruf wird ein neues, pseudozufälliges Byte generiert und zurückgeliefert.

Die folgende Funktion ist Bestandteil der DES-basierten Routinen, aber von teilweise großer Nützlichkeit, so daß sie nicht verschwiegen werden soll.

odd_parity:

Diese Funktion erhält einen Wert des Typs *int*, dessen untere sieben Bits betrachtet und um ein achtes ergänzt werden, so daß der so gebildetet Rückgabewert ungerade Parität aufweist.

```
unsigned char odd_parity (int value)
```

Eingabe: In *value* wird der zu konvertierende Wert übergeben. Dabei ist zu beachten, daß nur die unteren sieben Bits hiervon verwendet werden.

Rückgabewerte: Das zurückgelieferte Byte besteht aus den unteren sieben Bits aus *value*, ergänzt um ein zusätzliches achtes, um eine ungerade Parität zu erzielen.

A.3.2 Das Blum-Blum-Shub–Verfahren

bbs_new_key:

Mit Hilfe dieser Routine kann ein neues Teilschlüsselpaar P und Q entsprechend 4.2.1 erzeugt werden.

```
int bbs_new_key (MP_INT *p, MP_INT *q, long key_len, int mode);
```

Eingabe: Gesamtlänge des zu erzeugenden Schlüssels als Produkt von P und Q über *key_len*. Es gilt: $|P| = \lfloor \frac{key\_len}{2} \rfloor$ und $|Q| = key\_len - |P|$.

Der in *mode* übergebene Wert dient der Angabe, ob die beiden erzeugten Primzahlen lediglich Primzahlen oder spezielle Primzahlen nach Definition 8 sind. Zulässig hierfür sind die beiden in *bitstream_lib.h* definierten Konstanten *BBS$NORMAL* bzw. *BBS$SPECIAL*.

Ausgabe: P und Q über Pointer auf *MP_INT*-Objekte.

Rückgabewerte: **RC$NORMAL** - Die Funktion wurde korrekt beendet.

RC$VALERR - Die angegebene Schlüssellänge ist unzulässig. Sie unterschreitet die in *bitstream_lib.c* durch *BBS$KEY_LEN* definierte Mindestschlüssellänge.

RC$WRONG_MODE - Der in *mode* übergebene Wert ist fehlerhaft.

RC$KEY_NOT_GENERATED - Es konnten keine zwei Primzahlen p und q gefunden werden, die folgende Bedingungen erfüllen:

$$p \neq q \tag{A.3}$$
$$|p| = |q| = |p - q|. \tag{A.4}$$

Die in *bitstream_lib.c* definierte Konstante *BBS$MAX_RETRY* legt die Anzahl der Versuche, solche p und q zu finden, fest.

blum_random_prime:

Die vorliegende Prozedur erzeugt eine *zufällige* Primzahl, die in Abhängigkeit des Wertes von *mode* speziell im Sinne von Definition 8 ist.

```
int blum_random_prime (MP_INT *p, long n, int mode);
```

Eingabe: Länge n der zu erzeugenden Primzahl in Bits. Durch den in *mode* übergebenen Wert kann die Generierung einer speziellen Primzahl erzwungen werden. Für diese Steuerung stehen folgende Konstanten zur Verfügung: *BBS$NORMAL* für eine einfache Primzahl, bzw. *BBS$SPECIAL* für eine spezielle solche.

Ausgabe: Pointer auf ein Objekt des Typs *MP_INT*, das die generierte Zahl enthält.

Rückgabewerte: RC$NORMAL - Die Funktion wurde erfolgreich beendet.

RC$WRONG_MODE - Der in *mode* übergebene Wert ist unzulässig.

blum_prime:

Erzeugt die kleinste Primzahl *result*, die größer oder gleich einem vorgegebenen x ist. Es kann entweder eine normale oder eine spezielle Primzahl generiert werden.

```
int blum_prime (MP_INT *result, MP_INT *x. int mode);
```

Eingabe: Zeiger auf ein *MP_INT*-Objekt, das die untere Schranke x enthält.

Über den in *mode* übergebenen Wert kann die Erzeugung einer normalen oder speziellen Primzahl gesteuert werden. Die hierfür zulässigen Werte wurden bereits im vorangegangenen aufgelistet.

Ausgabe: Ein Pointer auf ein Objekt des Typs *MP_INT*, das die kleinste Primzahl größer gleich x enthält.

bbs_gen_seed:

Diese Prozedur erzeugt mit Hilfe des GNU-MP eigenen Pseudozufallsgenerators einen geeigneten Startwert *seed* für den Blum-Blum-Shub–Generator, der für vorgegebenes m folgende Bedingung erfüllt: $\sqrt{m} < seed < m - 1$.

```
void bbs_gen_seed (MP_INT *m, MP_INT *seed);
```

Eingabe: Pointer auf den verwendeten Modul m.

Ausgabe: Zeiger auf den erzeugten pseudozufälligen Startwert *seed*.

bbs_init:

Mit Hilfe dieser Funktion wird der BBS-Pseudozufallsgenerator initialisiert, d.h. die von ihm benötigten bibliotheksinternen globalen Variablen werden mit entsprechenden Werten vorbelegt, und es wird Speicherplatz für die benötigten *MP_INT*-Objekte allokiert.

```
int bbs_init (MP_INT *modul, int bits_per_step, MP_INT *seed);
```

Eingabe: Zeiger auf den verwendeten Modul des Verfahrens, Anzahl der pro Iterationsschritt zu extrahierenden Bits, sowie der initiale Startwert.

Rückgabewerte: **RC$NORMAL** - Die Initialisierung wurde erfolgreich abgeschlossen.

RC$VALERR - Der mit *bits_per_step* übergebene Wert ist kleiner Null.

bbs_prg_iterate:

Diese Funktion implementiert das eigentliche Kernstück des Blum-Blum-Shub–Pseudozufallsgenerators – die Iterationsfunktion. Mit ihrer Hilfe kann nach erfolgter Initialisierung (siehe oben) mit jedem Aufruf ein neues pseudozufälliges Byte zur Verschlüsselung des Klartextes gewonnen werden.

Der Aufruf dieser Funktion erfordert eine vorangegangene korrekte Initialisierung des Verfahrens! In jedem anderen Fall wird der Programmlauf unter Ausgabe einer entsprechenden Fehlermeldung abgebrochen!

```
unsigned char bbs_prg_iterate ();
```

Rückgabewerte: Ein pseudozufälliges, vorzeichenloses Byte.

bbs_next_seed:

Nach erfolgter Verschlüsselung des Klartextes liefert ein abschließender Aufruf dieser Funktion den nächsten Iterationswert zurück, der für die Entschlüsselung des Chiffretextes durch den Besitzer der Primfaktorzerlegung des verwendeten Moduls unerläßlich ist.

```
int bbs_next_seed (MP_INT *seed);
```

Ausgabe: Der nächste Iterationswert des BBS-Generators.

Rückgabewerte: **RC$NORMAL** - Der Wert konnte erfolgreich bestimmt und zurückgeliefert werden.

RC$NOT_INITIALIZED - Das Verfahren wurde noch nicht durch *bbs_init* initialisiert, folglich konnte kein nächster Iterationswert bestimmt werden.

bbs_backstep:

Hierin liegt das Kernstück der Entschlüsselung nach Blum-Blum-Shub verborgen. Die Prozedur implementiert den Algorithmus nach Abschnitt 4.2.3.

```
int bbs_backstep (long steps, MP_INT *p, MP_INT *q, MP_INT *seed);
```

Eingabe: Eine natürliche Zahl, die die Anzahl der zurückzurechnenden Schritte enthält, die Zerlegung des Moduls in Form zweier Zeiger auf Objekte des Typs *MP_INT*, die P resp. Q enthalten, sowie den im Zuge der Verschlüsselung der betreffenden Daten durch Aufruf von *bbs_next_seed* erhaltenen nächsten Iterationswert.

Ausgabe: In der durch *seed* adressierten Variablen wird am Ende des Algorithmus der errechnete Wert von x_0 abgelegt.

Rückgabewerte: **RC$NORMAL** - Die Funktion wurde korrekt abgeschlossen.

RC$VALERR - Der in *steps* übergebene Wert ist kleiner 1.

bbs_cleanup:

Nach vollzogener Ver- bzw. Entschlüsselung von Daten mit Hilfe des Blum-Blum-Shub–Generators dient diese Routine einerseits zum Zurücksetzen der betreffenden globalen Variablen, sowie andererseits zur Freigabe des für die Langzahlvariablen allokierten Speichers. Nach einem solchen Aufruf muß das Verfahren gegebenenfalls durch *bbs_init* neu initialisiert werden.

```
int bbs_cleanup ();
```

Rückgabewerte: **RC$NORMAL** - Der allokierte Speicherplatz wurde korrekt freigegeben.

RC$NOT_INITIALIZED - Das System war nicht initialisiert!

A.3.3 Der Micali-Schnorr–Generator

mschnorr_new_mod:

Diese Funktion dient der Generierung eines neuen Moduls für das Verfahren nach Micali und Schnorr. In ihrer Funktionalität entspricht sie der zuvor besprochenen Funktion *bbs_new_key*, mit dem Unterschied, daß die Erzeugung spezieller Primzahlen nicht vorgesehen ist, bei Bedarf jedoch sehr leicht nachrüstbar

erscheint, da *mschnorr_new_mod* auf denselben BBS-Routinen wie *bbs_new_key* basiert, denen im Bedarfsfall lediglich ein entsprechender *mode*-Wert übergeben werden müsste.

```
int mschnorr_new_mod (MP_INT *p, MP_INT *q, long mod_len)
```

Eingabe: Der Wert der Variablen *mod_len* legt die Länge des zu erzeugenden neuen Moduls fest. Da dieser in Form seiner beiden Primfaktoren zurückgeliefert wird (die für eine sich eventuell anschließende Bestimmung eines neuen Exponenten benötigt werden), enthalten p und q im Anschluß an den Aufruf dieser Funktion Primzahlen, deren Länge jeweils der Hälfte von *mod_len* entspricht. Diese Längen verhalten sich wie bereits bei *bbs_new_key* beschrieben.

Ausgabe: In den über die Pointer p, sowie q referenzierten Objekten des Typs *MP_INT* werden die beiden erzeugten Primfaktoren des neuen Moduls zurückgeliefert.

Rückgabewerte: **RC$NORMAL** - Die Funktion wurde korrekt und erfolgreich beendet.

RC$VALERR - Der in *mod_len* übergebene Wert unterschritt die in *bitstream_lib.c* durch *MSCHNORR$MOD_LEN* definierte Mindestmodullänge.

RC$KEY_NOT_GENERATED - Es konnten keine zwei Primzahlen p und q gefunden werden, die folgende Bedingungen erfüllen:

$$p \neq p \tag{A.5}$$

$$|p| = |q| = |p - q|. \tag{A.6}$$

Die maximale Anzahl von Versuchen, die unternommen werden, um ein solches p und q zu finden, wird durch die in *bitstream_lib.c* definierte Konstante *MSCHNORR$MAX_RETRY* festgelegt.

mschnorr_gen_d:

Mit Hilfe dieser Funktion kann ein neuer Exponent für einen gegebenen Modul erzeugt werden.

```
long mschnorr_gen_d (MP_INT *p, MP_INT *q)
```

Eingabe: Die durch p bzw. q adressierten *MP_INT*-Variablen enthalten die Primfaktoren des betreffenden Moduls.

Rückgabewerte: Es wird der neu erzeugte Exponent als Funktionswert zurückgeliefert. Konnte kein solcher gefunden werden, ist das Ergebnis 0.

mschnorr_gen_seed:

Zur Erzeugung eines neuen Startwertes für das Verfahren nach Micali und Schnorr dient diese Funktion:

```
int mschnorr_gen_seed (MP_INT *seed, MP_INT *modul, long d)
```

Eingabe: Über den Pointer *modul* wird der Modul des Verfahrens, für das ein neuer Startwert erzeugt werden soll, übergeben; der Parameter *d* enthält den zu verwendenden Exponenten.

Ausgabe: Der generierte Startwert wird über *seed* an das rufende Programm zurückgeliefert.

Rückgabewerte: **RC$NORMAL** - Die Funktion konnte korrekt beendet werden, es wurde ein neuer Startwert erzeugt.

RC$VALERR - Der übergebene Modul ist unzulässig (kleiner als die in *bitstream_lib.c* definierte Konstante *MSCHNORR$MOD_LEN*).

mschnorr_init:

Die vorliegende Funktion gestattet die Initialisierung des Verfahrens nach Micali-Schnorr.

```
int mschnorr_init (MP_INT *modul, long d, MP_INT *seed)
```

Eingabe: Die Pointer *modul* und *seed* zeigen auf Objekte des Typs *MP_INT*, die den zu verwendenden Modul des Verfahrens, sowie den Startwert enthalten. *d* enthält den hierfür gewählten Exponenten.

Ausgabe: Die globalen, bibliotheksinternen Variablen werden entsprechend den übergebenen Werten gesetzt.

Rückgabewerte: **RC$NORMAL** - Das Verfahren konnte einwandfrei initialisiert werden.

RC$VALERR - Der übergebene Exponent ist kleiner als 3.

mschnorr_prg_iterate:

Diese Funktion implementiert das eigentliche Verfahren zur Erzeugung pseudozufälliger Bytes.

```
unsigned char mschnorr_prg_iterate ()
```

Zu beachten ist, daß diese Funktion eine korrekt initialisierte Umgebung zwingend voraussetzt! Ist dies nicht gewährleistet, wird mit einer entsprechenden Fehlermeldung abgebrochen!

Rückgabewerte: Jeder Funktionsaufruf liefert ein neues pseudozufälliges Byte zurück.

mschnorr_cleanup:

Mit Hilfe der vorliegenden Funktion können die durch *mschnorr_init* gesetzten bibliotheksglobalen Variablen freigegeben werden.

```
int mschnorr_cleanup ()
```

Rückgabewerte: **RC$NORMAL** - Der allokierte Speicherplatz wurde freigegeben.

RC$NOT_INITIALIZED - Das Verfahren war nicht initialisiert, es konnte keine Deallokation stattfinden.

A.3.4 Das Verfahren nach Impagliazzo-Naor

inaor_gen_par_set:

Diese Funktion erzeugt einen neuen Parametersatz mit Hilfe des GNU-MP eigenen Pseudozufallsgenerators. Dieser wird in einem globalen Array aus Objekten des Typs *MP_INT* abgelegt. Hierbei muß angemerkt werden, daß dieses Array eine Obergrenze besitzt, die durch die in *bitstream_lib.h* definierte Konstante *INAOR$MAXPAR* festgelegt wird. Dies wird einerseits dadurch begründet, daß die in der GNU-MP Bibliothek vorhandene Routine *mpz_array_init* zum einen nicht in der Lage ist, Arrays anzulegen, deren Objekte vergrößert werden können; andererseits können auf diese Art und Weise erzeugte Felder nicht wieder freigegeben werden, was bei dem teilweise enormen Speicherplatzbedarf des Impagliazzo-Naor–Verfahrens einen wirklichen Nachteil darstellt, der auch zu vorliegender Implementation führte.

```
int inaor_gen_par_set (int number, int len);
```

Eingabe: Anzahl (*number*) der zu erzeugenden Parameter, sowie deren Länge in Bits (*len*).

Ausgabe: Der erzeugte Parametersatz wird in einem globalen, bibliothekseigenen Feld gespeichert.

Rückgabewerte: **RC$NORMAL** - Es wurde ein neuer Parametersatz erzeugt.

RC$VALERR - $len < 1$ oder $len > number * INAOR\_C$.

RC$PAROVL - $number < 1$ bzw. $number > INAOR\$MAXPAR$.

RC$NOT_CLEANED_UP - Das System wurde bereits in irgendeiner Form initialisiert. Vor dem nächsten Aufruf der Routine ist *inaor_cleanup* zu starten.

inaor_set_par_set:

Für die vorliegende Funktion gilt generell das bereits zuvor Gesagte. Im Unterschied zu *inaor_gen_par_set* erlaubt sie das Belegen des globalen Parametersatzes mit vorgegebenen Werten.

```
int inaor_set_par_set (int number, int len, MP_INT *parameter []);
```

Eingabe: Zu *number* und *len* vergleiche obige Beschreibung, *parameter* ist ein Zeiger auf ein Array aus Objekten des Typs *MP_INT* des rufenden Programmteils, dessen Werte kopiert werden sollen.

Ausgabe: Der übergebene Parametersatz wird in das globale Array kopiert.

Rückgabewerte: Diese entsprechen exakt den Rückgabewerten von *inaor_gen_par_set*, siehe dort.

inaor_load_par_set:

Diese Routine erlaubt es, einen Parametersatz von einem File zu laden. Ansonsten gilt das für die beiden vorangegangenen Funktionen bereits Gesagte.

```
int inaor_load_par_set (char *file_name);
```

Eingabe: Ein Pointer auf den den Namen des zu lesenden Files enthaltenden String.

Ausgabe: Der geladene Parametersatz wird auf das globale Parameterfeld kopiert.

Rückgabewerte: Zusätzlich zu den bereits bei *inaor_gen_par_set* genannten kommen die folgenden hinzu:

RC$FORERR - Das spezifizierte File konnte nicht für lesenden Zugriff geöffnet werden.

RC$EOFENC - Während des Kopierens der Parameterwerte wurde unerwartet das Fileende erreicht. Bereits allokierte Speicherbereiche für das globale Parameterarray werden vor Verlassen der Funktion freigegeben, das File wird korrekt geschlossen.

inaor_save_par_set:

Diese Funktion vollzieht die umgekehrte Aufgabe der im vorangegangenen beschriebenen Routine. Sie speichert einen bereits global bekannten Parametersatz in einem File ab.

```
int inaor_save_par_set (char *file_name);
```

Eingabe: Ein Pointer auf den String, in dem der Name des Ausgabefiles abgelegt ist.

Ausgabe: Der Inhalt des globalen Parameterfeldes wird auf das spezifizierte File geschrieben.

Rückgabewerte: **RC$NORMAL** - Der Parametersatz wurde vollständig auf das Ausgabefile kopiert.

RC$FOWERR - Das File konnte nicht für schreibenden Zugriff geöffnet werden.

RC$NOT_INITIALIZED - Es existiert kein Parametersatz, der abgespeichert werden könnte, da das Verfahren noch nicht initialisiert wurde!

inaor_characteristics:

Diese Funktion ermöglicht es, die Anzahl, sowie die Länge der globalen Parameterwerte zu ermitteln.

```
int inaor_characteristics (int *number, int *len);
```

Ausgabe: In den durch die Pointer *number* resp. *len* referenzierten Integer-Objekten werden Länge und Anzahl der Parameterwerte des Verfahrens abgelegt.

Rückgabewerte: RC$NORMAL - Die Funktion wurde korrekt abgearbeitet.

RC$NOT_INITIALIZED - Das Verfahren ist nicht initialisiert – es existiert kein globaler Parametersatz, dessen Charakteristika ermittelt werden könnten.

inaor_get_par_set:

Wurden mit Hilfe der Funktion *inaor_characteristics* sowohl Anzahl als auch Länge des globalen Parametersatzes bestimmt, so kann dieser unter Zuhilfenahme der hier beschriebenen Routine auf ein Feld des rufenden Programms kopiert werden.

```
int inaor_get_par_set (int number, MP_INT *parameter []);
```

Ausgabe: In dem durch den Zeiger *parameter* adressierten Feld aus Objekten des Typs *MP_INT* wird eine Kopie des globalen Parametersatzes abgelegt. (Zu bemerken ist hierbei lediglich, daß die einzelnen Feldelemente innerhalb der Routine korrekt allokiert werden.)

Rückgabewerte: RC$NORMAL - Die Funktion wurde korrekt abgearbeitet.

RC$NOT_INITIALIZED - Das Verfahren ist nicht initialisiert – es existiert kein globaler Parametersatz, der kopiert werden könnte.

inaor_gen_seed:

Die vorliegende Funktion erzeugt einen pseudozufälligen Startwert der Länge *len* für das Impagliazzo-Naor–Verfahren

```
int inaor_gen_seed (int len, MP_INT *seed);
```

Eingabe: Länge des zu erzeugenden Wertes in Bits (*len*).

Ausgabe: Ein Zeiger auf ein *MP_INT*-Objekt, das den erzeugten Wert enhält.

Rückgabewerte: **RC$NORMAL** - Der neue Startwert wurde fehlerfrei erzeugt.

RC$VALERR - Die angegebene Seedlänge ist kleiner 1.

inaor_set_seed:

Die im folgenden beschriebene Funktion kopiert einen vorgegebenen Startwert in die entsprechende bibliothekseigene globale Iterationsvariable.

```
int inaor_set_seed (MP_INT *seed);
```

Eingabe: Pointer auf das den gewünschten Startwert enthaltende *MP_INT*-Objekt.

Rückgabewerte: **RC$NORMAL** - Der globale Startwert wurde erfolgreich gesetzt.

RC$NOT_INITIALIZED - Das Verfahren besitzt noch keinen globalen Parametersatz – es wurde kein Startwert geschrieben.

inaor_get_seed:

Die vorliegende Funktion liefert den aktuellen Wert des globalen Seeds des Impagliazzo-Naor–Pseudozufallsgenerators zurück.

```
int inaor_get_seed (MP_INT *seed);
```

Ausgabe: Der Wert des globalen Seeds wird in dem über *seed* adressierten *MP_INT*-Objekt zurückgeliefert.

Rückgabewerte: **RC$NORMAL** - Die Funktion wurde korrekt beendet.

RC$NOT_INITIALIZED - Das Verfahren ist nicht vollständig initialisiert - es existiert kein globaler Startwert, der kopiert werden könnte.

inaor_prg_iterate:

Hierin findet sich die eigentliche Iterationsfunktion des Impagliazzo-Naor–Pseudozufallsgenerators.

```
unsigned char inaor_prg_iterate ();
```

Rückgabewert: Jeder Aufruf dieser Funktion liefert ein neues vorzeichenloses Pseudozufallsbyte zur Klartextverschlüsselung zurück.

Zu beachten ist, daß diese Routine eine korrekte Initialisierung des Verfahrens voraussetzt! Im anderen Fall wird eine Fehlermeldung erzeugt und der Programmlauf unterbrochen.

inaor_cleanup:

Wie bei den beiden vorangegangenen Verfahren existiert auch für diese Routinengruppe eine Prozedur zur Freigabe des zur Laufzeit allokierten Speicherplatzes, was insbesondere bei diesem Verfahren ein nicht unwesentlicher Punkt ist, da die verwendeten Parametersätze eine in den meisten Fällen nicht zu unterschätzende Größe aufweisen.

```
int inaor_cleanup ();
```

Rückgabewerte: RC$NORMAL - Der nicht mehr benötigte Speicherplatz der globalen Variablen und Felder wurde freigegeben.

RC$NOT_INITIALIZED - Das Impagliazzo-Naor–Verfahren ist nicht initialisiert.

Anhang B

Das Programm bcrypt

Das vorliegende Kapitel beschreibt den Leistungsumfang, sowie die Benutzerschnittstelle des Programms *bcrypt*. Es dient als Frontend für die im Rahmen dieser Diplomarbeit entwickelten Bibliotheksroutinen zur Bitstromverschlüsselung und ist Teil dieser Arbeit. Sein Quellcode ist bewußt klar gehalten und kann als Beispiel für die Aufrufmechanismen der einzelnen Funktionen Anwendung finden. Dennoch besitzt es einen vergleichsweise großen Leistungsumfang und kann durchaus gewinnbringend in kryptographischen Anwendungen eingesetzt werden.

Nicht verschwiegen werden sollte, daß *bcrypt* durchaus erweiterungs- und auch verbesserungsfähig ist; so ist beispielsweise die Passwortverarbeitung, auf die noch ausführlich eingegangen wird, nur rudimentär implementiert und bietet keine Möglichkeit, ein Passwort ohne Echo auf *stdout* einzugeben, da der Hauptzweck des Programms in seiner Verwendung als Demonstrationsobjekt zu sehen ist. All diese Schwachpunkte sollten jedoch mit nur mäßigem Aufwand behebbar sein, so daß sie letztendlich der Brauchbarkeit des Programms keinen Abbruch tun.

bcrypt basiert ausschließlich auf den bereits in Kapitel 4 vorgestellten und in Anhang A aufgelisteten Funktion der Bibliothek *bitstream_lib.c*. Es ermöglicht die Verschlüsselung von Eingabedatenströmen mit Hilfe der dort implementierten Pseudozufallsgeneratoren unter einer einheitlichen Bedieneroberfläche. Wie die Bibliothek selbst, wurde es in der Programmiersprache *C* implementiert und basiert auf der GNU-MP Langzahl-Bibliothek.

Das Verhalten des Programms *bcrypt* wird mit Hilfe geeigneter Kommandozeilenparameter gesteuert, deren Beschreibung Ziel dieses Abschnittes ist.

Die sicherlich einfachste Form eines *bcrypt* Aufrufes ist die ohne jegliche weitere Parameter, was die Ausgabe eines einfachen Hilfstextes, der alle verfügbaren Parameter tabellarisch auflistet, zur Folge hat. (Ein ausführlicherer Hilfstext ist unter Verwendung des Parameters *-h* verfügbar.)

bcrypt stellt insgesamt neun verschiedene Verschlüsselungsverfahren, die sämtlich auf der Anwendung von Pseudozufallsgeneratoren beruhen. Diese Verfahren lassen sich in folgende 4 Gruppen einteilen:

OFB-basierte Generatoren - Diese Gruppe besteht im Unterschied zu den drei folgenden aus einer Reihe von Zufallsgeneratoren, die auf folgenden

bekannten Blockchiffre- bzw. Hashverfahren im OFB-Modus basieren:

- DES
- D3DES
- IDEA
- MD2
- MD4
- MD5

Da diesen Generatoren allen eine gemeinsame algorithmische Struktur zugrundeliegt, wurde sie in einer Gruppe zusammengefaßt, so daß sie sich von einer einzigen Benutzerschnittstelle aus steuern lassen.

Der Blum-Blum-Shub Generator - Dieser in Abschnitt 3.1 eingehend behandelte Pseudozufallsgenerator stellt das komplexeste Verfahren der gesamten Bibliothek und somit auch von *bcrypt* dar, da hiermit ein vollständiges public-key Verschlüsselungsverfahren implementiert wurde. Infolgedessen ist auch die Benutzerschnittstelle dieses Systems die mit Abstand komplizierteste.

Der Generator nach Micali Schnorr - Dieses Verfahren besitzt von seiner Steuerung her große Ähnlichkeit mit dem nachfolgenden, letzten Pseudozufallsgenerator, wurde jedoch aufgrund seines unterschiedlichen Parametersatzes nicht mit diesem kombiniert.

Das Verfahren nach Impagliazzo-Naor - Der Generator nach Impagliazzo-Naor stellt das letzte und zugleich auch aufwendigste Verfahren der Bibliothek dar und ist aufgrund seines bereits erwähnten Laufzeitverhaltens für eine Vielzahl praktischer Anwendungen sicherlich nicht hinreichend effizient.

Im folgenden soll die Bedienung der einzelnen Teilfunktionen von *bcrypt* näher betrachtet werden:

B.1 Die OFB-basierten Verfahren

Wie bereits angemerkt, sind in dieser Gruppe sechs verschiedene Pseudozufallsgeneratoren zusammengefaßt, wie dies auch direkt in den zugrundeliegenden Bibliotheksfunktionen der Fall ist. Der erste Übergabeparameter an *bcrypt* wählt in allen Fällen das gewünschte Verfahren aus, auf das sich im folgenden auch alle weiteren Angaben der Kommandozeile beziehen. Die möglichen Qualifizierer listet nachfolgende Tabelle auf:

Zugrundeliegendes Verfahren	Argument
DES	-des
D3DES	-d3des
IDEA	-idea
MD2	-md2
MD4	-md4
MD5	-md5

Nach Anwahl des gewünschten Pseudozufallsgenerators können folgende Funktionen vollzogen werden[1]:

- Erzeugung eines neuen Schlüssels
- Ver- bzw. Entschlüsselung von Daten

Zunächst sei das zur Generierung eines neuen Schlüssels notwendige Procedere beschrieben:

B.1.1 Schlüsselerzeugung

Obwohl alle schlüsselabhängigen Funktionen aus *bitstream_lib.c* einen Schlüssel in binärer Form erfordern, tritt dieser bei der Bedienung von *bcrypt* nach außen nie in Erscheinung, sondern wird stets durch seine Hexadezimaldarstellung repräsentiert.

Die Erzeugung eines Schlüssels wird generell durch den Schalter *-kg* gestartet, wie folgendes Beispiel demonstriert, das einen neuen Schlüssel für den DES-basierten Pseudozufallsgenerator erzeugt:

```
bcrypt -des -kg
```

Der solchermaßen erzeugte Schlüssel wird auf die Standardausgabe *stdout* geschrieben. Soll er hingegen direkt in ein File abgelegt werden, ist folgende Erweiterung des obigen Aufrufs verwendbar, die den erzeugten Schlüssel in ein File namens *key_file.dat* kopiert:

```
bcrypt -des -kg des_key.dat
```

Der Aufruf zur Generierung eines neuen Schlüssels hat für die OFB-basierten Verfahren folgende allgemeine Gestalt:

$$bcrypt \left\{ \begin{array}{c} -des \\ -d3des \\ -idea \\ -md2 \\ -md4 \\ -md5 \end{array} \right\} - kg\ [file\_name]$$

Nachdem nun ein gültiger Schlüssel für das gewünschte Verfahren erzeugt wurde, kann mit seiner Hilfe die Ver- bzw. Entschlüsselung von Nutzdaten vollzogen werden, der sich der nächste Abschnitt widmet.

[1] Außer den beiden genannten Grundfunktionen kann durch Angabe des Qualifizierers *-h* nach Wahl eines Verfahrens ein ausführlicher Hilfetext zu seiner Bedienung angefordert werden.

B.1.2 Ver- bzw. Entschlüsselung

Ausgehend von einem Startwert, der beispielsweise analog zu obigem Beispiel erzeugt werden kann, wird mit Hilfe des selektierten Generators eine Folge pseudozufälliger Bytes erzeugt, die mit den Bytes des Eingabedatenstromes Exklusiv-Oder-verknüpft werden. Die aus dieser Operation resultierenden Bytes stellen den zum Klartext korrespondierenden Chiffretext dar, der durch Verknüpfung mit denselben Ausgabebytes des Generators wieder in den Klartext konvertiert werden kann, so daß ein und derselbe Schlüssel sowohl für die Ent- als auch für die Verschlüsselung von Daten verwendet wird[2].

Der allgemeine Aufruf zur Verschlüsselung von Daten gestaltet sich wie folgt:

$$bcrypt \left\{ \begin{array}{c} -des \\ -d3des \\ -idea \\ -md2 \\ -md4 \\ -md5 \end{array} \right\} \left\{ \begin{array}{llll} -k & [\{hex\_key\} & [-b\ bits\_per\_step] & [infile\ [outfile]]] \\ -kf & [\{key\_file\} & [-b\ bits\_per\_step] & [infile\ [outfile]]] \\ -kp & [\{password\} & [-b\ bits\_per\_step] & [infile\ [outfile]]] \end{array} \right\}$$

Allen drei Varianten sind die letzten Argumente gemeinsam. Der optionale Qualifizierer $-b$ mit dem ihm folgenden Argument *bits_per_step* ermöglicht die Spezifizierung der Anzahl von Bits, die in jedem Iterationsschritt des Pseudozufallsgenerators zu extrahieren sind, um die Folge der Ausgabebytes zu generieren. Der Default-Wert für diese Anzahl liegt bei 1. Es ist stets so, daß sich ein großer Wert für *bits_per_step* zwar günstig auf das Laufzeitverhalten des jeweiligen Verfahrens auswirkt, ein kleiner jedoch aus kryptographischer Sicht mitunter vorzuziehen ist, da weniger Information über das verwendete Verfahren, sowie seinen internen Zustand preisgegeben wird.

Für *bits_per_step* gilt als untere Schranke stets 1, sein maximaler Wert ist von der Art des gewählten Verfahrens abhängig, wie nachstehende Tabelle verdeutlicht:

Zugrundeliegendes Verfahren	Obere Schranker für *bits_per_step*
DES	64
D3DES	64
IDEA	64
MD2	128
MD4	128
MD5	128

Die beiden letzten Parameter erlauben die Angabe eines Ein- bzw. Ausgabefiles. Wird hiervon kein Gebrauch gemacht, liest das Programm von der Standardeingabe *stdin* und schreibt auf die Standardausgabe *stdout*.

Den Qualifizierern $-k$, $-kf$, sowie $-kp$ kommt nun folgende Bedeutung zu:

-k: Dieser Schalter erwartet den zu verwendenen Schlüssel in hexadezimaler Form als nächstes Argument der Kommandozeile. Fehlt dieser, wird er explizit von *bcrypt* erfragt.

[2]Bis auf den Blum-Blum-Shub Generator gilt dies für alle der implementierten Verfahren.

-kf: Im Gegensatz zu $-k$ wird bei dieser Variante der Schlüssel (ebenfalls in seiner hexadezimalen Repräsentation) aus einem File gelesen, dessen Name als nächstes Kommandozeilenargument erwartet wird. Auch in diesem Fall wird ein nicht spezifizierter Filename vom Benutzer erfragt.

-kp: Diese Option erwartet keinen Schlüssel im bisherigen Sinne. Vielmehr wird aus einem Passwort (das bei Eingabe auf stdout geschrieben wird, da lediglich normale Standardfunktionen hierbei verwendet wurden – ein Punkt, der sicherlich verbesserungswürdig ist) ein Schlüssel generiert, der dann für die eigentliche Verschlüsselungsfunktion verwendet wird[3]. Analog zu den beiden oben stehenden Varianten wird auch hier ein fehlendes Folgeargument (das Passwort in diesem Fall) interaktiv von *bcrypt* erfragt.

Diese Auflistung schließt die Behandlung der ersten sechs implementierten Verfahren ab, so daß nun mit der Beschreibung des Verfahrens von Blum-Blum-Shub fortgefahren werden kann.

B.2 Der BBS-Pseudozufallsgenerator

Das dem folgenden zugrundeliegende Verfahren besitzt aufgrund seiner Eigenschaften als public-key System die komplexeste Benutzerschnittstelle. Zunächst sei die Erzeugung eines neuen Schlüssels betrachtet, der sich nach Abschnitt 4.2.1 aus zwei Primzahlen P und Q zusammensetzt. Das Produkt dieser beiden Zahlen, im folgenden N genannt, stellt den öffentlichen Schlüssel, P und Q selbst den privaten, geheimen Schlüssel dar.

B.2.1 Schlüsselerzeugung

Die Erzeugung eines wie eben beschriebenen Schlüsselpaares geschieht auch bei dem vorliegenden Verfahren unter Verwendung des Schalters $-kg$, dessen allgemeine Syntax folgende Gestalt aufweist:

$$bcrypt\ -bbs\ -kg\ [\{key\_len\}\ [\{pers\_name\}]]$$

Das auf $-kg$ folgende Argument *key_len* dient der Spezifizierung der gewünschten Länge des neuen öffentlichen Schlüssels N in Bits. Ein neu generiertes Schlüsselpaar wird in zwei Files festgelegten Namens abgelegt. Der öffentliche Schlüssel N wird in eine Datei *public_key.bbs* kopiert, der private Schlüssel, der die Primfaktorzerlegung von N in Gestalt von P und Q enthält, wird auf *private_key.bbs* geschrieben.

Mit jedem öffentlichen Schlüssel ist ein sogenannter *personal name* verbunden, der als fester Bestandteil ebenfalls in *public_key.dat* abgelegt wird. Dieser wird durch das letzte Argument der obigen Kommandozeile festgelegt. Fehlt eine der beiden letzten Angaben, wird sie von *bcrypt* automatisch vom Benutzer erfragt.

[3]Dieser Form der Schlüsselerzeugung liegt die in *bitstream_lib.c* enthaltene Funktion *ofb_pwd_to_key* zugrunde.

Das File *private_key.dat* besteht aus lediglich zwei Zeilen, die die Werte von P bzw. Q in hexadezimaler Form enthalten; auch *public_key.dat* besteht aus lediglich zwei Records, von denen der erste den mit dem betreffenden öffentlichen Schlüssel N verbundenen *personal name* enthält; der zweite besteht aus der Hexadezimalrepräsentation des eigentlichen Schlüssels.

Um nun Daten unter Zuhilfenahme eines öffentlichen Schlüssels zu chiffrieren, muß dieser zunächst in einer Datei *bbs_keys.dat* abgelegt worden sein. Hierfür steht der Qualifizierer $-i$ zur Verfügung, der einen öffentlichen Schlüssel aus einem File liest und ihn in *bbs_keys.dat* schreibt[4]. Jeder in dieser Datei vorhandene öffentliche Schlüssel besteht aus zwei Records, die wiederum seinen *personal name*, sowie das zugehörige N enthalten[5]. Der Aufruf zum Eintrag eines neuen öffentlichen Schlüssels gestaltet sich wie folgt:

$$bcrypt\ -bbs\ -i\ [\{public\_key\_file\}]$$

Ist die Datei *bbs_keys.dat* zum Zeitpunkt dieses Aufrufs noch nicht vorhanden, wird sie (nach Bestätigung durch den Benutzer) angelegt, um danach den gewünschten Schlüssel aufzunehmen.

Soll beispielsweise ein in der Datei *public.dat* enthaltener öffentlicher Schlüssel in das eigene System eingetragen werden, ist wie folgt vorzugehen:

```
bcrypt -bbs -i public.dat
```

Die Option $-l$ gestattet die Auflistung aller in *bbs_keys.dat* enthaltenen und somit frei verfügbaren öffentlichen Schlüssel:

$$bcrypt\ -bbs\ -l$$

Das Entfernen eines bestimmten öffentlichen Schlüssels aus *bbs_keys.dat* wird durch Angabe des Schalters $-del$ ermöglicht:

$$bcrypt\ -bbs\ -del\ [\{pers\_name\}]$$

Ein fehlendes letztes Argument (der *personal name* des zu löschenden Schlüssels) wird interaktiv vom Benutzer erfragt.

B.2.2 Verschlüsselung

Unter der Voraussetzung, daß *bbs_keys.dat* einerseits existiert und andererseits den zur gewünschten Verschlüsselung notwendigen öffentlichen Schlüssel beinhaltet, kann mit der eigentlichen Verarbeitung von Eingabedaten begonnen werden. Der *bcrypt*-Aufruf zur Verschlüsselung von Daten mit Hilfe des Verfahrens nach Blum-Blum-Shub hat folgende allgemeine Gestalt:

$$bcrpyt\ -e\ [-b\ bits\_per\_step]\ [\{pers\_name\}\ [\{infile\}\ [\{outfile\}]]]$$

[4]Auch diese Funktion ist nur sehr rudimentär implementiert – die kopierende Datei darf keinerlei weitere Einträge aufweisen. Ebenfalls wird keinerlei Plausibilitätsprüfung des einzutragenden Schlüssels vorgenommen, was nicht zuletzt aus der Absicht resultiert, mit *bcrypt* in der Hauptsache ein Demonstrationsprogramm für die Verwendung der *bitstream_lib.c* eigenen Funktionen zu erstellen.

[5]Das Dateiende wird durch einen speziellen Eintrag markiert – siehe *bcrypt.c*.

Dem optionalen Qualifizierer $-b$ und dem ihm folgenden Argument kommt dieselbe Bedeutung wie bei den bereits behandelten OFB-basierten Pseudozufallsgeneratoren zu. Auch hier liegt seine untere Schranke bei eins[6], während seine prinzipielle obere Schranke durch die Länge des verwendeten Moduls N in Bits gegeben ist.

Das nächste Argument repräsentiert den *personal name* des zu verwendenden öffentlichen Schlüssels aus *bbs_keys.dat*, gefolgt von den Namen der betreffenden Ein- bzw. Ausgabefiles. Alle diese Parameter sind nicht optional und werden bei ihrem Fehlen erfragt.

Das im Verlauf einer Verschlüsselungsoperation erzeugte Ausgabefile hat folgende Gestalt:

```
<Laenge des Eingabefiles> <bits_per_step>
   ...Chiffretext...
<letzter Seed>
```

B.2.3 Entschlüsselung

Zur Entschlüsselung eines gegebenen Chiffretextfiles muß der entsprechende geheime Schlüssel, d.h. die Primfaktorzerlegung des bei der Verschlüsselung zur Anwendung gekommenen Moduls N, bekannt sein. Dieser private Schlüssel wird in einem File, das aus genau zwei Einträgen besteht, nämlich der Hexadezimaldarstellung der beiden Primfaktoren P und Q, vorausgesetzt. Zur Entschlüsselung der betreffenden Daten werden aus dem Chiffretextfile zunächst die Länge der zugrundeliegenden Klartextdaten in Bytes, sowie der bei der Verschlüsselungsoperation verwendete Wert für *bits_per_step* gelesen. Aus diesen Daten sowie dem am Dateiende befindlichen Iterationswert kann gemäß Abschnitt 4.2.3 der ursprüngliche Startwert des Verfahrens rekonstruiert werden, um mit seiner Hilfe die Entschlüsselung durchführen zu können.

Der Aufruf zur Entschlüsselung eines Chiffretextfiles hat folgendes Aussehen:

$$bcrypt\ -bbs\ -d\ [\{private\_key\_file\}\ [\{infile\}\ [outfile]]]$$

Das auf den Qualifizierer $-d$ folgende Argument *private_key_file* spezifiziert den Namen der den zu verwendenden privaten Schlüssel enthaltenden Datei. Fehlt einer der Parameter *private_key_file* bzw. infile, wird er interaktiv vom Benutzer erfragt. Die Angabe eines Ausgabefiles ist optional; fehlt sie, wird der entschlüsselte Klartext auf die Standardausgabe geschrieben.

Nachdem nun hiermit alle das Verfahren nach Blum-Blum-Shub betreffenden Aufrufvarianten des Demonstrationsprogramms *bcrpyt* behandelt wurden, kann mit der Beschreibung des vorletzten Chiffriersystems fortgefahren werden:

B.3 Das Verfahren nach Micali-Schnorr

Dieser Pseudozufallszahlengenerator basiert ebenfalls auf der Potenzierung eines gegebenen Startwertes zu einem gewissen Modul, wie dies bereits dem vor-

[6]Für diesen Wert, sowie für 2 zeigt [1] die kryptographische Sicherheit des Verfahrens, so daß nur nach genauer Prüfung der Gegebenheiten ein größerer Wert spezifiziert werden sollte.

angehend beschriebenen Verfahren zu eigen war. Im Unterschied hierzu wird jedoch nicht von quadratischen Polynomen sondern von RSA-Polynomen der allgemeinen Gestalt

$$f(x) = x^d \bmod N \tag{B.1}$$

ausgegangen, wobei N wie bereits zuvor das Produkt zweier Primzahlen P und Q darstellt. d muß folgende Bedingung erfüllen:

$$\mathrm{ggT}(d, \varphi(N)) = 1. \tag{B.2}$$

Die Erzeugung eines Parametersatzes für das vorliegende Verfahren beruht nun auf der Generierung eines Moduls N sowie eines zugehörigen Exponenten d.

Durch Angabe des Qualifizierers $-ms$ als erstes Argument eines *bcrypt*-Aufrufes wird für den betreffenden Programmlauf das Verfahren nach Micali-Schnorr selektiert.

B.3.1 Erzeugung eines Parametersatzes

Der *bcrypt*-Aufruf zur Generierung eines neuen Moduls, der eine Länge von mod_len Bits aufweist, sowie eines neuen Exponenten d, hat folgende allgemeine Gestalt:

$$bcrypt\ -ms\ -pg\ [\{mod\_len\}\ [\{parameter\_file\}]]$$

Der solchermaßen erzeugte Parametersatz wird in ein durch $parameter\_file$ spezifiziertes File geschrieben, das den eigentlichen Modul N in hexadezimaler Darstellung, sowie den zugehörigen Exponenten d als Dezimalzahl enthält.

Fehlt einer der beiden Parameter mod_len bzw. $parameter\_file$, so wird er zur Laufzeit vom Benutzer angefordert.

Nach Vollzug dieses Schrittes muß vor Anwendung des Verfahrens von Micali-Schnorr zur Erzeugung pseudozufälliger Zahlen noch ein geeigneter Startwert erzeugt werden:

B.3.2 Startwertgenerierung

Da der zu erzeugende Startwert *seed* direkt von dem zugehörigen Modul N sowie dem verwendeten Exponenten d abhängt, setzt die Erzeugung eines solchen Wertes die Existenz eines geeigneten Parameterfiles voraus. Die allgemeine Form einer *bcrypt*-Kommandozeile zur Generierung eines neuen Startwertes ist:

$$bcrypt\ -ms\ -seed\ [\{parameter\_file\}\ [seed\_file]]$$

Das optionale Argument $seed\_file$ ermöglicht es, den erzeugten Startwert in Hexadezimaldarstellung auf eine Datei zu schreiben. Fehlt dieser Parameter, wird hierfür die Standardausgabe *stdout* verwendet.

Nach der Erzeugung eines Parametersatzes, sowie eines zugehörigen Startwertes, kann mit der Ver- bzw. Entschlüsselung von Daten fortgefahren werden:

B.3.3 Ver- bzw. Entschlüsselung

Der hierfür notwendige *bcrypt*-Aufruf hat große Ähnlichkeit mit den OFB-basierten Generatoren aus Abschnitt B.1.2; lediglich der zu verwendende Parametersatz muß zusätzlich spezifiziert werden:

$$bcrpyt\ -ms \left\{ \begin{array}{llll} -s & [\{seed\} & [\{parameter\_file\} & [infile\ [outfile]]]] \\ -sf & [\{seed\_file\} & [\{parameter\_file\} & [infile\ [outfile]]]] \end{array} \right\}$$

Der Schalter $-s$ erwartet den zu verwendenden Startwert in Form einer Hexadezimalzahl als nächstes Argument, während $-sf$ diesen Wert aus einem durch den nachfolgende Kommandozeilenparameter spezifizierten File liest. Der zu verwendende Parameterfile wird durch $parameter\_file$ festgelegt. Fehlt einer dieser Werte, wird er von *bcrypt* vom Benutzer erfragt.

Optional hingegen ist die Angabe eines Ein- bzw. Ausgabefiles. Bei Auslassung dieser Angaben wird die Standardeingabe *stdin* bzw. Standardausgabe *stdout* verwendet.

B.4 Der Impagliazzo-Naor-Generator

Dieses Verfahren ist ähnlich dem eben beschriebenen Generator aufgebaut, was seine Benutzerschnittstelle betrifft. Zu seiner Anwendung muß ein entsprechender Parametersatz sowie ein zugehöriger Startwert erzeugt werden.

Dieser Parametersatz besteht aus den Elementen des n-elementigen Vektors $\vec{a}$, der in folgender Summation

$$f(\vec{a}, \vec{x}) = \vec{a}, \sum_{i=1}^{n} a_i x_i \bmod 2^{l(n)} \tag{B.3}$$

eingesetzt wird, wobei die a_i eine Länge von jeweils $l(n)$-Bits aufweisen. Hiermit wird die Charakteristik eines solchen Parametersatzes durch die beiden Werte n bzw. $l(n)$ festgelegt.

Die Kommandozeile zur Erzeugung eines solchen Parametersatzes hat folgende Gestalt[7]:

$$bcrypt\ -inaor\ -pg\ [\{n\}\ [\{l(n)\}\ [\{parameter\_file\}]]]$$

Die Gestalt des zu erzeugenden Parametersatzes wird durch die Werte der Argumente n bzw. $l(n)$ bestimmt. $parameter\_file$ sepzifiziert die Datei, die der Aufnahme des neuen Parametersatzes dienen soll. Keine dieser Angaben ist optional – im Falle ihres Fehlens werden sie interaktiv vom Benutzer erfragt.

Sowohl n, als auch $l(n)$ unterliegen gewissen Einschränkungen hinsichtlich ihres Wertebereiches. Die Wahl eines Wertes für n muß eine gewisse obere Grenze einhalten, die in Form der Konstanten $INAOR\$MAX\_PAR$ in der Header-Datei *bitstream_lib.h* definiert ist.

[7] Hierbei ist zu beachten, daß das Verfahren von Impagliazzo-Naor durch Angabe von $-inaor$ als erstem Parameter von *bcrypt* selektiert wird.

Abgesehen von dieser implementationsabhängigen Schranke (eine dynamische Verwaltung der benötigten Variablen des Typs *MP_INT* ist denkbar, wurde jedoch noch nicht implementiert) für n, weist $l(n)$ einige verfahrenstechnische Besonderheiten auf: Einerseits muß gewährleistet sein, daß für $l(n)$ stets die Bedingung $l(n) \geq n$ erfüllt ist. Andererseits gilt nach [8] als obere Schranke $l(n) < c \cdot n$, wobei $1 < c < 1.5$ gilt.

Der Qualifizierer $-l$ ermöglicht die Bestimmung der Werte n bzw. $l(n)$ für einen gegebenen Parametersatz:

$$bcrypt \ -inaor \ -l \ [\{parameter\_file\}]$$

Hierbei ist die Angabe des betreffenden Parameterfile nicht optional und wird im Falle ihres Fehlens von *bcrypt* erfragt.

Der nächste Schritt nach Erzeugung eines neuen $\vec{a}$ ist die Erstellung eines geeigneten Startwertes $\vec{x}$, der als Steuervektor der Summation B.3 dient:

B.4.1 Startwertgenerierung

Ein solcher Startwert wird lediglich durch die Anzahl der Elemente aus $\vec{a}$ bestimmt. Der Aufruf zu seiner Erzeugung hat folgende allgemeine Gestalt:

$$bcrypt \ -inaor \left\{ \begin{array}{lll} -seed & [\{parameter\_file\} & [seed\_file]] \\ -seed & [-l \ [\{n\} & [seed\_file]]] \end{array} \right\}$$

Der erste Fall liest die für die Erzeugung eines neuen Startwertes benötigte Elementanzahl des zugehörigen Parametersatzes aus dem Parameterfile selbst, wohingegen die zweite Variante diesen Wert für n als nächstes Argument der Kommandozeile erwartet. Fehlt einer dieser Parameter, wird er zur Laufzeit vom Benutzer erfragt.

Der letzte Parameter, $seed\_file$, ist ein optionales Argument, das die Spezifikation eines Ausgabefiles für den neuen Startwert ermöglicht, der stets in Form einer Hexadezimalzahl ausgegeben wird. Fehlt ein expliziter Filename, wird der erzeugte Wert auf die Standardausgabe *stdout* kopiert.

Mit dem Vorliegen eines geeigneten Parametersatzes sowie eines zugehörigen Startwertes kann nun das Verfahren von Impagliazzo-Naor zur Verschlüsselung von Nutzdaten eingesetzt werden:

B.4.2 Ver- bzw. Entschlüsselung

An dieser Stellt kann direkt auf Abschnitt B.3.3 verwiesen werden, da sich der Aufruf des Impagliazzo-Naor-Verfahrens in dieser Betriebsart analog zu dem dort erläuterten gestaltet. Aus diesem Grunde sei an dieser Stelle lediglich die allgemeine Form des entsprechenden *bcrypt*-Aufrufs angegeben:

$$bcrpyt -inaor \left\{ \begin{array}{llll} -s & [\{seed\}] & [\{parameter\_file\} & [infile \ [outfile]]]] \\ -sf & [\{seed\_file\}] & [\{parameter\_file\} & [infile \ [outfile]]]] \end{array} \right\}$$

Anhang C

Installation

Die mit diesem Kapitel vorliegende Installationsanleitung der implementierten Bibliothek, sowie des zugehörigen Programms *bcrypt*, gliedert sich in zwei Teile: Zum einen werden die Installation und ihre Besonderheiten im Hinblick auf VMS-Systeme, zum anderen die Einrichtung auf UNIX-basierten Rechnern betrachtet.

Für beide Systeme liegen die betreffenden Quellcodemodule in Form von Archivdateien vor, die auf je einer MSDOS-lesbaren HD-Diskette verfügbar sind.

Die Savesets für UNIX bzw. VMS unterscheiden sich lediglich in der Hinsicht, daß der VMS-Kit bereits eine vorübersetzte Version der Bitstrombibliothek sowie des Programms *bcrypt* enthält. Desweiteren beeinhaltet er eine Minimalversion der GNU-MP Bibliothek (*LIGBMP.OLB*).

Aufgrund der Architekturvielfalt unter UNIX wurde auf Zugabe dieser Dateien Verzicht geübt[1], so daß hier nur die Quelltexte, sowie ein Makefile zur Übersetzung der einzelnen Module sowie der eigentlichen Bitstrombibliothek beigefügt wurden[2].

Eine Liste der einzelnen in den Savesets enthaltenen Dateien zeigt Tabelle C.1. Da sowohl die Bibliothek *bitstream_lib.c* selbst, als auch das Rahmenprogramm *bcrypt.c* auf der GNU-MP Bibliothek beruhen, erwarten sie für die Übersetzung die Headerdateien *gmp.h*, *gmp-impl.h*, bzw. *urandom.h* in ihrem Verzeichnis. Diese sind in den Savesets für beide Betriebssysteme enthalten und entstammen der GNU-MP Library, Version 1.3.2.

Ebenfalls erwarten die beiden Make-Prozeduren sowohl unter VMS, als auch unter UNIX das Vorhandensein der GNU-MP Bibliothek *LIBGMP.OLB* bzw. *libgmp.a* in dem aktuellen Verzeichnis, was jedoch durch Modifikation der Dateien *MAKE.COM* bzw. *Makefile* leicht abänderbar ist, da lediglich hier das Linken des Programms *bcrypt* modifiziert werden muß.

[1] Die Bibliothek wurde unter HP-UX, AIX, ULTRIX, sowie OSF/1 getestet.

[2] Auf die Erstellung eines eigenen Makefiles wurde verzichtet, da sich das System aus nur wenigen Modulen zusammensetzt, die im Bedarfsfall schnell manuell übersetzt und gelinkt werden können.

File	Mitgeliefert für		Funktion
	VMS	UNIX	
bitstream_lib.c	✓	✓	Enthält die Routinen der Bitstrom-Library
bitstream_lib.h	✓	✓	Konstantendefinitionen
bitstream_prcdef.h	✓	✓	Funktionsköpfe
bitstream_lib.olb	✓		Fertige Bibliothek für VMS
crypt.c	✓	✓	IDEA-Routinen von Richard De Moliner
crypt.h	✓	✓	Headerfile hierzu
d3des.c	✓	✓	DES- und D3DES-Routinen von Richard Outerbridge
d3des.h	✓	✓	Headerfile zu d3des.c
md2.c	✓	✓	MD2-Hashfunktion von RSA Data Security, Inc.
md2.h	✓	✓	Header zu md2.c
md4.c	✓	✓	MD4-Hashfunktion von RSA Data Security, Inc.
md4.h	✓	✓	Headerfile zu md4.c
md5.c	✓	✓	MD5-Hashfunktion von RSA Data Security, Inc.
md5.h	✓	✓	Headerfile zu md5.c
global.h	✓	✓	RSAREF-Typen und -Konstanten zu MD2,4,5
libgmp.olb	✓		Benötigter Teil der GNU-MP Bibliothek für VMS
gmp.h	✓	✓	Headerfile für GNU-MP
gmp-impl.h	✓	✓	"
gmp-mparam.h	✓	✓	"
urandom.h	✓	✓	"
bcrypt.c	✓	✓	Das Rahmenprogramm
bcrypt.exe	✓		Ausführbarer Code hierzu für VMS (6.1)
make.com	✓		Installationsprozedur für VMS
Makefile		✓	Makefile für UNIX

Tabelle C.1: Die Backup-Savesets

C.1 VMS

Alle zur Installation der Bibliothek *bitstream_lib.olb* sowie des Rahmenprogrammes *bcrypt* notwendigen Files befinden sich in einem Backup-Saveset des Namens *BITSTREA.B_Z*[3]. Zur eigentlichen Installation ist wie folgt vorzugehen:

Kopieren der Datei: Am zweckmäßigsten wird hierzu das Übertragungsprotokoll *ftp* von einem PC aus genutzt. (Dieser Name der Datei kann für die folgenden Schritte beibehalten werden.)

Dekomprimieren: Voraussetzung für diesen Schritt ist das Programm *COMPRESS* unter VMS, das sich in der *DECUS*-Bibliothek befindet:

```
$ DECOMPRESS BITSTREA.B_Z
```

Entpacken des Savesets: `$ BACKUP BITSTREA.B/SAVE *`

Übersetzen und Linken: Dieser Schritt ist nur notwendig, falls die mitgelieferten EXE-Files nicht unter der betreffenden VMS-Version lauffähig sind, oder Änderungen des Quellcodes vorgenommen wurden.

Es wird vorausgesetzt, daß auf dem Zielsystem eine Version des GNU C-Compilers existiert; ist dies nicht der Fall, sind in *MAKE.COM* entsprechende Änderungen vorzunehmen:

- Zuerst muß der zu verwendende C-Compiler in das Symbol *CC* eingetragen werden.
- Weiterhin ist zu beachten, daß das Link-Kommando am Ende der Kommandoprozedur die korrekten Bibliotheken verwendet; an dieser Stelle ist *GNU_CC:[000000]GCCLIB/L* auszukommentieren und eventuell *SYS$LIBRARY:VAXCRTL/L* durch eine andere Bibliothek zu ersetzen[4].

Alle notwendigen Schritte zur Übersetzung der einzelnen Quelltextmodule sowie zur Erstellung einer Library aus den entsprechenden Objectcodemodulen werden von *MAKE.COM* übernommen. Es ist lediglich

```
$ @MAKE
```

aufzurufen. Neben der Generierung der Bibliothek *BITSTREAM_LIB.OLB*, nimmt *MAKE.COM* auch die Übersetzung des Rahmenprogramms *BCRYPT.C* vor. Für dieses, wie für alle anderen Programme, die die Bitstrom-Bibliothek verwenden, wird das Vorhandensein einer funktionsfähigen GNU-MP-Bibliothek vorausgesetzt! Für den Fall, daß diese Library nicht vorhanden ist, enthält der eben verwendete Backupsaveset die Bibliothek *LIBGMP.OLB*, mit der sowohl die Bitstrombibliothek, als auch

[3] Diese Datei wurde unter VMS durch *COMPRESS* gepackt.

[4] **Achtung:** Die VAXC- bzw. DECC-Runtimebibliotheken sind nicht miteinander verträglich. Liegt eine unter VAXC compilierte GNU-MP-Bibliothek vor, müssen auch mit ihr zu linkende Programme unter VAXC compiliert werden. Für DECC gilt entsprechendes.

das Programm *BCRYPT* erstellt werden können. Ebenfalls beinhaltet das Distributionskit die für die GNU-MP Bibliothek benötigten Headerdateien, wie bereits zu Beginn dieses Anhangs erwähnt.

Darüberhinaus befindet sich auf der Distributionsdiskette in Form eines gepackten (*COMPRESS*) Backupsavesets die Version 1.3.2 der GNU-MP Routinen. Die hiermit vorliegende Version wurde speziell für die Verwendung unter VMS angepaßt, wobei sich die hierfür notwendigen Modifikationen auf das Speichermanagement beziehen[5].

Sollen eigene Programme mit der Bitstrom-Library verwendet werden, muß darauf geachtet werden, daß *BITSTREAM_LIB.OLB* **vor** der System-Bibliothek hinzugelinkt wird, da ansonsten viele Referenzen nicht korrekt aufgelöst werden können. Unter GNU C könnte ein entsprechendes Link-Kommando folgende Gestalt besitzen:

```
LINK programm, BITSTREAM_LIB/L, LIBGMP/L, -
               GNU_CC:[000000]GCCLIB/L, SYS$LIBRARY:VAXCRTL/L
```

Die Verwendung des Programms *BCRYPT.EXE* setzt die Definition eines Foreign-Commands voraus, um die Übergabe der Kommandozeilenparameter zu ermöglichen:

```
$ BCRYPT :== $<device>:<pfad>BCRYPT.EXE
```

C.2 UNIX

Alle für die Generierung der Bitstrombibliothek *bitstream_lib.a* sowie des Programms *bcrypt* notwendigen Files befinden sich in Form eines tar-Files (*BITSTREA.TAR* auf einer MSDOS-lesbaren HD-Diskette. Die eigentliche Installation gliedert sich in die folgenden Schritte:

Kopieren des Savesets: In den meisten Fällen wird es sich anbieten, die Datei *BITSTREA.TAR* mit Hilfe des *ftp*-Protokolls auf die Zielmaschine zu kopieren.

Entpacken: `tar xvf bitstrea.tar`

Übersetzen und Linken: Alle für die Erstellung der Bitstrombibliothek *bitstream_lib.a* sowie des Programms *bcrypt* notwendigen Schritte werden durch Verwendung des Kommandos *make* vollzogen. Das hierfür mitgelieferte *Makefile* geht dabei von folgenden Voraussetzungen aus:

- Es stehen die benötigten GNU-MP Headerfiles in dem Verzeichnis, in dem die Übersetzung vollzogen werden soll, zur Verfügung. Da diese ebenfalls Inhalt des tar-File sind, ist dies in den meisten Fällen gewährleistet. Die mitgelieferten Headerdateien *gmph*, *gmp-impl.h*

[5]Diese Patches gehen auf Herrn *Wolfgang J. Möller* von der *Gesellschaft für wissenschaftliche Datenverarbeitung, Göttingen* zurück.

und *urandom.h* beziehen sich auf das GNU-MP Paket in der Version 1.3.2.

Liegt eine andere Version dieser Bibliothek vor, sind die entsprechenden Dateien zu ersetzen.

- Weiterhin wird erwartet, daß sich die eigentliche GNU-MP Bibliothek *libgmp.a* ebenfalls in dem aktuellen Verzeichnis befindet. Ist dies nicht gegeben, muß entweder durch ein entsprechendes Kopierkommando hierfür Sorge getragen oder die Datei *Makefile* modifiziert werden (die hierfür in Frage kommende Zeile ist durch einen Kommentar kenntlich gemacht).

  Sollen eigene Programme Routinen der Bitstrom-Bibliothek verwenden, müssen sie entsprechend übersetzt und gelinkt werden, wie folgendes Beispiel zeigt:

  ```
  gcc -I. -o <zielname> <quellfile.c> bitstream_lib.a libgmp.a
  ```

  Auch in diesem Fall gilt für *libgmp.a* und die zugehörigen Includefiles *gmp.h*, *gmp-impl.h*, bzw. *urandom.h* das eben Gesagte.

Anhang D

Über die Konstruktion von Pseudozufallsgeneratoren

Dieser kurze Abschnitt soll einige Ergebnisse aus Arbeiten von *Goldreich*, *Levin* und anderen vorstellen, die die Konstruktion von Pseudozufallsgeneratoren auf eine neue, fundierte Grundlage gestellt haben. Wurde bisher meist ein Verfahren vorgeschlagen, um dann anhand bestimmter statistischer bzw. eventuell kryptologischer Gesichtspunkte beurteilt zu werden, gestatten diese neueren Ergebnisse die Konstruktion von Zufallgeneratoren aus beliebigen Funktionen, die lediglich einige verhältnismäßig leicht nachzuprüfende Eigenschaften aufweisen müssen.

Hat man eine solche Funktion mit den im nachfolgenden beschriebenen Eigenschaften gefunden, so läßt sich zeigen, daß aus ihr ein Pseudozufallsgenerator konstruierbar ist.

D.1 Einleitung

D.1.1 Präliminarien

Für die folgenden Betrachtungen ist die Einführung des Begriffes einer Einwegfunktion unerläßlich:

Definition 12:

Eine Funktion $f : \{0,1\}^* \longrightarrow \{0,1\}^*$ heißt (schwache) *Einwegfunktion*, falls sie zwar in polynomialer Zeit berechenbar, jedoch nicht in ebensolcher Zeit invertierbar ist. Mit anderen Worten:

Es exisitert eine Konstante $c > 0$, so daß für alle probabilistischen, in polynomialer Zeit abarbeitbaren Algorithmen A und hinreichend großes $k \in \mathbb{N}$ gilt:

$$\text{Prob}\left(A(f(x), 1^k) \notin f^{-1}(f(x))\right) > \frac{1}{k^c} \tag{D.1}$$

Definition 13:

Eine Funktion $f : \{0,1\}^* \longrightarrow \{0,1\}^*$ heißt *starke* Einwegfunktion, falls für alle probabilistischen Algorithmen A, für konstantes $c > 0$ und hinreichend großes $k \in \mathbb{N}$ gilt:

$$\text{Prob}\left(A(f(x), 1^k) \in f^{-1}(f(x))\right) < \frac{1}{k^c} \qquad . \tag{D.2}$$

Eine *notwendige* Bedingung für die Existenz von Pseudozufallsgeneratoren ist nun die Existenz einer solchen schwachen Einwegfunktion nach (D.1) (der Pseudozufallsgenerator selbst stellt eine Einwegfunktion dar). Allerdings ist nicht bekannt, ob diese Bedingung auch *hinreichend* ist.

Ausgehend von dieser Erkenntnis wurde unter der Voraussetzung, daß die Faktorisierung großer Zahlen nicht in polynomialer Zeit durchführbar ist, beispielsweise der bereits beschriebene Blum-Blum-Shub–Generator entwickelt, der auf der Tatsache beruht, daß unter obiger Annahme die Quadrierung über der Menge der Quadratreste eine Einwegfunktion darstellt.

D.1.2 Levins Bedingung

Der erste, der eine sowohl *hinreichende* als auch *notwendige* Bedingung für die Existenz von Pseudozufallsgeneratoren angab, war *Levin* – hierfür wird zunächst der Begriff einer Funktion, die auf ihren Iteraten Einwegfunktion ist, benötigt:

Definition 14:

Eine Funktion $f : \{0,1\}^* \longrightarrow \{0,1\}^*$ heißt (schwache) Einwegfunktion auf ihren *Iteraten*, falls für konstantes $c > 0$, alle probabilistischen Algorithmen A, sowie hinreichend großes $k \in \mathbb{N}$ gilt[1]:

$$\text{Prob}\left(A(f^{(i)}(x), 1^k) \notin f^{-1}(f(x))\right) > \frac{1}{k^c} \qquad \forall\, i, 1 \leq i \leq k^{\frac{3}{2}}. \tag{D.3}$$

Levin zeigte nun, daß die Existenz einer solchen Funktion sowohl hinreichend, als auch notwendig für die Existenz eines Pseudozufallsgenerators ist. Nachteilig hieran ist, daß die geforderte Bedingung nur schwer nachprüfbar ist – ein Umstand, dem erst durch die Arbeiten von *Goldreich*, *Krawczyk* und *Luby* [3] Rechnung getragen wurde, auf die im folgenden kurz eingegangen wird:

D.2 Goldreich, Krawczyk, Luby

D.2.1 Vorbereitung

Definition 15:

Eine Funktion f heiße *regulär*, falls eine Funktion m existiert, so daß für alle $n \in \mathbb{N}$ und jedes $x \in \{0,1\}^*$ die Kardinalität von $f^{-1}(f(x))$ gleich $m(n)$ ist, d.h. jedes Element der Zielmenge der Funktion f besitzt dieselbe Anzahl von Urbildern[2].

[1] Hierbei repräsentiert $f^{(i)}(x)$ die i-fache Anwendung von f auf x.

[2] Somit ist jede bijektive Funktion regulär mit $m(n) = 1 \qquad \forall\, n \in \mathbb{N}$.

D.2.2 Hauptsatz

> Aus jeder regulären Einwegfunktion kann ein Pseudozufallsgenerator konstruiert werden.

Diese neue Bedingung ist nun von derselben Allgemeinheit wie die Erkenntnis Levins, jedoch in den meisten Fällen erheblich leichter nachprüfbar. So sind beispielsweise viele verhältnismäßig einfache Einwegfunktionen regulär und somit zur Konstruktion von Pseudozufallsgeneratoren geeignet.

Yao zeigte, daß aus jeder Einwegfunktion eine starke Einwegfunktion konstruiert werden kann, wobei diese Konstruktion zudem regularitätserhaltend ist!

Der Beweis von *Goldreich*, *Krawczyk* und *Luby* sei im folgenden skizzenhaft geschildert:

Beweisskizze:

Der Beweis nach [3] benötigt als Ausgangspunkt zunächst eine starke Einwegfunktion, die mit Hilfe des zuvor erwähnten Satzes von Yao aus der vorgegebenen Einwegfunktion unter Erhaltung ihrer Regularität konstruiert wird.

Die eigentliche Schwierigkeit der Beweisführung steckt nun in der Entwicklung eines Verfahrens, das die Konvertierung einer starken Einwegfunktion in eine Funktion, die auf ihren Iteraten eine Einwegfunktion ist, ermöglicht. Dieser Teil erfordert die allergrößten Anstrengungen und ist derart umfangreich, daß kein Versuch seiner Skizzierung unternommen wird; es sei auf [3] verwiesen.

Ist mit Hilfe dieses Verfahrens die Konstruktion einer solchen besonderen Einwegfunktion geglückt, kann die Beweisführung durch Anwendung der Erkenntnis von Levin, die bereits in D.1.2 angeführt wurde, abgeschlossen werden.

D.3 Bemerkung

Interessanterweise ist nicht jede (reguläre) Einwegfunktion auch gleichzeitig Einwegfunktion auf ihren Iteraten, was ein pathologisches Beispiel aus [3] demonstriert:

Sei f eine längenerhaltende Einwegfunktion, d.h. es gilt

$$|f(x)| = |x|, \tag{D.4}$$

und es seien $xy \in \{0,1\}^{2n}$ mit $|x| = |y| = n$. Definiere

$$f'(xy) = 0^{|y|} f(x). \tag{D.5}$$

Dieses f' ist eine Einwegfunktion. Allerdings gilt für jedes xy der obigen Form auch

$$f'(f'(xy)) \quad = \quad f'(0^{|y|} f(x)) \tag{D.6}$$

$$= \quad 0^{|f(x)|} f(0^{|y|}) \tag{D.7}$$
$$= \quad 0^n f(0^n). \tag{D.8}$$

Damit ist aber

$$0^n f(0^n) \in f'^{-1}(0^n f(0^n)), \tag{D.9}$$

d.h. f' ist bei zweimaliger Iteration keine Einwegfunktion mehr.

Anhang E

Parallelisierung von Zufallsgeneratoren

Der vorliegende Abschnitt beruht auf einem Vorschlag von *Goldreich*, *Goldwasser* und *Micali*, der in [11] ausgearbeitet wurde, so daß seine Anwendung speziell auf das bereits in Abschnitt 3.2 behandelte Verfahren von *Micali-Schnorr* bezogen betrachtet werden soll.

E.1 Der parallele Polynomial-Generator

Diese Form eines Pseudozufallsgenerators erzeugt, ausgehend von einem zufälligen Startwert $x \in [1, N^{\frac{2}{d}}]$, einen Baum mit Wurzel x, dessen Knoten Pseudozufallszahlen der Länge l Bits (die sämtlich im Intervall $[1, N^{\frac{2}{d}}]$ liegen) sind. Hierbei gilt: $l = \lfloor \frac{2n}{d} \rfloor$.

Ein Knoten x des Grades s besitzt nun s Nachfolger $x(1), \ldots, x(s)$ und eine Ausgabe $\mathrm{Out}(x)$, die im folgenden definiert werden. Es gelte

$$P(x) \bmod N = \sum_{i=1}^{n} b_i 2^{n-i}, \tag{E.1}$$

d.h. $b_1, \ldots, b_n$ ist die Binärdarstellung von $P(x)$ mit höchstwertigem Bit b_1, wobei $P(x)$ ein Polynom entsprechend 3.2.1 darstellt.

Teilt man nun die $s \cdot l$ höchstwertigsten Bits von $P(x)$ mod N in s Bitblöcke der Länge l auf, so werden die Nachfolgeknoten $x(j)$ wie folgt definiert:

$$x(j) := 1 + \sum_{i=1}^{l} b_{(j-1)l+i} 2^{l-i} \quad \text{für } j = 1, \ldots, s. \tag{E.2}$$

Die Ausgabe $\mathrm{Out}(x)$ des Knotens x erhält man durch

$$\mathrm{Out}(x) := b_{sl+1} \cdots b_n. \tag{E.3}$$

Abbildung E.1 stellt einen solchen Iterationsbaum eines parallelen Polynomial-Generators mit Bezeichnungen der einzelnen Knoten dar[1].

[1] Man sieht, daß der parallele Polynomial-Generator für den Fall, daß alle Knoten lediglich Grad 1 besitzen, identisch mit dem in 3.2 eingeführten sequentiellen Polynomial-Generator ist.

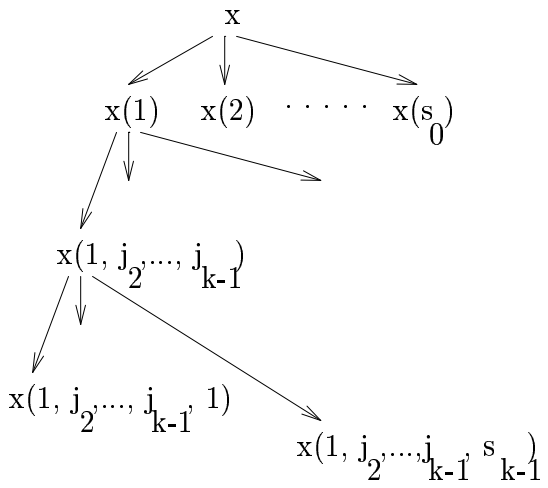

Abbildung E.1: Der parallele Polynomial-Generator

Definition 16:

Die k-Ausgabe $\mathrm{PPG}_{k,P}(x, N)$ eines gegebenen parallelen Polynomial-Generators PPG mit Startwert x wird nun als die Konkatenation aller Bitketten $\mathrm{Out}(x(j_1, \ldots, j_i))$ der Ebenen i mit $0 \leq i \leq k$ definiert:

$$\mathrm{PPG}_{k,P}(x, N) = \prod_{i=1}^{k} \prod \mathrm{Out}(x(\ldots)) \tag{E.4}$$

Analog zu Satz 9 wird in [11] gezeigt, daß auch die k-Ausgabe $\mathrm{PPG}_{k,P}(x, N)$ des parallelen Polynomial-Generators pseudozufällig ist, sofern vorausgesetzt werden kann, daß die Länge dieser k-Ausgabe polynomial beschränkt ist.

Wie man anhand von Abbildung E.1 sieht, läßt sich dieses Parallelisierungsverfahren hervorragend auf Mehrprozessor-Architekturen übertragen, da ausgehend von jedem einzelnen Knoten ein neuer Teilbaum berechnet werden kann, wobei hierfür nur der eine Startwert aus dem Wurzelknoten benötigt wird, so daß ein wirklich linearer Geschwindigkeitszuwachs mit steigender Anzahl von Prozessorelementen möglich ist.

In [11] wird gezeigt, daß nicht unbedingt $d \bmod 2 = 1$ erfüllt sein muß, um die kryptographische Sicherheit in vielen Fällen gewährleisten zu können, was in folgendem Beispiel für eine Vierprozessor-Konfiguration verwendet wird:

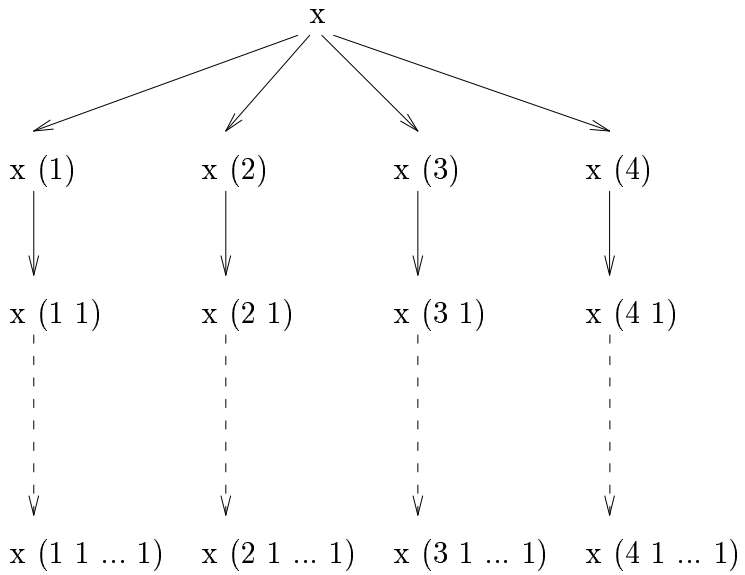

Abbildung E.2: Vierprozessor-Beispiel

Beispiel 5:

N sei das Produkt zweier Primzahlen entsprechend 4.3.1 und besitze eine Länge von 512 Bits. Mit $P(x) = x^8 \bmod N$ läßt sich nun ausgehend von einem Startwert $x \in [1, 2^{128}]$ ein Baum wie in Abbildung E.1 mit vier Knoten pro Ebene konstruieren[2].

Initialisierung: In diesem vorbereitenden Schritt wird zunächst der gewählte Startwert mit Hilfe von $P(x)$ verrechnet. Das Ergebnis hiervon wird nun in vier 128 Bits lange Bitketten $x(1), x(2), x(3), x(4)$ aufgeteilt. Setze außerdem $k := 1$, $\mathrm{PPG}_{1,P}(x, N)$ sei die leere Bitkette.

Iterationsschritt: Für $j = 1, 2, 3, 4$ bestehe $x(j1^k)$ aus den 128 höchstwertigen Bits von $x(j1^{k-1})^8 \bmod N$, während $\mathrm{Out}(x(j1^k))$ aus den verbleibenden 384 unteren Bits hiervon bestehe.

Schleife:

$$\mathrm{PPG}_{k+1,P}(x, N) = \mathrm{PPG}_{k,P}(x, N) \prod_{j=1}^{4} \mathrm{Out}(x(j1^k)) \qquad \text{(E.5)}$$

Es wird $k := k + 1$ gesetzt und mit dem nächsten Iterationsschritt fortgefahren.

In jedem Iterationsschritt liefert der solchermaßen parallelisierte Pseudozufallsgenerator $4 \cdot 384$ Pseudozufallsbits, da jeder der vier Teilbäume ohne jeglichen Kommunikationsaufwand für sich allein berechnet werden kann.

[2] Abbildung E.2 verdeutlicht das Vorgehen.

E.2 Parallelisierung anderer Generatoren

Das im vorangegangenen beschriebene Verfahren zur Parallelisierung des Polynomial-Generators läßt sich auch auf andere Pseudozufallsgeneratoren mit den gleichen Vorteilen übertragen, sofern es sich bei diesen um perfekte Verfahren handelt. Ist diese Voraussetzung nicht gewährleistet, können durch die beschriebene Vorgehensweise Schwächen des zugrundeliegenden Verfahrens eventuell verstärkt werden, so daß vor einer Implementation den sicherheitstechnischen Aspekten ein großes Maß an Aufmerksamkeit geschenkt werden sollte[3].

[3]Gelingt es, den Wert eines Knotens x zu erhalten, wie dies bei einem nicht-perfekten Verfahren nicht auszuschließen ist, sind ab dieser Position alle folgenden Teilzweige des Iterationsbaumes mit ihren Knotenwerten bekannt!

Literaturverzeichnis

[1] L. Blum, M. Blum, M. Shub: *A SIMPLE UNPREDICTABLE PSEUDO-RANDOM NUMBER GENERATOR,* SIAM J. COMPUT Vol. 15, No. 2, May 1986.

[2] J. Eichenauer-Herrmann: Vorlesungsmitschrift *Pseudozufallszahlen*, WS 1994/95, Johannes-Gutenberg-Universität, Mainz.

[3] O. Goldreich, H. Krawczyk, M. Luby: *On the Existence of Pseudorandom Generators,* 29th Annual Symp. on FOCS, 1989.

[4] Torbjörn Granlund: *The GNU Multiple Precision Arithmetic Library*, Edition 1.2, Dezember 1991, Published by the *Free Software Foundation*, 675 Massachusetts Avenue, Cambridge, MA 02139 USA.

[5] Andreas Grube: *Moderne Erzeugung von Zufallszahlen,* S. Toeche-Mittler-Verlag, 1975.

[6] Gerd Hofmeister, Mehdi Djawadi: *Zahlentheorie* nach den Vorlesungen im Wintersemester 93/94 und Sommersemester 1994.

[7] Patrick Horster: *Kryptologie,* BI Wissenschaftsverlag, Reihe Informatik/47, 1987.

[8] Russel Impagliazzo, Moni Naor: *Efficient Cryptographic Schemes Provably as Secure as Subset Sum,* 30th Annual Symp. on FOCS, 1989.

[9] Birger Jansson: *Random Number Generators,* Victor Pettersons Verlag, Stockholm, 1966.

[10] Donald E. Knuth: *Seminumerical Algorithms*, Band 2 von *The Art of Computer Programming*, Addison Wesley, 1981.

[11] S. Micali, C.P. Schnorr: *EFFICIENT, PERFECT RANDOM NUMBER GENERATORS,* Lecture Notes in Computer Science 403, 1989.

[12] National Technical Information Service: *DATA ENCRYPTION STANDARD FACT SHEET,* U.S. Department of Commerce, 5285 Port Royal Road, Sprinfield, VA 22161.

[13] *Federal Information Processing Standards Publication 46,* National Bureau of Standards, Department of Commerce, FIPS PUB 46, 1997 January 15.

[14] Prof. Dr. K. Pommerening: Vorlesungsmitschrift *Kryptographie I/II*, Johannes-Gutenberg-Universität, Mainz.

[15] Prof. Dr. K. Pommerening: Skriptauszug *Kryptographie, Kapitel V, Perfekte Zufallszahlen*, S. 179-251, Johannes-Gutenberg-Universität, Mainz.

[16] Press, Flannery, Teukolsky, Vetterling: *Numerical Recepies in Pascal,* Cambridge University Press, 1989.

[17] Sidney Siegel: *NONPARAMETRIC STATISTICS FOR THE BEHAVIOURAL SCIENCES*, McGRAW-HILL BOOK COMPANY, INC., 1956.

[18] SPSS Inc.: *SPSS© Statistical Algorithms*, 2nd Edition 1991, SPSS Inc., 444 North Michigan Avenue, Chicago, IL 60611.

[19] Reinhard Wittenberg: *Grundlagen computerunterstützter Datenanalyse*, Gustav Fischer Verlag, Stuttgart, 1991.

Hiermit versichere ich, daß ich die vorliegende Arbeit selbständig verfaßt und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Mainz, den 13. September 1995