Studienarbeit

Vergleichende Analyse und VHDL-basierte Implementation von Zufallszahlengeneratoren auf Chipkarten (SmartCards)

Universität Hamburg

Fachbereich Informatik

Autoren:

Peter Hartmann (mail@mathematic.de), Stefan Witt (post@stefan-witt.de)

29. Juni 2001

Zusammenfassung

Zufallszahlen haben die unterschiedlichsten Anwendungsgebiete. Ein immer wichtiger werdendes Einsatzgebiet sind Chipkarten. Die Zufallszahlen werden dort vor allem zur Produktsicherung benutzt. Erzeugt werden sie von Zufallsgeneratoren, die mit mathematischen Verfahren Zahlen liefern. In der vorliegenden Studienarbeit haben wir vier Zufallszahlengeneratoren auf ihre Eignung für eine Chipkartenimplementation untersucht. Die Generatoren erzeugen die Zufallszahlen auf unterschiedliche Weise, mit einem linear rückgekoppelten Schieberegister, mit quadratischen Resten, mit Permutationen und mit Punktverdopplung über einer elliptischen Kurve. Die Ergebnisse werden hier vergleichend dargestellt. Die einzelnen Zufallszahlengeneratoren sind für die statistische Untersuchung in der Programmiersprache C implementiert und für die praktische Untersuchung in der Hardwarebeschreibungssprache VHDL.

Abstract

Random numbers are used in many different domains. An increasingly important domain is smart cards, in which random numbers are mainly used for protection of products. They are generated by random number generators, which provide numbers by mathematical methods. In this study we have examined four different random number generators with respect to their suitability of implementing them on smart cards. The generators produce random numbers in different ways: With a linear feedback shift register, with quadratic residui, with permutations, and with doubling of points over an elliptic curve. The results are presented and compared. All these random number generators have been implemented for the statistic analysis in the programming language C and for the practical analysis in the hardware description language VHDL.

Inhaltsverzeichnis

1 Einleitung

Chipkarten gehören mittlerweile zum alltäglichen Leben und werden mit Selbstverständlichkeit benutzt – beispielsweise Telefonkarten, Geldkarten, oder Krankenversicherungskarten, um nur einige zu nennen. „Ende 2000 waren in Deutschland knapp 18 Mio. Kreditkarten im Umlauf, zusammen mit den EC-Karten befanden sich rund 100 Mio. Zahlungskarten in deutschen Portemonnaies“ (aus: [Barnick, S.12]). Immer dann, wenn es im Zusammenhang mit Chipkarten um Sicherheit geht, so zum Beispiel um Schutz vor dem unerlaubten Kopieren der Karten oder um Authentifikation, spielen Zufallszahlen eine wichtige Rolle. Wegen des großen Bedarfs an Sicherheit ist es daher von Bedeutung, sich mit Zufallszahlengeneratoren auseinanderzusetzen.

In dieser Studienarbeit werden aus diesem Grund vier verschiedene Algorithmen untersucht, die Zufallszahlen erzeugen. Aus dem Anwendungsgebiet der Chipkarten ergeben sich drei fundamentale Anforderungen, die die Zufallsgeneratoren erfüllen müssen:

Erstens müssen sie dazu geeignet sein, auf Chipkarten implementiert zu werden. Durch die Implementation *auf* der Chipkarte ist gewährleistet, dass der Zustand des Zufallsgenerators nicht von außen angesehen werden kann, weil er in der Karte gekapselt ist und die Karte sehr gut gegen unerlaubtes Auslesen geschützt ist. Alle vier in dieser Arbeit untersuchten Zufallsgeneratoren erfüllen diese Eigenschaft, und sie werden exemplarisch in der Hardwarebeschreibungssprache VHDL umgesetzt.

Zweitens müssen die Zufallsgeneratoren deterministisch sein, das heißt mehrere identische Zufallsgeneratoren erzeugen unabhängig voneinander dieselben Zufallszahlen in derselben Reihenfolge. Obwohl diese Eigenschaft in der Literatur im Allgemeinen als nachteilig angesehen wird, ist sie für viele Anwendungen notwendig. Beispielsweise wird der Determinismus benötigt, um Duplikate von Chipkarten aufzudecken, indem die erzeugte Zufallszahlenfolge des Generators auf der Chipkarte mit der Zahlenfolge eines identischen Zufallsgenerators verglichen wird. Alle in dieser Studienarbeit analysierten Generatoren erfüllen die Eigenschaft des Determinismus’.

Die dritte Anforderung ist, dass die Zufallsgeneratoren skalierbar sein sollen. Das bedeutet, dass die Anzahl der Bits, aus denen die Zufallszahlen bestehen, beliebig eingestellt werden kann. Dadurch wird eine skalierbare Sicherheit gewährleistet, an der es großen Bedarf gibt. Sie bietet die Möglichkeit, zwischen verschiedenen Sicherheitsanforderungen beliebig zu wählen. Diese Studienarbeit zielt explizit auf den Low-Security-Bereich ab und setzt deshalb als obere Grenze für die Bitbreite der Zufallsgeneratoren 20 Bit. Im Low-Security-Bereich sind im Allgemeinen 10 Bit bereits ausreichend. Der Ausschluss der Anwendungsgebiete mit sehr hohen Sicherheitsanforderungen ist allerdings kein Nachteil: Der Low-Security-Bereich gewinnt zunehmend an Bedeutung, beispielsweise wird zur Sicherung von Produkten, die nach einem Verfallsdatum wertlos werden, lediglich Low-Security-Schutz benötigt.

Überblick über die folgenden Kapitel

Kapitel 2 stellt verschiedene Anwendungsbereiche vor, in denen Zufallszahlen benötigt werden. Dazu zählen auch Anwendungen, die noch nicht realisiert sind, sondern erst in der näheren Zukunft umgesetzt werden.

In Kapitel 3 werden die Grundlagen vorgestellt, auf die in den darauf folgenden Kapiteln aufgebaut wird. Das sind Zufallszahlen und Zufallsgeneratoren, Gruppentheorie, elliptische Kurven und statistische Verfahren als mathematische Grundlagen sowie eine Einführung in die Technik von Chipkarten.

Kapitel 4 beschreibt Eigenschaften, die Zufallsgeneratoren erfüllen sollen, und Bewertungsverfahren, um diese Eigenschaften zu prüfen.

In Kapitel 5 werden vier ausgewählte Zufallsgeneratoren vorgestellt und die von ihnen erzeugten Zufallszahlen analysiert.

In Kapitel 6 werden die Implementationen in C und VHDL der zuvor vorgestellten Zufallsgeneratoren dargestellt und untersucht.

Kapitel 7 gibt schließlich einen zusammenfassenden Überblick und zeigt weitere Perspektiven auf.

2 Motivation – Wozu Zufallszahlen?

In diesem Kapitel sollen verschiedene Anwendungsbereiche für Zufallszahlengeneratoren vorgestellt werden. Dabei liegt das Hauptaugenmerk auf solchen Anwendungsgebieten, in denen Chipkarten eingesetzt werden oder werden könnten. Einige der vorgestellten Anwendungen sind noch nicht realisiert, werden aber in der näheren Zukunft fraglos Einzug in das alltägliche Leben erlangen und immens an Bedeutung gewinnen.

2.1 Schutz vor unerlaubtem Kopieren von Chipkarten

Der wichtigste Anwendungsbereich für Zufallsgeneratoren auf Chipkarten ist der Schutz vor dem unberechtigten Kopieren. Es gibt grundsätzlich zwei Möglichkeiten, wer versuchen könnte, so eine Kopie anzufertigen: Entweder ist es der Besitzer der Karte selbst oder eine Person, die kurzzeitig im Besitz der Karte war, beispielsweise ein Dieb. Wer Interesse haben kann, eine Chipkarte zu duplizieren, hängt ganz von der Art der Chipkarte ab. Für beide Fälle sollen Beispiele angegeben werden.

Kopieren einer Telefonkarte

Der Besitzer einer handelsüblichen Telefonkarte für öffentliche Telefonzellen könnte versuchen, eine illegale Kopie seiner Telefonkarte anzufertigen. Welchen Vorteil er dadurch erhält, ist klar – er verdoppelt oder vervielfacht das Guthaben der Karte. Geschädigt wird durch die kopierte Karte die Telefongesellschaft, die die Telefonkarten verkauft. Sie muss also ihre Telefonkarten vor dem Duplizieren schützen.

Dazu benötigt jede Karte eine eindeutige Kennung, die auf der Karte abgespeichert und der Telefongesellschaft bekannt ist. Jedes Mal, wenn die Karte in einer Telefonzelle benutzt wird, wird diese Kennung an den Rechner der Telefongesellschaft übermittelt, und alle Benutzungszeitpunkte der Telefonkarte werden nachvollzogen. Als Schutz vor dem unerlaubten Kopieren reicht das allein aber nicht aus, weil der Besitzer der Karte die kopierte Karte mit derselben Kennung versehen kann.

Als Lösung werden zusätzlich zwei identische Zufallszahlengeneratoren verwendet: Einer befindet sich auf der Telefonkarte und der andere auf dem Rechner der Telefongesellschaft. Beide Generatoren müssen mit demselben Startwert initialisiert werden (siehe Kapitel 3.1), so dass sie bei jeder erneuten Ausführung beide jeweils dieselbe Zufallszahl erzeugen. Dann ist es möglich, dass die Telefonkarte bei der Benutzung eine Zufallszahl erzeugt, diese an den Rechner der Telefongesellschaft sendet und dieser durch den Aufruf seines Zufallszahlengenerators prüft, ob es sich um dieselbe Zufallszahl handelt. Eine Telefonkarte muss also die richtige Kennung und die jeweils gültige Zufallszahl senden.

Auf diese Weise ist es möglich, festzustellen, ob eine Kopie der Telefonkarte angefertigt wurde: Das Original und die Kopie benutzen denselben Zufallszahlengenerator, und daher liefern beide bei der ersten Benutzung nach dem Duplizieren dieselbe Zufallszahl. Wenn die Originalkarte nach dem Duplizieren erstmalig benutzt wird, berechnet sie eine Zufallszahl, die der Rechner der Telefongesellschaft als gültig anerkennt. Wenn daraufhin die Kopie benutzt wird, liefert sie genau die Zufallszahl, die die Originalkarte zuvor erzeugt hat. Der Recher der Telefongesellschaft weist diese Zahl als ungültig zurück, weil er bereits die nächste Zufallszahl in der Sequenz der vom benutzten Generator erzeugten Zufallszahlen erwartet. Somit wird die Kopie zurückgewiesen und kann nicht zum Telefonieren benutzt werden.

Es könnte jedoch auch der umgekehrte Fall eintreten, dass zuerst die Kopie benutzt wird. Dann wird das Original zurückgewiesen. Es ist also nicht möglich, festzustellen, welche Karte das Original und welche die Kopie ist, sondern nur, dass eine unerlaubte Kopie angefertigt wurde. Bei dieser Anwendung ist das auch nicht von Bedeutung, weil die Telefongesellschaft trotzdem die Benutzung zweier identischer Telefonkarten verhindern kann.

Kopieren einer Geldkarte

Auch bei Geldkarten, beispielsweise EC-Karten („Electronic Cash“), ist es notwendig, das unerlaubte Kopieren der Karten zu erkennen. Es kann zwar passieren, dass eine Geldkarte entwendet wird, allerdings wird ein Dieb nicht versuchen, die gestohlene Karte zu kopieren, weil er ja mithilfe der Karte direkt auf das Konto des Karteneigentümers zugreifen kann.

Es kann stattdessen der Fall eintreten, dass ein Verkäufer, der ein Kartenlesegerät an der Kasse benutzt, dieses fälscht. Das gefälschte Gerät liest aus der Karte des ahnungslosen Kunden die Kartenkennung aus und speichert die vom Kunden eingegebene Geheimzahl, verhält sich aber ansonsten wie ein echtes Lesegerät, das heißt es übermittelt die Kennung und die Geheimzahl an die Bank. Der Kunde bemerkt den gefälschten Kartenleser nicht, weil die Buchung sich für ihn nicht von einer an einem echten Lesegerät unterscheidet. Durch die Speicherung der Kartenkennung und der Geheimzahl hat der Verkäufer nun aber die Möglichkeit, eine illegale Kopie der Geldkarte anzufertigen, die er dazu benutzen kann, um Geld vom Konto des Kartenbesitzers abzubuchen.

Auch dieser Versuch des unerlaubten Kopierens kann mithilfe von Zufallszahlen verhindert werden. Dazu wird die gleiche Methode benutzt wie bei der Verhinderung des Duplizierens von Telefonkarten (siehe oben, Abschnitt „Kopieren einer Telefonkarte“): Auf der Geldkarte und auf dem Rechner der Bank werden identische Zufallszahlengeneratoren verwendet. Bei jeder Transaktion erzeugt die Geldkarte eine neue Zufallszahl und übermittelt sie an die Bank. Der Rechner der Bank erzeugt ebenfalls die nächste Zufallszahl der Sequenz und prüft, ob sie mit der Zahl übereinstimmt, die die Geldkarte gesendet hat. Wenn das nicht der Fall ist, wird die Geldkarte gesperrt, weil sie als gefälscht erkannt wurde.

Es können hierbei wiederum zwei Fälle eintreten: Entweder führt der Verkäufer mit der gefälschten Karte oder der rechtmäßige Kartenbesitzer mit der Originalkarte die nächste Transaktion oder die nächsten Transaktionen durch. Im ersten Fall wird die Karte des Kunden gesperrt, weil die gefälschte Karte bereits die folgende Zufallszahl an die Bank übermittelt hat und die Originalkarte die bereits ungültig gewordene Zufallszahl an die Bank sendet. Im zweiten Fall wird die Karte des Verkäufers gesperrt, weil sie eine bereits ungültige Zufallszahl übermittelt. Das hat zur Folge, dass auch die Originalkarte gesperrt wird, weil sie dieselbe Kartenkennung besitzt.

In beiden Fällen bemerkt also der rechtmäßige Kartenbesitzer, dass seine Geldkarte gesperrt ist. Außerdem kann die Bank genau nachvollziehen, welcher Verkäufer die Geldkarte gefälscht hat, indem sie die an sie übermittelten Zufallszahlen vergleicht. Die Bank muss nur prüfen, welche Zufallszahl doppelt an sie gesendet wurde. Die Transaktion *vor* dem ersten Auftreten der doppelten Zufallszahl[1] gibt die Transaktion an, bei der der Verkäufer die Karte ausgelesen und gefälscht hat.

2.2 Produktsicherung

Zufallszahlen lassen sich dazu benutzen, unautorisierten Zugriff auf Informationen zu verhindern. Dieses geschieht in engem Zusammenhang mit der Verschlüsselung der betreffenden Informationen.

Die Verschlüsselung der zu schützenden Informationen besitzt einen entscheidenden Nachteil: Sie bietet nicht die Möglichkeit von zeitlich begrenzter Gültigkeit. Mithilfe eines Zufallszahlengenerators hingegen ist es möglich, an definierten Zeitpunkten eine neue Zufallszahl zu generieren. Diese Zeitpunkte werden *Transaktionen* genannt. Im Folgenden werden Anwendungen vorgestellt.

Wechsel von Zuständigkeiten

Mittels der Zufallszahlen ist es möglich, ein Produkt in einer Wertschöpfungskette vor Manipulation zu schützen. In einer solchen Kette sind nacheinander verschiedene Personen für das Produkt zuständig, wobei jede der Personen mit einem Gerät für den Zugriff auf den Gegenstand ausgestattet sein muss. Wünschenswert ist es, jeder der an der Wertschöpfung beteiligten Personen nur so lange Zugriff auf das Produkt zu gewähren, wie es nötig ist.

Eine Lösung hierfür ist die Verwendung eines Zufallszahlengenerators. Jeder produzierte Gegenstand wird mit einem Mikrochip ausgestattet, der einen Zufallsgenerator enthält, und jeder Bearbeiter erhält ein Gerät,

[1] In der Sequenz der Zufallszahlen kann eine Zahl selbstverständlich doppelt auftreten, obwohl keine Fälschung vorliegt. Dann hat die Bank keine Möglichkeit, festzustellen, wann die Karte gefälscht wurde.

das mit genau einer Zufallszahl der Sequenz der vom Generator erzeugten Zufallszahlen ausgestattet ist. Das Gerät wird zum Zugriff auf die Produktdaten benutzt.

Dabei kennt das Gerät des ersten Bearbeiters in der Wertschöpfungskette die erste Zufallszahl der Sequenz, das Gerät des zweiten Bearbeiters die zweite Zufallszahl und so weiter. Sobald eine Übergabe des produzierten Gegenstandes an den folgenden Bearbeiter erfolgt, wird er von der Person abgezeichnet, die ihn weitergibt. Diese Abzeichnung erfolgt mithilfe des Gerätes des Bearbeiters. Dabei sendet das Abzeichnungsgerät die Zahl an den Mikrochip, und dieser vergleicht sie mit der zuletzt mit seinem Generator erzeugten Zufallszahl. Wenn die Zahlen übereinstimmen, ist das Gerät zum Abzeichnen berechtigt, und der Zufallsgenerator erzeugt die nächste Zufallszahl. Damit ist ausschließlich das Gerät des folgenden Bearbeiters zum Zugriff auf die Daten im Chip und zum nächsten Abzeichnen berechtigt.

Ausleseschutz

Unabhängig von Transaktionen sind Zufallszahlen dazu geeignet, mit Chipkarten ausgestattete Produkte vor dem unautorisierten Auslesen zu schützen, beispielsweise durch die Konkurrenz oder Spione.

Dazu wird die gleiche Methode wie im obigen Abschnitt „Wechsel von Zuständigkeiten“ benutzt, so dass nur ein Lesegerät den Chip auslesen kann, das über die jeweils gültige Zufallszahl verfügt.

Diebstahlsicherung

Zufallszahlen können auch für die Diebstahlsicherung verwendet werden. Das soll anhand zweier Beispiele erläutert werden.

Die elektronischen Komponenten in einem Auto sind über einen gemeinsamen Systembus verbunden und kommunizieren miteinander. Nun kann mithilfe von Zufallsgeneratoren das nachträgliche Einbauen einer gestohlenen Komponente verhindert werden. Dazu muss jede der elektronischen Komponenten mit dem gleichen Zufallsgenerator ausgestattet werden. Jedes Mal, wenn der Fahrer das Auto startet, wird in allen Komponenten die nächste Zufallszahl erzeugt. Wenn alle Komponenten Originale sind, müssen alle Zufallszahlen übereinstimmen.

In dem Fall, dass eine der Komponenten ausfällt, muss eine zentrale Komponente mitzählen, wie oft sich eine Komponente nicht authentifiziert hat und bei jedem Start des Autos eine um eins erhöhte Anzahl an Zufallszahlen verlangen. Die ausgefallene Komponente muss nach ihrer Reparatur mehr als eine Zufallszahl senden, und wenn die letzte dieser Zufallszahlen mit der aktuell gültigen Zahl übereinstimmt, wird die Komponente akzeptiert.

Wenn ein Betrüger eine gestohlene Komponente in das Auto einbauen will, beispielsweise einen Airbag (die Airbag-Sicherung ist momentan aktuell), muss er den Zufallsgenerator kennen, damit die neue Komponente mit der schon im Auto vorhandenen Elektronik zusammenarbeitet. Wenn das nicht der Fall ist, wird das falsche Teil vom Auto erkannt, und das Auto kann den Dienst verweigern oder nur in abgeschwächter Form erbringen.

Momentan in der Entwicklung sind Systeme, die im Falle falscher Komponenten nach und nach das Auto unbrauchbar werden lassen, beispielsweise durch langfristig abnehmende Leistung des Motors. Das Auto verliert dadurch an Wert, so dass sich das Einbauen gestohlener Komponenten nicht lohnt. Momentan aktuell ist die Sicherung von Airbags mithilfe der beschriebenen Methode, so dass gestohlene Airbags nicht in anderen Autos verwendet werden können. Ein Airbag, der eingebaut wird und nicht mit dem Bordsystem zusammenarbeitet, löst sofort aus. Eine Dienstverweigerung des Airbags bei einem Unfall kommt nicht in Frage, weil sie gesetzlich nicht zulässig ist.

Eine weitere Anwendung zur Diebstahlsicherung sind ferngesteuerte Autoschlösser, die mit einem Funksender anstatt eines Autoschlüssels bedient werden. Beim Senden des Funksignals wird ein Code übermittelt, den der Empfänger im Auto auf Gültigkeit prüft. Wenn er korrekt ist, öffnet oder schließt er die Türverriegelung, ansonsten nicht. Ein Angreifer könnte einen Empfänger in der Nähe des Autos platzieren und das übermittelte Funksignal mitlesen, so dass er den korrekten Code zum Öffnen der Tür mithilfe eines entsprechend angepassten Senders an den Empfänger des Autos senden kann. Der Empfänger würde den Code als korrekt anerkennen und die Türen öffnen. Dieser Angriff ist als *Replay-Attacke* bekannt.

Um diesen Angriff zu verhindern, müssen Sender und Empfänger mit identischen Zufallsgeneratoren ausgestattet werden. Der Sender sendet also den Schloss-Code und die nächste Zufallszahl in der Sequenz. Nur wenn der Code korrekt ist und die Zufallszahl mit der des Empfängers übereinstimmt, wird das Schloss geöffnet. Da aber jedes Auto über einen legalen Zweitschlüssel verfügt, muss das Verfahren etwas erweitert werden: Wenn der Zweitschlüssel verwendet wird, muss er alle vom Erstschlüssel bereits gesendeten Zufallszahlen senden und zusätzlich die nächste in der Sequenz, weil diese einem Angreifer nicht bekannt sein kann. Eine Alternative zu diesem Verfahren ist die Verwendung verschiedener Zufallszahlengeneratoren für jeden legalen Schlüssel.

2.3 Authentifikation

Zufallsgeneratoren können für die Authentifikation von zwei Kommunikationspartnern, zum Beispiel Nutzer oder Gesprächspartner, verwendet werden. Unter Authentifikation versteht man den Nachweis, dass eine Person oder ein Objekt tatsächlich die Identität besitzt, die sie bzw. es vorgibt zu sein. Beispielsweise ist eine Passwortabfrage bei der Anmeldung an einem Rechner ein Authentifikationsvorgang.

Zufallszahlen statt Transaktionsnummern

Im Bereich der Tätigung von Bankgeschäften vom heimischen Rechner aus („Home-Banking“) werden zurzeit sehr oft so genannte Transaktionsnummern (TANs) verwendet. Bei jeder Transaktion, die der Kunde durchführen möchte, muss er eine solche Transaktionsnummer angeben, mit der er sich authentifiziert. Er erhält von der Bank eine Liste, die für sein Konto gültige Transaktionsnummern enthält, und kann bei jeder Transaktion eine Nummer aus dieser Liste angeben. Dabei ist jede Nummer genau einmal gültig, das heißt der Kunde benötigt erneut eine Liste, sobald er alle Transaktionsnummern einmal benutzt hat.

Die Authentifikation mittels Transaktionsnummern ist in mehrfacher Hinsicht mit Nachteilen behaftet. Zum einen ist es für den Kunden recht umständlich, bei jeder Transaktion eine Nummer aus der Liste herauszusuchen, und zum anderen hat die Liste den gravierenden Nachteil, dass der Kunde sie verlieren kann. Jeder, der die Liste der Transaktionsnummern und die Kontonummer besitzt, ist dazu imstande, Buchungen auf dem entsprechenden Konto durchzuführen.

Das Verfahren kann unter der Verwendung von Zufallszahlengeneratoren wesentlich vereinfacht werden: Sowohl der Rechner der Bank als auch der des Kunden benutzen denselben Zufallsgenerator. Das bedeutet, dass die jeweils nächste gültige Transaktionsnummer vom Zufallsgenerator des Kunden erzeugt wird. Sie wird, genauso wie die Transaktionsnummern des Listenverfahrens, an den Rechner der Bank übermittelt, und dieser kann mithilfe seines identischen Zufallsgenerators die übermittelte Zahl prüfen. Auf diese Weise muss der Kunde nicht mehr die Liste seiner gültigen Transaktionsnummern verwalten und geht auch nicht das Risiko ein, dass er die Liste verliert.

Schlüsselaustausch

Wenn zwei Kommunikationspartner über einen sicheren Kanal kommunizieren möchten, müssen sie dazu die auszutauschenden Daten verschlüsseln. Für eine schnelle Verschlüsselung ist ein symmetrisches Verfahren, beispielsweise Triple-DES („Data Encryption Standard“) notwendig. Das Problem, das dabei auftritt, ist das Vereinbaren eines gemeinsamen Schlüssels, weil zur Vereinbarung die Kommunikation nur über den unsicheren Kanal möglich ist.

Zur Vereinbarung eines Schlüssels gibt es Verfahren, die so genannten *Zero-Knowledge-Protokolle*. Ihnen ist gemeinsam, dass sie einen Zufallszahlengenerator benötigen.

Im Folgenden soll eines der Zero-Knowlegde-Protokolle vorgestellt werden, und zwar der Schlüsselaustausch nach dem Verfahren von Diffie und Hellman, der so genannte *Diffie-Hellman-Key-Exchange* (nach [Tanenbaum 96, S. 624f.]). Das Verfahren ist in Abb. 1 graphisch veranschaulicht.

Angenommen, Alice möchte mit Bob[2] einen gemeinsamen Schlüssel vereinbaren. Dazu wählen Alice und Bob mithilfe eines Zufallszahlengenerators jeweils eine große Zufallszahl: Alice wählt x, und Bob wählt y.

Dann wählt Alice zwei große Primzahlen n und g, wobei $\frac{n-1}{2}$ auch eine Primzahl sein muss.

Alice sendet dann n, g und $g^x \bmod n$ an Bob. Das Senden kann über einen unsicheren, das heißt abhörbaren Kommunikationskanal erfolgen. Als Antwort sendet Bob die Zahl $g^y \bmod n$ an Alice.

Sodann stehen Alice und Bob alle Informationen zur Verfügung, um den Schlüssel zu berechnen: Alice berechnet

$$(g^y \bmod n)^x = g^{xy} \bmod n,$$

und Bob berechnet

$$(g^x \bmod n)^y = g^{xy} \bmod n.$$

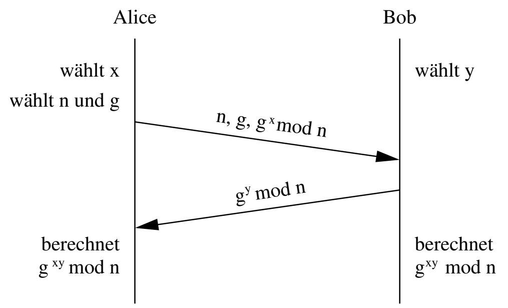

Abbildung 1: Ort-Zeit-Diagramm des Schlüsseltauschs nach der Methode von Diffie und Hellman. Horizontal sind die Kommunikationspartner dargestellt, und die Zeitachse verläuft vertikal nach unten (nach [Tanenbaum 96, S. 625]).

Ein Angreifer, der die ausgetauschten Nachrichten mithört, verfügt zwar über alle Informationen, um den Schlüssel $g^{xy} \bmod n$ zu berechnen, jedoch ist die Berechnung in der Praxis wegen des sehr hohen Aufwandes nicht machbar. Der Angreifer müsste aus $g^x \bmod n$ und $g^y \bmod n$ die Werte x bzw. y (so genannte diskrete Logarithmen) berechnen, aber für die Berechnung diskreter Logarithmen ist kein Algorithmus bekannt, der deterministisch in Polynomzeit arbeitet.

2.4 Stromchiffrierung mit Zufallszahlen

Ein weiteres wichtiges Einsatzgebiet für Sequenzen von Zufallszahlen ist der Einsatz in der Stromchiffrierung. Die Stromchiffrierung verschlüsselt im Gegensatz zur Blockchiffrierung den Klartext bitweise und nicht in Bitgruppen (siehe [Schneier 96, S. 4]). Der Algorithmus wird als *One-Time-Pad* bezeichnet und wurde 1917 von Joseph Mauborgne und Gilbert Vernam bei AT&T erfunden (siehe [Kahn 96, S. 403]).

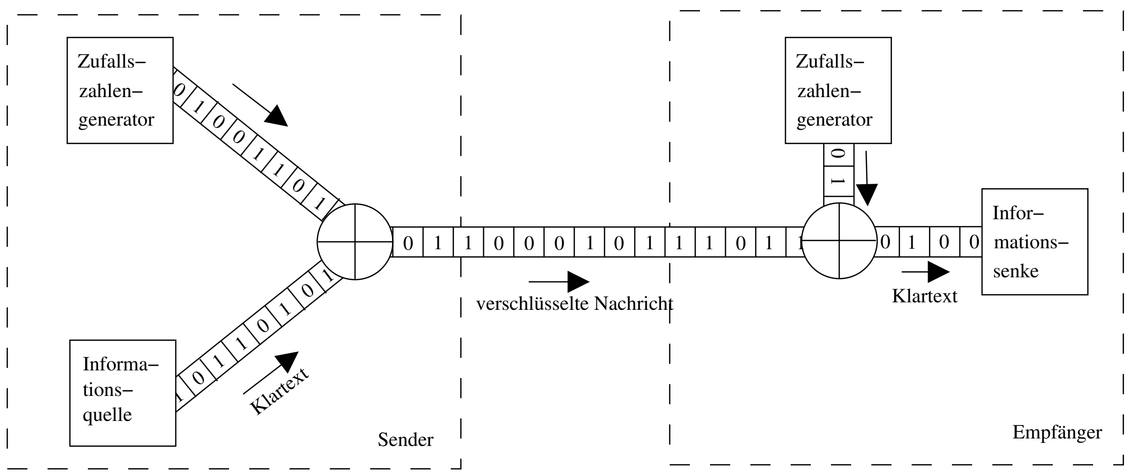

Abbildung 2: Prinzip der Stromchiffrierung. Dargestellt sind die Flüsse der Bitströme. Die beiden Zufallszahlengeneratoren sind identisch, das heißt sie generieren beide denselben Bitstrom. Die eingezeichneten Nullen und Einsen sind willkürlich gewählt und dienen lediglich der Illustration.

Wurden vor Erfindung des Computers noch Lochstreifen benutzt und die Rechenoperationen auf Buchstaben angewandt, so wird heute für die Verschlüsselung des Klartextes der Bitstrom, der den Klartext darstellt, mit dem Bitstrom der Zufallszahlensequenz addiert, also eine einfache Exklusiv-Oder-Verknüpfung (XOR-Verknüpfung) durchgeführt. Für die Entschlüsselung wird der verschlüsselte Bitstrom nochmals mit dem Bitstrom der Zufallszahlensequenz addiert, weil: $a \oplus b \oplus b = a$ gilt (siehe Abb. 2).

[2] Alice und Bob sind die in der Kryptologie üblichen Beispielnamen.

Der Stromchiffierurungs-Algorithmus hat zwei entscheidende Vorteile gegenüber anderen Verschlüsselungsalgorithmen. Er ist sehr schnell und benötigt kaum Hardware- oder Rechenaufwand, da für das Ver- bzw. Entschlüsseln nur die XOR-Verknüpfung ausgeführt werden muss. Der zweite entscheidende Vorteil ist die absolute Sicherheit. Ist die Zufallszahlensequenz wirklich zufällig, also nicht vorhersagbar, dann bleibt der verschlüsselte Klartext für immer unbekannt. Da jeder Klartext gleich wahrscheinlich ist, gibt es für einen unbefugten Dritten keine Möglichkeit, zu ermitteln, welcher davon der richtige ist. Eine echte Zufallszahlensequenz, die zu einem nicht zufälligen Klartext addiert wird, erzeugt einen völlig willkürlichen Chiffretext. Keine noch so überragende Verarbeitungsleistung kann daran irgendetwas ändern. Der Beweis für die Sicherheit ist in [Shannon 49] zu finden.

Nachteilig ist das Problem der Synchronisation. Sender und Empfänger müssen perfekt synchronisiert sein. Geht bei der Übertragung ein Bit verloren, dann wird beim Entschlüsseln nur noch Klartext ohne jeden Sinn erzeugt ([Rueppel 86]). Die Sicherheit von diesem Algorithmus hängt von der Qualität der Zufallszahlen ab, erforderlich sind hier Zufallszahlen, die nicht vorhersagbar sind. Besitzt die Zufallszahlensequenz beispielsweise eine Periode und wird der Klartext auch nur zweimal mit den sich wiederholenden Zufallszahlen verschlüsselt, dann ist das One-Time-Pad nicht mehr sicher ([Bauer 97, S. 149]).

2.5 Weitere Anwendungen von Zufallszahlen in der Informatik

Ein weiteres Einsatzgebiet von Zufallszahlen ist die Verwendung bei Problemstellungen, die mittels einer zufälligen Komponente gelöst werden. Das sind beispielsweise die Simulation, das Testen von Programmen oder der Einsatz von randomisierten Algorithmen.

Simulation

Die so genannte *Monte-Carlo-Methode*, auch als Simulationsmethode bezeichnet, dient der approximativen Berechnung von Wahrscheinlichkeiten. Diese Methode macht Gebrauch von dem schwachen Gesetz der großen Zahlen, das besagt, dass das Stichprobenmittel $(X_1 + ... + X_n)/n$ von n Zufallsvariablen eine Approximation für den Erwartungswert EX_1 ist. Benutzt man also für die $X_1, ..., X_n$ die Zufallszahlen $x_1, ..., x_n$, dann kann man das arithmetische Mittel $(x_1 + ... + x_n)/n$ als Approximation für den Erwartungswert EX_1 wählen. Genauere Erklärungen und Beispiele findet man in [Neuhaus 95, S. 294ff.].

Testen von Programmen

Für die Qualitätssicherung ist das Testen von Programmen (siehe [Rechenberg 97, S. 663]) und von integrierten Schaltungen (siehe [Schiffmann 96, S. 269ff.]) ein wichtiger Punkt. Allerdings kann meist wegen der Komplexität kein vollständiger Test durchgeführt werden. Eine Schaltwerk mit n Eingängen und m Speichergliedern erfordert für einen vollständigen Test 2^{m+n} Testmuster. Angenommen, der Test eines Testmusters würde eine Mikrosekunde benötigen, dann würde die Testzeit eines Schaltwerkes mit $m + n = 45$ Eingängen bereits über ein Jahr betragen. Aus diesem Grund wird unter anderem mittels zufälliger Testmuster getestet. Diese lassen sich mit Zufallszahlen realisieren.

Randomisierte Algorithmen

Bei vielen deterministischen Berechnungen ist der Zeitaufwand für die Lösung des Problems zu hoch. Es gibt aber randomisierte Algorithmen, die einige dieser Probleme effizient lösen. Für diese Algorithmen benötigt man Zufallszahlen. Als Beispiel sei hier der Algorithmus von Rabin (siehe [Rabin 76, S. 27ff.]) erwähnt. Dieser testet, ob eine gegebene Zahl eine Primzahl ist oder nicht. Für diesen sehr schnellen Algorithmus werden zufällige natürliche Zahlen benötigt. Wenn für eine zufällige Zahl der Algorithmus die zu untersuchende Zahl als Primzahl ausweist, dann ist das mit Sicherheit wahr. Ist das Ergebnis negativ, so beträgt die Irrtumswahrscheinlichkeit weniger als $0,5$. Testet man eine Zahl einhundert Mal mit Zufallszahlen und ist das Ergebnis immer positiv, dann sinkt die Irrtumswahrscheinlichkeit, dass doch eine zusammengesetzte Zahl vorliegt, auf unter 2^{-100}.

3 Grundlagen

In diesem Kapitel werden die Grundlagen dargestellt, auf die die folgenden Kapitel aufbauen. Dazu zählen Zufallszahlen und deren Erzeugung, mathematische Gruppentheorie, elliptische Kurven, statistische Verfahren und Chipkartentechnik.

3.1 Zufallszahlen

3.1.1 Was sind Zufallszahlen?

Es gibt je nach Standpunkt und Anwendungsbereich unterschiedliche Definitionen von Zufallszahlen. Vom philosophischen Standpunkt aus ist es beispielsweise eine offene Frage, ob Zufall überhaupt existiert. Da hier der praktische Einsatz von Zufallszahlengeneratoren auf Chipkarten im Vordergrund steht, soll diese Frage aber nicht weiter interessieren.

Bei einer gegebenen Zahl, beispielsweise der 42, ist es nicht nachvollziehbar, ob diese Zahl eine Zufallszahl ist oder nicht, da nicht gezeigt werden kann, ob diese Zahl zufällig ermittelt wurde oder das Ergebnis deterministischer Berechnungen ist.

In [Knuth 98, S. 1ff.] wird stattdessen die Definition von Zufallszahlen über Sequenzen von Zufallszahlen angegeben. Diese Folgen von Zahlen werden als zufällig angesehen, wenn die einzelnen generierten Zahlen unabhängig von den vorhergegangenen Zahlen zufällig erzeugt werden. Hier wird es dann problematisch, überhaupt von Zufallszahlen zu sprechen, wenn diese von deterministischen Maschinen erzeugt werden. Diese Zahlen werden eindeutig vorhersagbar nach einem Algorithmus erzeugt. Eine einmal erzeugte Zufallszahlensequenz könnte immer wieder erzeugt werden und ist damit überhaupt nicht mehr zufällig. Abgesehen von theoretischen Modellen, welche in der Lage sind, nichtdeterministisch zu arbeiten, gibt es keine mit Maschinen erzeugten Zufallszahlen. Eine Ausnahme ist die Benutzung des physikalischen Zufalls aus der Realität. Diese allerdings auf Chipkarten nicht praktikable Methode wird unter dem nachfolgenden Punkt besprochen.

Das Ziel ist es also, Sequenzen von Zufallszahlen zu erzeugen, die zufällig aussehen. In diesem Zusammenhang spricht man von *Pseudo-* oder *Quasizufallszahlen*. Die Erzeugung von Pseudozufallszahlen ist gegenüber der Erzeugung von echten Zufallszahlen nicht mit Nachteilen behaftet, im Gegenteil – teilweise sind in den Anwendungen Pseudozufallszahlen erwünscht, siehe dazu Kapitel 2. Der Einfachheit halber wird zukünftig von Zufallszahlen gesprochen, obwohl Pseudozufallszahlen gemeint sind. In enger Anlehnung an [Schneier 96, S. 54f.] wird hier eine weniger formale Definition von Zufallszahlen angegeben.

Definition: Ein Generator erzeugt *Pseudozufallszahlen*, wenn er die folgende Eigenschaft besitzt:

1. Der Generator scheint zufällig zu sein. Das bedeutet, dass die erzeugten Zahlen sämtliche bekannten statistischen Zufallstests bestehen.

Ist zusätzlich die folgende Eigenschaft erfüllt, dann spricht man von einem *kryptographisch sicheren Pseudozufallszahlengenerator*:

2. Die erzeugten Zahlen sind nicht voraussagbar. Es ist unmöglich, zu berechnen, welche Zufallszahl als nächstes kommt, selbst wenn der Algorithmus oder die Hardware, die die Zahlen erzeugen, sowie alle vorhergehenden Zahlen bekannt sind.

Diese zwei Eigenschaften sind für die Beurteilung der hier vorgestellten Zufallszahlengeneratoren maßgebend, wobei die Sicherheit bzw. die Qualität der erzeugten Zufallszahlen entweder von der Komplexität eines mathematischen Problems oder von der Geheimhaltung einer zur Erzeugung der Zufallszahlen erforderlichen Information abhängt. Dabei ist auch abzuwägen, mit welchem Aufwand welche Werte geschützt werden sollen, so dass die Eigenschaft 2 nicht und Eigenschaft 1 nicht besonders gut erfüllt werden müssen, wenn die geschützten Werte gering und demzufolge der Aufwand zur Berechnung der Zufallszahlen in keinem vernünftigen Verhältnis mehr zum erreichten Vorteil steht, der mit der Kenntnis der Zufallszahlen erreichbar ist.

Definition: Von *echten Zufallszahlen* spricht man, wenn zusätzlich zu den schon genannten Eigenschaften noch die folgende erfüllt ist:

3. Der Generator ist nicht zuverlässig reproduzierbar. Wenn man den Generator zweimal mit exakt derselben Eingabe, soweit dies möglich ist, laufen lässt, erhält man zwei Zufallsfolgen, die keinerlei Ähnlichkeiten aufweisen.

3.1.2 Überblick über Zufallsgeneratoren

In diesem Abschnitt soll ein kurzer Überblick über Zufallszahlengeneratoren gegeben werden, dabei ist das Ziel nicht der vollständige Überblick, da dies den Rahmen dieser Arbeit sprengen würde. Es gibt von den hier vorgestellten Generatoren jeweils modifizierte oder kombinierte Versionen, so dass ein umfassender Überblick auch gar nicht möglich ist. Umfangreiche und ausführliche Übersichten zu Zufallszahlengeneratoren findet man in [Schneier 96, S. 425ff.], [Knuth 98, S. 1ff.] und [Härtel 94, S. 39ff.].

Glücksspiele

Eine der ältesten Methoden, um Zufallszahlen zu erzeugen, sind die Glücksspiele ([Hogben 63, S. 250ff.]). Zumindest sollten die benutzten Ergebnisse unabhängig davon, ob nun ein Glücksrad, Würfel, Karten oder anderes benutzt wird, zufällig sein. Die Methode, um heutzutage beispielsweise die Lottozahlen zu erzeugen, ist allgemein bekannt: Eine drehbare große durchsichtige Kugel, in der sich weitere nummerierte Kugeln befinden, wird gedreht, so dass sich die Kugeln, die mit den Lottozahlen bezeichnet sind, vermischen. Bei Stillstand der großen Kugel fällt dann eine von den kleinen Kugeln heraus, und diese bestimmt eine Gewinnzahl. Im Interesse der Lottogesellschaften ist dies abgesehen von der Einschränkung, dass eine bereits gezogene Zahl nicht wieder gezogen werden kann, eine wirklich zufällige Auswahl von Gewinnzahlen. Allerdings ist diese Methode zu langsam und schon gar nicht auf Chipkarten implementierbar. Dieser letztere Nachteil trifft auf alle Methoden zur Erzeugung von echten Zufallszahlen zu.

Echter Zufall

Für die Erzeugung von Zufallszahlen lässt sich selbstverständlich der Zufall aus dem realen Leben benutzen. Dafür existieren die unterschiedlichsten Möglichkeiten.

Bereits 1955 wurde von der RAND Corporation ein Buch [RAND 55] mit einer Million Zufallsziffern herausgegeben. Die Zahlen für das Buch wurden mit einer elektronischen Roulettescheibe erzeugt, welche von einer zufälligen Frequenzquelle angesteuert wurde. Auch die Benutzung des Buches sollte nach den Verfassern zufällig geschehen. Man sollte das Buch, welches im Hauptteil die Zufallsziffern in tabellarischer Form enthält, zufällig aufschlagen, eine Zahl blind wählen. Von dieser Zahl soll dann die erste Stelle modulo 2 reduziert werden, damit man die Startzeile erhält, und die beiden letzten Stellen der Zahl sollen modulo 50 reduziert werden, um die Startspalte zu erhalten. Weiterhin sollen die bereits für die Berechnung von Startspalte und -zeile benutzten Zahlen markiert werden, damit sie nicht ein zweites Mal als solche verwendet werden.

Wenn man einen Computer benutzt, dann bieten sich auch hier zufällige Ereignisse an, die leicht zu ermitteln sind. Man kann beispielsweise die Tastaturverzögerung beim Bedienen der Tastatur messen, die Bewegung der Maus, die Zugriffszeiten der Festplatte oder die CPU-Auslastung. Diese Messdaten lassen sich dann mit einer Hashfunktion zu einer Zufallszahlensequenz verarbeiten. Weitere Einzelheiten zu dieser Methode sind in [Schneier 96, S. 485ff.] zu finden.

Die beste Möglichkeit, eine große Anzahl von Zufallsbits zu erzeugen, bietet der natürliche Zufall der realen Welt. Hier nutzt man Ereignisse, die regelmäßig, aber zufällig stattfinden, beispielsweise atmosphärisches Rauschen, welches einen bestimmten Schwellenwert überschreitet. Mittels Geigerzähler lässt sich auch der radioaktive Zerfall als Zufallsquelle nutzen. Die Zeit zwischen den Ereignissen dient dann der Erzeugung der Zufallsbits: Ist das nachfolgende Intervall größer als das vorhergehende, dann schreibt man eine 1 in die Zufallszahlensequenz, ansonsten eine 0 (siehe [Schneier 96, S. 484ff.]).

Von dieser Sorte Zufallszahlengeneratoren existiert eine breitere Auswahl. In [Richter 92] wird ein Generator beschrieben, welcher das thermische Rauschen einer Halbleiterdiode nutzt. Auch Spannungsdifferenzen zwischen elektronischen Bauteilen (siehe [Agnew 87, S. 77ff.]) oder die Frequenzschwankungen eines frei schwingenden Oszillators (siehe [Fairfield 84, S. 203ff.]) lassen sich nutzen. Es existieren auch frei käufliche hardwarebasierte Zufallszahlengeneratoren[3].

Allen diesen Methoden zur Erzeugung von Zufallszahlen ist gemein, dass sie sich gar nicht oder nur mit erheblichem Aufwand auf Chipkarten implementieren lassen. Eine weitere Eigenschaft dieser Generatoren macht sie für die in Kapitel 2.1 beschriebene Anwendung unbrauchbar. Diese Generatoren sind nicht reproduzierbar, das heißt es ist nicht möglich, gleiche Zufallszahlen an zwei verschiedenen Orten oder Zeiten zu erzeugen. Ein Abgleich zwischen einer Geldkarte und der dazugehörigen Bank ist damit nicht möglich. Sind diese Generatoren, die nach der Definition in Kapitel 3.1.1 zwar echte Zufallszahlen erzeugen, für kryptographische Anwendungen erwünscht, so sind sie für die Sicherung von Chipkarten unbrauchbar.

Kongruenzen

Die am häufigsten eingesetzte Methode zur Zufallszahlengenerierung bzw. die am häufigsten in Programmiersprachen eingesetzte Methode (siehe [Härtel 94, S. 39]) ist die lineare Kongruenz. Vorgestellt wurde diese Methode in [Lehmer 51] und ist aktuell ausführlich beschrieben in [Knuth 98, S. 10ff.]. Für einen Zufallszahlengenerator wählt man dafür vier Zahlen:

$$\begin{array}{lll} x_0, & \text{den Startwert,} & x_0 > 0. \\ a, & \text{den Multiplikator,} & a > 0. \\ c, & \text{den Inkrementen,} & c \geq 0. \\ m, & \text{den Modulus,} & m > x_0, m > a, m > c. \end{array}$$

Die Folge der Zufallszahlen, die aus der Folge der x_n besteht, erhält man dann mit der wiederholten Lösung der Rekurrenzgleichung:

$$x_{n+1} = (ax_n + c) \bmod m.$$

Die maximal erreichbare Periode ist offensichtlich m. Ein Generator, der lineare Kongruenzen zur Erzeugung der Zufallszahlen benutzt, ist vorhersagbar und damit nicht gut geeignet für den sicheren Einsatz. Diese Aussage gilt auch für abgeschnittene lineare Kongruenzgeneratoren, das heißt lineare Kongruenzgeneratoren, bei denen nur einige Ziffern von x_{n+1} ausgegeben werden, und ebenso für solche mit unbekannten Parametern (siehe [Schneier 96, S. 425]).

Ein weiterer linearer Kongruenzgenerator ist der additive oder auch verzögerte Fibonacci-Generator (siehe [Schneier 96, S. 449]). Die Rekurrenzgleichung für diesen Generator lautet:

$$x_{i+1} = (x_{i-a} + x_{i-b} + ... + x_{i-m}) \bmod 2^n$$

Der Vorteil, den man damit erreicht, dass man die m Vorgänger speichert, ist die maximal erreichbare Periode von $2^n - 1$.

Es ist nahe liegend, den linearen Ausdruck in der Rekurrenzgleichung durch einen nichtlinearen zu ersetzen. Damit kommt man dann zu den nichtlinearen Kongruenzgeneratoren. Ein Vertreter dieser Sorte ist der *Blum-Blum-Shub-Generator*. Dieser wird in Kapitel 5.2 vorgestellt.

[3] `http://valley.interact.nl/av/com/orion/home.html`, `http://www.protego.se/sg100_en.htm`

Rückgekoppelte Schieberegister

Die neben den Kongruenzgeneratoren am häufigsten eingesetzte Methode zur Erzeugung von Zufallszahlen ist das rückgekoppelte Schieberegister. Die ersten theoretischen Betrachtungen finden sich in [Selmer 66]. Man unterscheidet je nach der Rückkopplungsfunktion zwischen linear rückgekoppelten und nicht linear rückgekoppelten Schieberegistern. Das linear rückgekoppelte Schieberegister („Linear Feedback Shift Register"), hier als LFSR abgekürzt, wird in Kapitel 5.1 ausführlich vorgestellt.

Nutzung von Verschlüsselungsalgorithmen

Generell lassen sich alle Algorithmen, die der Verschlüsselung dienen, zur Erzeugung von Zufallszahlen nutzen. Primäres Ziel eines Verschlüsselungsalgorithmus' ist es, die Redundanzen, die ein Text enthält, vollkommen zu beseitigen, so dass ein Text mit zufälligem Inhalt entsteht. Von diesem so erzeugten Chiffretext ist es dann nicht mehr möglich, auf den Klartext zu schließen. Die Stromchiffren sind am einfachsten für die Erzeugung von Zufallszahlen zu benutzen, da diese Algorithmen bereits eine Zufallszahlensequenz erzeugen (siehe Kapitel 2.4). Diese Methode der Erzeugung von Zufallszahlen wird in den Kapiteln 5.3 und 6.4 am Beispiel von RC4 untersucht. Andere Verschlüsselungsalgorithmen kann man ebenso benutzen, indem man einfach für den Klartext einen beliebigen unbekannten Text benutzt. Die Auslagerungsdatei des Betriebssystems oder temporäre Programmdateien bieten sich zum Verschlüsseln an. Der damit erzeugte Chiffretext lässt sich dann als Quelle für die Zufallszahlen benutzen. Es besteht auch die Möglichkeit, ein Startwort zu verschlüsseln und das verschlüsselte Wort erneut zu verschlüsseln und dies immer wieder zu tun. Die damit rekursiv erzeugten verschlüsselten Wörter lassen sich als Zufallszahlensequenz benutzen.

Mathematische Probleme

Es besteht eine allgemeine Möglichkeit, aus mathematischen Problemen einen Zufallszahlengenerator zu erzeugen. Unter einem mathematischen Problem wird hier die Existenz einer Einwegfunktion („One-Way-Function") verstanden, das heißt es existiert eine effizient berechenbare Funktion $f(x)$, für die die Umkehrfunktion einen sehr hohen Rechenaufwand erfordert.

Prominentestes Beispiel hierfür ist das Problem der Faktorisierung von ganzen Zahlen, welches in dem RSA-Algorithmus („Rivest, Shamir, Adleman", [RSA 78, S. 120ff.]) eingesetzt wird. Es ist einfach, zwei große Primzahlen zu multiplizieren, aber erheblich aufwändiger, das Produkt der Primzahlen bei deren Unkenntnis zu faktorisieren. Solche Einwegfunktionen ermöglichen es, Verschlüsselungsalgorithmen und damit auch Zufallszahlengeneratoren zu entwickeln, deren Sicherheit vom Berechnungsaufwand der Umkehrfunktion abhängt. Weitere Beispiele sind die Berechnung des diskreten Logarithmus ([Odlyzko 84, S. 224ff.]; siehe auch Kap. 2.3) in der mutiplikativen Gruppe der Primkörper $\mathcal{GF}(p)$, in der multiplikativen Gruppe der Charakteristik 2, also $\mathcal{GF}(2^n)$ oder in der Gruppe der elliptischen Kurven über endlichen Körpern $\mathcal{EC}(F)$. Der diskrete Logarithmus ist in allen Fällen die Berechnung der Umkehrung der modularen Potenzierung (siehe 2.3, Abschnitt Schlüsselaustausch).

3.2 Mathematische Grundlagen

3.2.1 Gruppentheorie

Im nachfolgenden Abschnitt sollen die gruppentheoretischen Grundlagen kurz vorgestellt werden. Auf Beweise wird dabei verzichtet, diese finden sich in [Biggs 89, S. 271ff.].

Eine *Gruppe* ist eine algebraische Struktur, die über eine Menge G definiert wird. Weiterhin wird eine binäre Operation $\circ$ benötigt, die zwei Elemente aus G verknüpft.

Definition: Das Paar $(G, \circ)$ heißt genau dann *Gruppe*, wenn folgende Axiome erfüllt werden:

$G1$	Abgeschlossenheit	$\forall x, y \in G$:		$x \circ y \in G$
$G2$	Assoziativität	$\forall x, y, z \in G$:		$(x \circ y) \circ z = x \circ (y \circ z)$
$G3$	Neutrales Element	$\exists e \in G$:	$\forall x \in G$:	$e \circ x = x \circ e = x$
$G4$	Inverses Element	$\forall x \in G$:	$\exists x' \in G$:	$x \circ x' = x' \circ x = e$

Für die in dieser Arbeit vorgestellten Zufallszahlengeneratoren LFSR (siehe Kapitel 5.1) und EC (siehe Kapitel 5.4) ist die Gruppe das zugrunde liegende mathematische Modell. Für die elliptischen Kurven $\mathcal{EC}(F)$ über $\mathcal{GF}(p)$ wird das im nächsten Abschnitt erläutert werden.

Definition: Die Anzahl der Elemente einer Gruppe $(G, \circ)$ wird als *Ordnung* der Gruppe $(G, \circ)$ bezeichnet und mit $\mid G \mid$ notiert. Ist $\mid G \mid$ endlich, dann liegt eine *endliche* Gruppe vor.

Definition: Sei $x \in G$, dann ist die Potenz x^n wie folgt rekursiv definiert:

$$x^0 = e, \qquad x^1 = x, \qquad x^n = x \circ x^{n-1} \qquad (\text{mit} \quad n \geq 2)$$

Für negative Exponenten n lautet die Definition analog, dabei wird für das in Axiom $G4$ eingeführte inverse Element x' auch x^{-1} geschrieben.

Definition: Sei $x \in G$ und $\mid G \mid$ endlich, dann ist die kleinste positive ganze Zahl m mit $x^m = e$ die *Ordnung* von x in $(G, \circ)$.

Definition: Eine Gruppe $(G, \circ)$ wird als *zyklische Gruppe* bezeichnet, wenn es ein $x \in G$ gibt, so dass jedes Element von G eine Potenz von x ist, also $G = \{x^n \mid n \in \mathbb{Z}\}$. Das Element x wird als *erzeugendes Element* bezeichnet, und man schreibt $G = \langle x \rangle$.

Definition: Eine Teilmenge H der Elemente der Gruppe $(G, \circ)$ bildet eine Untergruppe von $(G, \circ)$, wenn für H die Gruppenaxiome bezüglich der in G definierten Verknüpfung $\circ$ gelten.

Theorem: Wenn $(G, \circ)$ eine endliche Gruppe der Ordnung m ist und $(H, \circ)$ ist eine Untergruppe von $(G, \circ)$ mit der Ordnung n, dann ist n ein Teiler von m.

Dieser Satz ist als Theorem von Lagrange bekannt, und er ist hier sehr wichtig für die Beurteilung der Periode der von den Generatoren erzeugten Zufallszahlensequenzen. Weiterhin gilt, wenn $(G, \circ)$ eine zyklische Gruppe ist, dann gibt es für jeden Teiler d von $\mid G \mid$ genau eine Untergruppe.

Definition: Sei in einer Sequenz x_i das i-te Zeichen, dann besitzt die Sequenz genau dann eine *Periode* der Länge m, wenn ab einem Wert x_i für alle x_j, mit $j \geq i$, $x_j = x_{j+m-1}$ ist, dabei sind m und i die kleinsten positiven ganzen Zahlen, die diese Bedingung erfüllen. Die Sequenz $x_1 x_2 \ldots x_{i-1}$ wird dann als *Vorperiode* bezeichnet.

Da die hier vorgestellten Zufallszahlengeneratoren deterministisch arbeiten und nur endlich viele Werte verarbeiten, erzeugt jeder der Generatoren Sequenzen mit Perioden. Diese sollten am besten so groß wie möglich sein, um die Vorhersagbarkeit der nachfolgenden Bits zu erschweren. Wenn nun für die Generierung der Zufallszahlen endliche Gruppen zugrunde liegen, dann muss man, um eine große Periode zu erreichen, auf die Ordnung der Gruppe achten. Ist die Gruppe primer Ordnung, dann gelangt man erst dann in einen schon durchlaufenen Zustand und damit an die Stelle, ab der sich die Bits wiederholen, wenn so viele Verknüpfungsschritte ausgeführt worden sind, wie die Ordnung der Gruppe ist. Ist die Gruppe nicht primer Ordnung, so kann es passieren, dass man in einer Untergruppe rechnet und diese dann eine sehr kleine Ordnung, das heißt eine kurze Periode, besitzt. Wegen der Abgeschlossenheit bleibt man, einmal dorthin gelangt, bei den Verknüpfungsschritten in der entsprechenden Untergruppe.

3.2.2 Elliptische Kurven

In diesem Kapitel werden die wichtigsten Grundlagen für elliptische Kurven $\mathcal{EC}$ über endlichen Körpern F, also $\mathcal{EC}(F)$, und hier in dem speziellen Fall des Körpers $\mathcal{GF}(p)$ vorgestellt. Diese Einschränkungen beziehen sich auf den in Kapitel 5.4 vorgestellten Zufallszahlengenerator. Ausführliche und weitergehende Darstellungen zu allen in diesem Kapitel angesprochenen Bereichen finden sich in [IEEE P1363 / D13].

Bevor die elliptischen Kurven definiert werden, sind noch einige Definitionen zum Verständnis notwendig.

Körper

Definition: Eine Gruppe $(G, \circ)$ heißt *abelsche* oder *kommutative* Gruppe, wenn zusätzlich zu den vier Gruppenaxiomen Folgendes erfüllt ist:

$$G5 \quad \text{Kommutativität} \quad \forall x, y \in G : \quad x \circ y = y \circ x$$

Definition: Es seien zwei kommutative Gruppen $(G_1, \circ)$ und $(G_2, \oplus)$ gegeben und die neutralen Elemente dieser Gruppen mit e_1 bzw. e_2 bezeichnet. Wenn $G_1 = G_2$ ist, die Verknüpfungen $\circ$ und $\oplus$ sich unterscheiden und für die Gruppe $(G_1, \circ)$ bei Axiom $G4$ für $x \circ x' = e_1$ einschränkend $x \neq e_2$ vorausgesetzt wird, dann ist das Tripel $(K, \circ, \oplus)$ ein *Körper*, wenn $G_1 = K$ ist und zusätzlich folgendes Axiom erfüllt ist:

$$K1 \quad \text{Distributivität} \quad \forall x, y, z \in K : \quad (x \oplus y) \circ z = (x \circ z) \oplus (y \circ z)$$

Ist $\mid K \mid$ ein endlicher Wert, dann bezeichnet man $(K, \circ, \oplus)$ als *endlichen* Körper.

Der Körper $\mathcal{GF}(p)$

Die elliptischen Kurven, die für den vorgestellten Zufallszahlengenerator benutzt werden, sind über dem endlichen Körper $\mathcal{GF}(p)$ definiert. Das $\mathcal{GF}$ steht für Galois-Feld und das p für eine Primzahl $p > 2$. Die Elemente des Körpers sind dann die Menge $\mathbb{Z}_p = \{0, 1, 2, \ldots, p-1\}$. Die Verknüpfungen $\circ$ und $\oplus$ entsprechen dann der Multiplikation $\cdot$ bzw. der Addition $+$ modulo p. Das bedeutet, dass jedes Ergebnis aus $x \cdot y = z$ bzw. $x + y = z$ mit $z > p - 1$ wegen des Abgeschlossenheitsaxioms $G1$ durch Reduktion mit dem Modulus p in ein Element aus $\mathbb{Z}_p$ umgewandelt werden muss.

Definition: Zwei Zahlen x und y sind kongruent modulo p, geschrieben $x \equiv y \bmod p$, wenn p die Differenz $x - y$ teilt.

Addiert man beispielsweise im Körper $\mathcal{GF}(5)$ die Zahlen 3 und 4, dann liegt bei der gewöhnlichen Addition das Ergebnis 7 nicht in der Menge $\mathbb{Z}_5 = \{0, 1, 2, 3, 4\}$, also wird das Ergebnis 7 mit dem Modulus 5 zu 2 reduziert, da $7 \equiv 2 \bmod 5$ ist.

Die Division im Körper $\mathcal{GF}(p)$ lässt sich auf die Multiplikation mit dem Inversen des Divisors zurückführen, da der Quotient $\frac{x}{y}$ sich als $x \cdot y'$ schreiben und dann als Produkt ausrechnen lässt. Das heißt die Division lässt sich auf die Multiplikation zurückführen, wenn das Inverse des Divisors bekannt ist.

Ist beispielsweise der Quotient $\frac{4}{3}$ in $\mathcal{GF}(5)$ gesucht, so muss die 4 mit dem inversen Element multipliziert werden, das heißt $\frac{4}{3} = 4 \cdot 3^{-1}$. Das inverse Element der 3 ist 2, da $3 \cdot 2 \equiv 1 \bmod 5$ ist. Für den Quotienten ergibt sich also $\frac{4}{3} = 4 \cdot 3^{-1} = 4 \cdot 2 = 3$, denn $4 \cdot 2 \equiv 3 \bmod 5$. Das Finden des inversen Elementes in einem endlichen Körper ist im Allgemeinen keine einfache Aufgabe und sollte nach Möglichkeit vermieden werden.

Elliptische Kurven über $\mathcal{GF}(p)$

Elliptische Kurven sind keine, wie der Name vermuten lässt, Kurven, die Ellipsen beschreiben, sondern Gleichungen, die bei der Berechnung von Ellipsenumfängen im Integranden auftreten.

Definition: Eine elliptische Kurve $\mathcal{EC}$ über $\mathcal{GF}(p)$ ist eine Menge von Punkten (x, y), die die Gleichung $y^2 = x^3 + ax + b$ erfüllen. Weiterhin gilt $4a^3 + 27b^2 \neq 0$ und $a, b, x, y, \in \mathbb{Z}_p$. Zusätzlich wird noch der so genannte Unendlichkeitspunkt $\mathcal{O}$ hinzugenommen.

Als Beispiel für eine elliptische Kurve (siehe Abb. 3) hat $y^2 = x^3 + 2x + 4$ über $\mathcal{GF}(5)$ die folgenden Punkte $\{(0, 2), (0, 3), (2, 1), (2, 4), (4, 1), (4, 4), \mathcal{O}\}$.

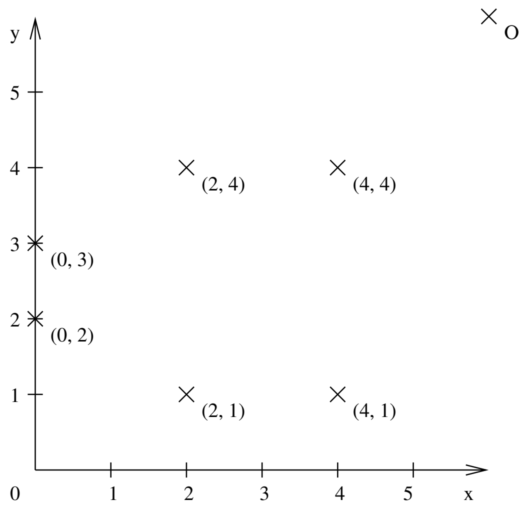

Abbildung 3: Die elliptische Kurve $y^2 = x^3 + 2x + 4$. Der Punkt $\mathcal{O}$ rechts oben ist die Darstellung des Unendlichkeitspunktes.

Elliptische Kurven als Gruppe

Das Besondere an den elliptischen Kurven ist, dass sie bezüglich der Addition von Punkten eine Gruppe bilden. Die Addition zweier Punkte $P_1(x_1, y_1)$ und $P_2(x_2, y_2)$ einer elliptischen Kurve ergibt einen dritten Punkt $P_3(x_3, y_3)$ und ist wie folgt definiert:

$$\begin{aligned} -P_1 &= (x_1, -y_1) \\ P_3 = P_1 + P_2 &= \begin{cases} \mathcal{O}, & \text{falls} \quad P_1 = -P_2 \\ P_2, & \text{falls} \quad P_1 = \mathcal{O} \\ P_1, & \text{falls} \quad P_2 = \mathcal{O}. \end{cases} \end{aligned}$$

Ansonsten ist $P_3(x_3, y_3) = P_1 + P_2$ mit

$$\begin{aligned} \lambda &= \begin{cases} \frac{y_1 - y_2}{x_1 - x_2}, & \text{falls} \quad P_1 \neq P_2 \\ \\ \frac{3x_1^2 + a}{2y_1}, & \text{falls} \quad P_1 = P_2 \end{cases} \\ x_3 &= -x_1 - x_2 + \lambda^2 \\ y_3 &= -y_1 + \lambda(x_1 - x_3). \end{aligned}$$

Wenn zwei Punkte P_1 und P_2 auf der Kurve liegen, dann liegt nach dem Abgeschlossenheitsaxiom $G1$ auch die Summe der Punkte, also der Punkt P_3, auf der Kurve.

Für die Fälle $P_1 = -P_2$, $P_1 = \mathcal{O}$ und $P_2 = \mathcal{O}$ ist das offensichtlich. Die weiteren Fälle mit $P_1 \neq P_2$ und $P_1 = P_2$ erfordern etwas längere Rechnungen, auf die hier verzichtet wird.

Als Beispiel wird hier der Punkt $P_1(2,1)$ aus unserer Beispielkurve $y^2 = x^3 + 2x + 4$ verdoppelt, das heißt der Punkt wird zu sich selbst addiert, also der Fall $P_1 = P_2$. Nach der Additionsdefinition ergeben sich für $P_1 + P_1 = P_2$ folgende Werte [4]:

$$x_2 = -2 - 2 + \left(\frac{3 \cdot 2^2 + 2}{2 \cdot 1}\right)^2 = -4 + \left(\frac{14}{2}\right)^2 = -4 + (14 \cdot 3)^2 = 1760 = 0$$

$$y_2 = -1 + \left(\frac{3 \cdot 2^2 + 2}{2 \cdot 1}\right)(2 - x_2) = -1 + (14 \cdot 3) \cdot 2 = 43 = 3$$

Die fortlaufende Addition, also die Vervielfachung, ist in Abb. 4 dargestellt.

Der Punkt $P_2(0,3)$ liegt auch auf unserer Beispielkurve (siehe Abb. 3 und Abb 4).

Die Addition von Kurvenpunkten folgt auch dem Assoziativitätsaxiom $G2$. Das neutrale Element nach Axiom $G3$ ist der Unendlichkeitspunkt $\mathcal{O}$, da die Addition von einem Punkt P_1 mit $\mathcal{O}$ per Definition wieder den Punkt P_1 ergibt.
Das inverse Element zu einem Punkt $P_1(x_1, y_1)$ ist $P_2(x_1, -y_1)$, da die Addition von den Punkten P_1 und $-P_1$ den Unendlichkeitspunkt $\mathcal{O}$ ergibt. Zu zeigen, dass die Gruppe kommutativ ist, erfordert längere Rechnungen, auf die hier verzichtet wird.

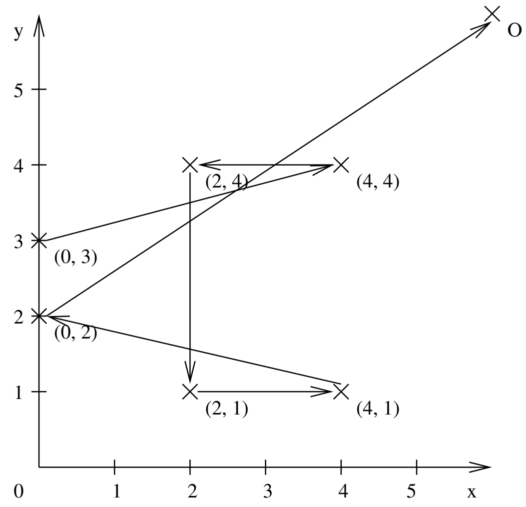

Abbildung 4: Die elliptische Kurve $y^2 = x^3 + 2x + 4$ wie in Abb. 3. Die Pfeile zeigen den Zyklus des Punktes (0, 3), das heißt wenn der Punkt vervielfacht wird, ergeben sich die Punkte in der Reihenfolge, wie sie mit den Pfeilen verbunden sind.

Projektive Koordinatendarstellung

Bisher wurden die Punkte der elliptischen Kurven mit 2 Koordinaten in der so genannten affinen Form angegeben, ein Punkt hatte also die Form (x, y) und konnte grafisch in einem zweidimensionalen Koordinatensystem veranschaulicht werden. Bei der Addition von Punkten einer elliptischen Kurve ist es in dieser Darstellung notwendig, im zugrunde liegenden Körper, hier $\mathcal{GF}(p)$, zu dividieren, und damit ist es erforderlich, das Inverse eines Elementes aus $\mathcal{GF}(p)$ zu berechnen. Dies lässt sich vermeiden, wenn man zur projektiven Koordinatendarstellung übergeht.

Im Folgenden werden zur Unterscheidung der Darstellungen für die affine kleine und für die projektive Koordinatendarstellung große Buchstaben benutzt. Bei der projektiven Koordinatendarstellung ist eine dritte Variable Z notwendig. Der Punkt (x, y) bezeichnet einen Punkt in der affinen und der Punkt (X, Y, Z) einen in der projektiven Darstellung. Die projektive Darstellung ist nicht eindeutig, so bezeichnet jeder Punkt der Form $(\lambda^2 X,\ \lambda^3 Y,\ \lambda Z)$ mit $\lambda \neq 0,\ \lambda \in \mathcal{GF}(p)$ ein und denselben Punkt. Damit lässt sich in der projektiven Darstellung nichts mehr einfach grafisch veranschaulichen.

Die Umrechnung von der affinen zur projektiven Darstellung erfolgt mit:

$$X = x, \qquad Y = y, \qquad Z = 1.$$

Die Umrechnung von der projektiven zur affinen Darstellung ist schwieriger und erfolgt mit:

$$x = \frac{X}{Z^2}, \qquad y = \frac{Y}{Z^3}.$$

[4] Beachte, dass in $\mathcal{GF}(5)$ gerechnet wird.

Der Unendlichkeitspunkt $\mathcal{O}$ hat die Form $(\lambda^2, \lambda^3, 0)$ mit $\lambda \neq 0, \lambda \in \mathcal{GF}(p)$. Es existieren weitere projektive Koordinatendarstellungen, die allerdings mit einem höheren Rechenaufwand verbunden sind. Der entscheidende Vorteil der projektiven Darstellung ist das Vermeiden der Division in $\mathcal{GF}(p)$.

Im Folgenden wird die Punktaddition von einem Punkt mit sich selbst dargestellt, da diese Operation bei dem in dieser Arbeit vorgestellten EC-Zufallszahlengenerator (siehe Kapitel 5.4) benutzt wird. Die anderen Fälle finden sich in [IEEE P1363 / D13, S. 126ff.].

Gegeben ist ein Punkt $P(X_1, Y_1, Z_1)$, und gesucht ist der Punkt $Q(X_2, Y_2, Z_2)$ mit $Q = 2P$. Zu berechnen ist in diesem Fall:

$$\begin{aligned} X_2 &= M^2 - 2S, \\ Y_2 &= M(S - X_2) - T, \\ Z_2 &= 2Y_1 Z_1. \end{aligned}$$

Die drei Hilfsvariablen M, S und T erhält man wie folgt:

$$\begin{aligned} M &= 3X_1^2 + aZ_1^4, \\ S &= 4X_1 Y_1^2, \\ T &= 8Y_1^4. \end{aligned}$$

Das a wird der gegebenen elliptischen Kurve der Form $y^2 = x^3 + ax + b$ entnommen, und alle Berechnungen werden im Körper $\mathcal{GF}(p)$ durchgeführt, über dem die Kurve definiert ist.

3.3 Statistik

Für die Bewertung der Qualität von Zufallszahlen (siehe Kapitel 4) werden Verfahren aus der Statistik verwendet. Das sind zum einen der Chi-Quadrat-Anpassungstest und zum anderen die Intervallschätzung einer Normalverteilung. Beide Verfahren werden in diesem Abschnitt kurz vorgestellt, allerdings ohne auf die statistischen Herleitungen näher einzugehen. Für detaillierte Darstellungen muss auf [Hübner 96, S. 175ff.] und [Härtel 94, S. 12ff.] verwiesen werden.

Chi-Quadrat-Anpassungstest

Mit einem Anpassungstest wird eine unbekannte Wahrscheinlichkeitsverteilung dahingehend überprüft, ob sie gleich einer vermuteten, also hypothetischen Verteilung ist. Ein solcher Anpassungstest ist der *Chi-Quadrat-Anpassungstest* (χ^2-Anpassungstest), der nun in einer vereinfachten Weise vorgestellt werden soll (angelehnt an [Härtel 94, S.16.]).

Aus der Grundgesamtheit der Größe N werde eine Stichprobe $x_1, x_2, \ldots, x_n$ vom Umfang n gezogen. Die unbekannte Wahrscheinlichkeitsverteilung der Stichprobe sei F, und die hypothetische Verteilung sei F_0. Die zu prüfende Hypothese lautet dann $\mathcal{H}_0 : \quad F = F_0$.

Der Test wird in folgenden Schritten durchgeführt:

1. Zunächst wird das Signifikanzniveau α festgelegt[5], also die Wahrscheinlichkeit, mit der die Hypothese $\mathcal{H}_0$ gelten soll, beispielsweise $\alpha = 0,95 = 95\%$. Weiterhin muss die Anzahl k der Werte festgelegt sein, die die Werte der Stichprobe annehmen kann. k heißt auch *Anzahl der Freiheitsgrade*. Es muss gelten: $k \geq 1$ und k ist endlich, denn der Chi-Quadrat-Test ist nur zum Testen diskreter Verteilungen

[5] In der Literatur ist die Bezeichnung des Signifikanzniveaus unterschiedlich; so bezeichnet es beispielsweise [Hübner 96] mit α, aber [Härtel 94] bezeichnet es mit $1 - \alpha$. Hier wird also die Benennung nach [Hübner 96] verwendet.

geeignet. Mit $z_1, z_2, \ldots, z_k$ seien die k Werte bezeichnet, die die Stichprobe annehmen kann.
Falls die Stichprobe $x_1, x_2, \ldots, x_n$ beispielsweise nur die Werte 0 und 1 annehmen kann, so folgen $k = 2$, weil nur 2 Werte möglich sind, $z_1 = 0$ und $z_2 = 1$.

2. Für jeden Wert z_i, $1 \le i \le k$, wird gezählt, wie oft ein Stichprobenwert $x_1, x_2, \ldots, x_n$ den Wert z_i annimmt. Die Anzahl wird als Häufigkeit H_i bezeichnet.

3. Für jeden Wert z_i ($1 \le i \le k$) wird die Wahrscheinlichkeit $p_i = P(X = z_i | F_0)$ berechnet, also die Wahrscheinlichkeit, mit der ein Stichprobenwert nach der hypothetischen Verteilung F_0 den Wert z_i annimmt.

4. Es muss die Testgröße χ^2 berechnet werden mit der Formel

$$\chi^2 = \sum_{i=1}^{k} \frac{1}{n \cdot p_i} (H_i - n \cdot p_i)^2. \tag{1}$$

   Dabei ist $(H_i - n \cdot p_i)^2$ die quadratische Abweichung der Anzahl an Stichprobenwerten, die z_i annehmen, von der mit der hypothetischen Verteilung F_0 berechneten Anzahl $n \cdot p_i$. Die Division durch $n \cdot p_i$ bewirkt, dass die quadratische Abweichung normiert wird, also unabhängig von der hypothetischen Verteilung wird. Für die Begründung der Wahl der Testgröße siehe [Hübner 96, S. 188].

5. Aus einer Tabelle für die χ^2-Verteilung muss das α-Quantil c_α der χ^2-Verteilung abgelesen werden. Solch eine Tabelle ist beispielsweise in [Hübner 96, S. 197] zu finden. Die Tabellenspalten geben die α-Quantile an, also das Signifikanzniveau, und in den Zeilen steht der Freiheitsgrad.

   Der Ablehnungsbereich der Hypothese $\mathcal{H}_0$ ist das Intervall $K_\alpha = (c_\alpha, \infty)$, denn es gilt folgender Satz:

$$\text{Die Hypothese } \mathcal{H}_0 \text{ wird genau dann abgelehnt, wenn } \chi^2 \in K_\alpha.$$

   Das α-Quantil muss also mit dem oben berechneten χ^2 verglichen werden, und wenn es größer als χ^2 ist, weicht die Stichprobe von der vermuteten Verteilung F_0 ab.

Zur Verdeutlichung des Verfahrens folgt ein Beispiel: Ein Würfel soll dahin gehend geprüft werden, ob alle Augenzahlen mit gleicher Wahrscheinlichkeit auftreten. Der Würfel wird $n = 40$ Mal geworfen (das ist vielleicht nicht sinnvoll, aber die auftretenden Zahlen sollen überschaubar klein bleiben), und die Häufigkeiten H_i der erzielten Augenzahlen werden gezählt. Dabei soll das Ergebnis wie in Tabelle 1 aufgetreten sein.

i	z_i	H_i	p_i
1	1	7	$\frac{1}{6}$
2	2	7	$\frac{1}{6}$
3	3	7	$\frac{1}{6}$
4	4	7	$\frac{1}{6}$
5	5	6	$\frac{1}{6}$
6	6	6	$\frac{1}{6}$

Tabelle 1: Beispiel eines Zufallsexperiments: Häufigkeiten der aufgetretenen Augenzahlen bei 40 Würfen mit einem Würfel.

Die zu prüfende Hypothese ist

$$\mathcal{H}_0: \quad P(X = k) = \frac{1}{6},$$

also die Hypothese der Gleichverteilung. Die Wahrscheinlichkeiten, dass die gewürfelte Augenzahl X gleich k wird, sind ebenfalls in Tabelle 1 eingetragen. Das Signifikanzniveau sei $\alpha = 0,95$, das heißt das Ergebnis des Tests soll mit einer Wahrscheinlichkeit von 95% zutreffen.

Es ist $k = 6$, weil es 6 verschiedene Versuchsausgänge gibt, und die Werte, die gewürfelt werden können, sind $z_1 = 1$, $z_2 = 2$, $z_3 = 3$, $z_4 = 4$, $z_5 = 5$ und $z_6 = 6$.

Nun kann χ^2 nach Formel 1 berechnet werden:

$$\chi^2 = \sum_{i=1}^{6} \frac{1}{40 \cdot \frac{1}{6}} \left(H_i - 40 \cdot \frac{1}{6}\right)^2 = \frac{6}{40} \cdot \left(4 \cdot \left(7 - \frac{40}{6}\right)^2 + 2 \cdot \left(6 - \frac{40}{6}\right)^2\right) = \frac{6}{40} \cdot \left(\frac{4}{9} + \frac{8}{9}\right) = \frac{1}{5}.$$

Aus einer Tabelle für die Quantile der χ^2-Verteilung, beispielsweise aus [Hübner 96, S. 188], entnimmt man für $k = 6$ Freiheitsgrade das Quantil $c_\alpha = c_{0,95} = 12,60$. Da $\frac{1}{5} < 12,60$, wird die Hypothese $\mathcal{H}_0$ angenommen, das heißt die Verteilung ist annähernd eine Gleichverteilung.

Intervallschätzung für die Normalverteilung

In diesem Abschnitt wird eine Stichprobe $x_1, x_2, \ldots, x_n$, die in der Theorie normalverteilt sein müsste, durch eine Normalverteilung angenähert. Dazu werden der Erwartungswert μ und die Streuung σ der zu bestimmenden Normalverteilung durch das Stichprobenmittel bzw. die empirische Stichprobenstandardabweichung der Stichprobe approximiert. Für den Erwartungswert μ wird ein Vertrauensintervall, auch Konfidenzintervall genannt, geschätzt, in dem der Wert mit einer vorgegebenen Wahrscheinlichkeit liegen soll. Die Schätzung wird in folgenden Schritten durchgeführt (angelehnt an [Hübner 96, S. 182]):

1. Festlegen der Signifikanzniveaus α, also der Wahrscheinlichkeit, mit der der Erwartungswert μ im Vertrauensintervall liegen soll.

2. Berechnen des Stichprobenmittels $\overline{X}$ der Stichprobe $x_1, x_2, \ldots, x_n$ mit der Formel
$$\overline{X} = \frac{1}{n} \cdot \sum_{i=1}^{n} X_i. \tag{2}$$
Das Stichprobenmittel $\overline{X}$ ist eine Schätzung für den unbekannten Erwartungswert μ der Grundgesamtheit, und es ist zudem sogar die beste aller möglichen Schätzungen.

3. Berechnen der empirischen Stichprobenstandardabweichung σ mit der Formel
$$\sigma_S = \sqrt{\frac{1}{n-1} \cdot \sum_{i=1}^{n} (x_i - \overline{X})^2}. \tag{3}$$
Die empirische Stichprobenstandardabweichung σ_S ist die beste Schätzung für die Standardabweichung σ der Grundgesamtheit. Sie gibt an, um welchen Betrag die Werte der Grundgesamtheit durchschnittlich vom Stichprobenmittel abweichen.

4. Nun kann das reellwertige, geschlossene Konfidenzintervall K des Stichprobenmittels $\overline{X}$ berechnet werden. Es ergibt sich zu
$$K = \left[\overline{X} - t_{n-1,\,1+\frac{\alpha}{2}} \cdot \frac{\sigma}{\sqrt{n}} \quad ; \quad \overline{X} + t_{n-1,\,1+\frac{\alpha}{2}} \cdot \frac{\sigma}{\sqrt{n}}\right].$$
Dabei bedeutet n wiederum der Umfang der Stichprobe, und $t_{n-1,\,1+\frac{\alpha}{2}}$ ist das $1 + \frac{\alpha}{2}$-Quantil der Student-Verteilung mit $n - 1$ Freiheitsgraden. Für die Definition der Student-Verteilung wird auf [Hübner 96, S. 181f.] verwiesen. Da bei der Auswertung von Zufallszahlen umfangreiche Stichproben auftreten, kann näherungsweise $n - 1 \to \infty$ angenommen und mit $t_{\infty,\,1+\frac{\alpha}{2}}$ gerechnet werden. Die Quantile der Student-Verteilung sind tabelliert und beispielsweise in [Hübner 96, S. 196] zu finden.

Die Dichtefunktion der resultierenden Normalverteilung mit den eben berechneten Parametern $\mu = \overline{X}$ und $\sigma = \sigma_S$ ist
$$P(X = x_i) = \frac{1}{\sqrt{2\pi} \cdot \sigma} \cdot e^{-\frac{1}{2} \cdot (\frac{x_i - \mu}{\sigma})^2},$$
wobei $\overline{X}$ mit der Wahrscheinlichkeit α im oben berechneten Vertrauensintervall liegt.

3.4 Chipkarten

Da in den späteren Kapiteln Zufallszahlengeneratoren für Chipkarten vorgestellt und implementiert werden, soll im Folgenden auf die Technik von Chipkarten eingegangen werden.

Im alltäglichen Gebrauch werden verschiedene Arten von Identifikationskarten benutzt, die sich im Aussehen, im Aufbau und im Verwendungszweck unterscheiden. Allen gemeinsam ist jedoch, dass sie der ISO-Norm 7810 („Identification Cards – Physical Characteristics“) entsprechen, die die physikalischen Eigenschaften wie Abmessungen oder Werkstoffe festlegt.

Eine Art der Identifikationskarten ist die *hochgeprägte Karte*, die sich dadurch auszeichnet, dass die Informationen, die sie bereitstellt, als in Plastik gegossene Zeichen auf der Kartenoberfläche fest angebracht sind. Mit einem Lesegerät können ein Abdruck dieser Zeichen erstellt und die Informationen der Karte kopiert werden. Ein Beispiel für eine hochgeprägte Karte ist die verbreitete EuroCard, auf der die Kartennummer aufgeprägt ist.

Neben den hochgeprägten Karten gibt es auch *Magnetstreifenkarten*, auf denen die Information in digitaler Form auf einem Magnetstreifen, der sich auf der Kartenrückseite befindet, gespeichert sind (Details siehe [Rankl 99, S.44ff.]). Zum Beispiel sind Eurocheque-Karten Magnetstreifenkarten.

Die neueste Art der Identifikationskarte ist die *Chipkarte* (Smart Card), in die ein Mikrochip eingebaut ist. Solche Karten sollen in den folgenden Abschnitten beschrieben werden (angelehnt an [Rankl 99]).

3.4.1 Überblick über Chipkarten-Typen

Chipkarten sind dadurch charakterisiert, dass in ihnen ein Mikrochip, der einen integrierten Schaltkreis enthält, untergebracht ist. Der Chip verfügt über Funktionseinheiten zum Speichern, Verarbeiten und Übertragen von Daten, ist also ein eigenständiges Rechensystem.

Es gibt mehrere Gründe dafür, Chipkarten zu benutzen:

- Chipkarten bieten viel Platz zum Speichern von Informationen. Während auf einer hochgeprägten Karte nur einige Byte, das heißt einige Zeichen, und auf einer Magnetstreifenkarte etwa 1000 Bit (Quelle: [Rankl 99, S. 44f.]) gespeichert werden können, sind auf einer Chipkarte mehrere Kilobyte üblich. Es gibt bereits Karten mit mehr als 32 KByte Speicher. Nicht für jede Anwendung wird so viel Speicher benötigt, aber für sicherheitsrelevante Fälle, z.B. die Speicherung eines Datenschlüssels, sind einige KByte nötig.

- Die auf der Chipkarte gespeicherten Daten sind geschützt. Dieser Schutz gilt sowohl gegen das Auslesen der Daten als auch gegen ihre Manipulation. Es ist möglich, geheime Daten in den Mikrochip zu laden, die von außen nicht mehr gelesen werden können, sondern nur noch im Rechenwerk des Chips verarbeitet werden können. Ermöglicht wird dieser Schutz durch eine Vielzahl von Sicherheitsmaßnahmen (Details siehe [Rankl 99, S. 469ff.]).

- Chipkarten können Berechnungen durchführen. Während hochgeprägte Karten und Magnetstreifenkarten lediglich zum Speichern von Informationen dienen, kann eine Chipkarte Rechenoperationen durchführen. Die Berechnungen können von gespeicherten Daten auf der Chipkarte und von äußeren Eingaben abhängen. Die Ergebisse können gespeichert oder nach außen abgegeben werden.

  Dadurch eröffnet sich eine Vielzahl von Anwendungsgebieten, beispielsweise die Verwendung von Authentifikationsalgorithmen (siehe Kapitel 2.3) mithilfe einer Chipkarte.

- Chipkarten sind günstig in der Herstellung. Als Werkstoff für den Kartenkörper wird meist Polyvinylchlorid (PVC) benutzt, das sehr preiswert ist (siehe [Rankl 99, S.70]). In der Massenproduktion ist der Mikrochip ebenfalls günstig herzustellen, und die Montage ist einfach und kostengünstig durchführbar. Teuer ist allerdings der Eigenentwurf einer Chipkarte.

Speicherkarten

Speicherkarten dienen dem Abspeichern von Informationen, die vor Manipulation und Löschung geschützt sein sollen. In Abb. 5 ist schematisch der Aufbau einer Speicherkarte skizziert.

Die Anwendungsdaten sind in einem EEPROM („Electrically Erasable Read Only Memory") abgelegt, das heißt sie sind in einem wiederbeschreibbaren, nichtflüchtigen Speicher abgelegt und können demnach geändert werden. Die Karte besitzt weiterhin einen Nur-Lese-Speicher („Read-Only Memory"), in dem die Identifizierungsdaten der Karte abgelegt sind, die nicht geändert werden können. Der Zugriff auf den Speicher wird hardwaremäßig durch die Sicherheitslogik kontrolliert.

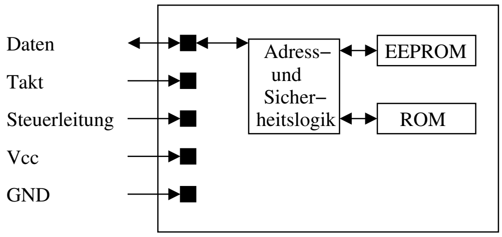

Abbildung 5: Schema einer Speicherkarte. Die äußeren Anschlüsse der Karte sind angedeutet (nach [Rankl 99, S.49]).

Speicherkarten sind synchrone Karten, das heißt die Kommandos werden von außen durch elektrische Signale an die Karte gesendet.

Ein Beispiel für eine Speicherkarte ist die Krankenversicherungskarte, auf der Informationen über Krankenversicherte abgespeichert sind.

Mikroprozessorkarten

Eine Weiterführung des Konzeptes, integrierte Schaltungen auf einer Chipkarte unterzubringen, ist die Mikroprozessorkarte, die in Abb. 6 schematisch dargestellt ist.

Auf ihr ist ein vollständiger Mikrorechner untergebracht, der *Mikrocontroller* genannt wird. Seine Hauptkomponente ist ein Mikroprozessor (CPU).

Zusätzlich verfügt der Mikrocontroller über drei Arten von Speicher: ein ROM, auf dem das Betriebssystem der Karte fest verankert ist, ein EEPROM, das heißt ein nichtflüchtiger Speicher, für Teile des Betriebssystems und zur Speicherung von Daten (entsprechend dem EEPROM einer Speicherkarte, s.o.) und RAM als flüchtigen Arbeitsspeicher.

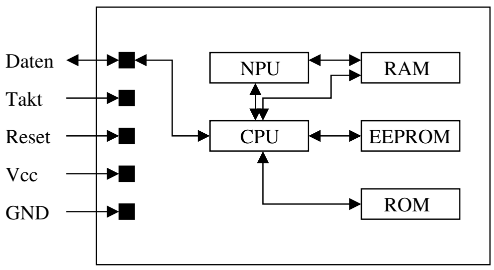

Abbildung 6: Schema einer Mikroprozessorkarte. Die äußeren Anschlüsse der Karte sind angedeutet (nach [Rankl 99, S.50]).

Als weitere Komponente kann in einem Mikrocontroller ein Coprozessor (NPU), auch *Akzelerator* genannt, vorhanden sein. Dabei handelt es sich um eine Funktionseinheit, die für eine bestimmte Rechenoperation sehr hoch optimiert ist und diese Operation wesentlich schneller ausführen kann als der Zentralprozessor. Angestoßen wird die Berechnung durch den Zentralprozessor, der ein Signal an den Coprozessor sendet und danach auf das Fertig-Signal des Coprozessors wartet. Die Rechenoperation ist zumeist eine Berechnung im Kontext von kryptographischen Algorithmen, ist aber je nach Anforderung in verschiedenen Anwendungsgebieten unterschiedlich.

Als Schnittstelle nach außen besitzt der Mikrocontroller eine Datenleitung (siehe Abb. 6). Eine Steuerleitung wie die von Speicherkarten gibt es nicht, weil die Steuerung nach festgelegten Kommandos erfolgt, die im Betriebssystem des Mikrocontrollers festgelegt sind. Mikroprozessorkarten sind also asynchrone Karten.

3.4.2 Aufbau einer Chipkarte

Der physikalische Aufbau einer Chipkarte soll im Folgenden beschrieben werden. Dabei werden nur Mikroprozessorkarten betrachtet, weil nur sie für die Anwendung, nämlich die Implementierung von Zufallszahlengeneratoren, geeignet sind. Speicherkarten sind zwar vom äußeren Aufbau identisch mit Mikroprozessorkarten, sind aber nicht programmierbar, weil kein Prozessor auf ihnen vorhanden ist (siehe 3.4.1).

Es wird in den folgenden Abschnitten analog zu [Rankl 99, S. 56ff.] vorgegangen.

Äußerer Aufbau

Die Formate der Karten sind durch die ISO-Norm 7810 festgelegt. In Abb. 7 sind die Abmessungen einer Karte im ID-1-Format dargestellt. Neben der ID-1-Karte, die am verbreitetsten ist, gibt es das kleinere ID-00-Format, das bisher kaum verwendet wird, und das ID-000-Format, das speziell für Karten in Mobiltelefonen definiert wurde. Neben dem Kartenformat sind in der ISO-Norm auch die physikalischen Eigenschaften von Chipkarten festgelegt. Dazu zählen nicht nur die Werkstoffe, sondern auch die physikalische Belastbarkeit der Karte, beispielsweise die mechanische Belastbarkeit und die Temperaturbeständigkeit. Details sind in [Rankl 99, S. 521ff.] zu finden.

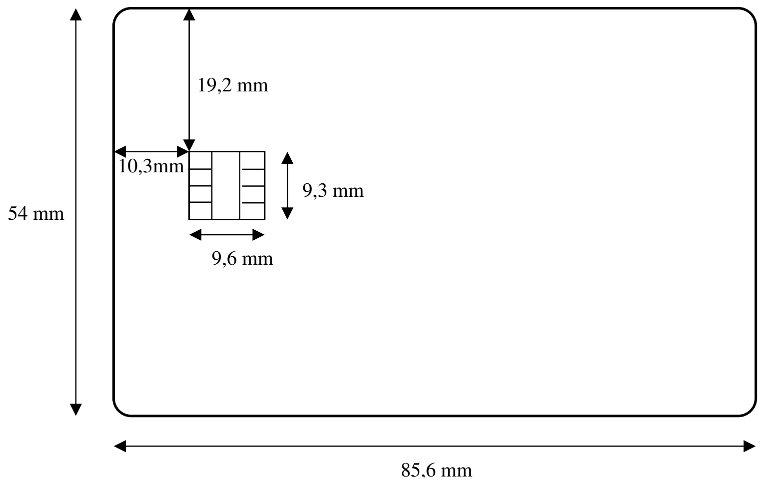

Abbildung 7: Maße einer Chipkarte im ID-1-Format. Die Anschlüsse des Chips sind angedeutet. Die Toleranzen sind nicht eingezeichnet (nach [Rankl 99, S.57, S.110]).

Anschluss des Mikrochips

Die heutzutage am weitesten verbreitete Technik, den Mikrochip in der Karte unterzubringen, ist die Chip-on-Flex-Technik.

Dabei wird zunächst ein Gehäuse für den Chip, das Chip-on-Flex-Modul, hergestellt. Das Modul umfasst erst einmal nur die nach außen sichtbaren metallischen Anschlüsse der Chipkarte. Der Chip wird auf das Modul aufgesetzt, und die Kontakte des Chips werden mit dünnen Drähten mit den Metallkontakten des Moduls verbunden (Draht-Bond-Verfahren, Wire-Bonding). Es wird ein Loch in den Kartenkörper gefräst und das Modul in dieses Loch geklebt. Das Ergebnis des Einbaus ist in Abb. 8 skizziert.

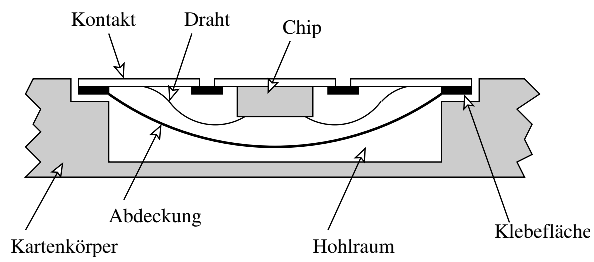

Abbildung 8: Querschnitt durch eine Chipkarte in der Chip-on-Flex-Technik (nach [Rankl 99, S.76]).

Das Chipmodul verfügt über 8 Anschlüsse. Die Nummerierung dieser Kontakte und deren Belegung sind in Abb. 9 dargestellt. Die Signale sind wie folgt definiert:

- **`CLK`**: Takt-Signal für die Hardware. Auf der Chipkarte ist kein Taktgenerator vorhanden, daher muss der Takt vom Terminal erzeugt und an die Karte angelegt werden.
- **`I/O`**: Datenleitung. Da es nur *eine* Datenleitung gibt, ist die Kommunikation nur im Halbduplex-Modus möglich. Also sendet entweder das Terminal Daten oder die Chipkarte, aber nie beide gleichzeitig.
- **`GND`**: Masse
- **`RST`**: Reset-Signal

➯ **Vcc**: Versorgungsspannung

➯ **Vpp**: Programmierspannung (wird nicht mehr benutzt)

C1		C5
C2		C6
C3		C7
C4		C8

Vcc		GND
RST		Vpp
CLK		I/O
NC		NC

Abbildung 9: Nummerierung und Belegung der Anschlüsse einer Chipkarte. Die Anschlüsse C4 und C8 sind unbenutzt („Not Connected“) (nach [Rankl 99, S.76]).

3.4.3 Datenübertragung von/zu einer Chipkarte

Ablauf einer Chipkartensitzung

Wenn eine Chipkarte in ein Kartenterminal hineingeführt wird, soll eine Verbindung zwischen Karte und Terminal etabliert werden. Der erste Schritt zur Herstellung der Verbindung ist die Initialisierung auf elektrischer Ebene. Weil keine undefinierten Zustände der Signale auftreten sollen, ist die Reihenfolge der Signalwechsel bei der Beschaltung von großer Bedeutung. Zuerst wird die Masse angelegt, dann die Versorgungsspannung und erst dann das Taktsignal für den Mikroprozessor (siehe Abb. 10). Der Mikrocontroller ist nach dem Anlegen des Taktes in Betrieb und wird durch das Reset-Signal in einen definierten Anfangszustand gebracht. Sobald von der Chipkarte an die Datenleitung ein definierter Wert angelegt wird, wird das Reset-Signal vom Terminal zurückgesetzt (wegen negativer Logik auf „high“).

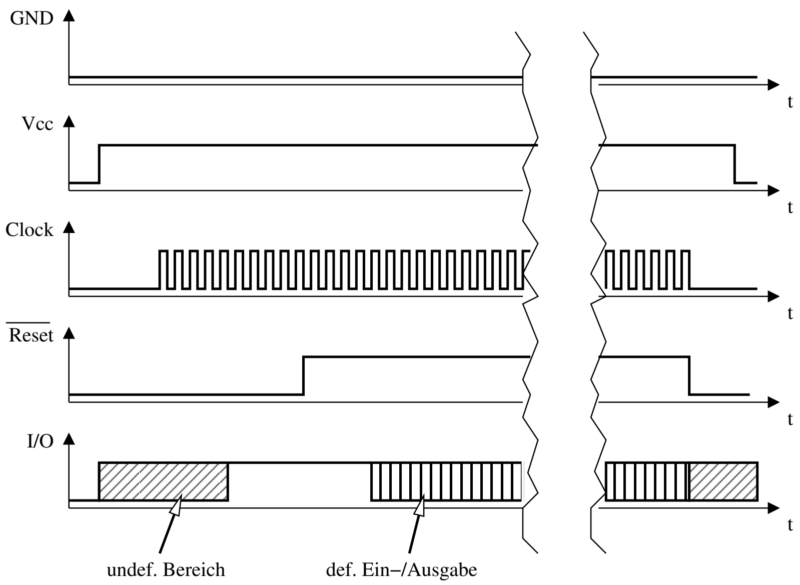

Abbildung 10: Ein-/Ausschaltsequenz einer Chipkarte. Die Abfolge der Signalwechsel an den Anschlüssen der Karte ist qualitativ skizziert (nach [Rankl 99, S.87]).

Nach dem Initialisieren der Chipkarte auf elektrischer Ebene erfolgt die Initialisierung der Kommunikation. Sie ist zusammen mit dem Kommandoaustausch zwischen Terminal und Karte als Ort-Zeit-Diagramm in Abb. 11 dargestellt.

Die Chipkarte reagiert auf den Reset mit einem ATR-Datenpaket („Answer to Reset“). Der ATR ist ein Datenstring, der Übertragungsparameter für die Kommunikation festlegt; ausführliche Details sind in [Rankl 99,

S. 330ff.] zu finden.

Wenn das Kartenterminal einige von der Chipkarte festgelegten Übertragungsparameter ändern möchte, so kann es das direkt im Anschluss an das empfangene ATR tun. Dazu kann es eine Protocol Type Selection (PTS) durchführen, indem es eine PTS-Anfrage an die Karte sendet. Die Karte antwortet auf die Anfrage und kann bestimmte Änderungen zulassen oder nicht (Details in [Rankl 99, S. 342ff.]).

Nach der Verhandlungsphase über die Kommunikationsparameter erfolgt die eigentliche Kommunikation. Dabei sendet das Chipkartenterminal Kommandos an die Chipkarte, und die Chipkarte sendet jeweils Antworten, wie in Abb. 11 skizziert. Das heißt, dass die Chipkarte keine Kommandos an das Terminal senden darf. Diese Art der Kommandoabarbeitung heißt *Challenge-Response-Verfahren*. Die Kommandos werden weiter unten in diesem Abschnitt vorgestellt.

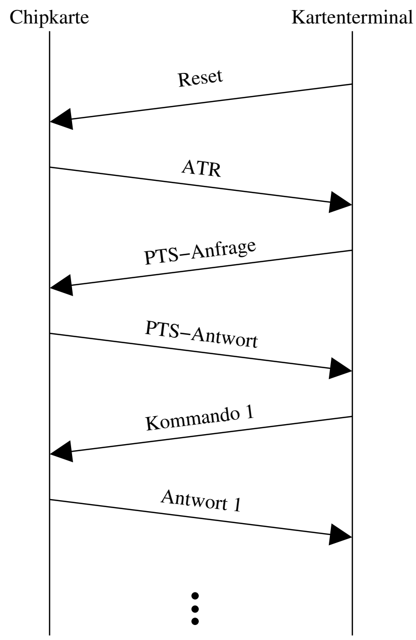

Abbildung 11: Ort-Zeit-Diagramm des Datenaustauschs zwischen Kartenterminal und Chipkarte. Horizontal sind die beiden Orte aufgetragen, und die Zeitachse verläuft nach unten (nach [Rankl 99, S.324]).

Kommunikationsprotokolle

Für den Entwurf und die Beschreibung der Datenübertragung zwischen dem Kartenterminal und der Chipkarte wird das OSI-Modell („Open Systems Interconnection“) benutzt, das ebenfalls zur Darstellung der Kommunikation in Rechnernetzwerken verwendet wird.

Im Falle von Chipkarten besteht das Modell allerdings nicht aus 7, sondern aus 3 Schichten (siehe Abb. 12).

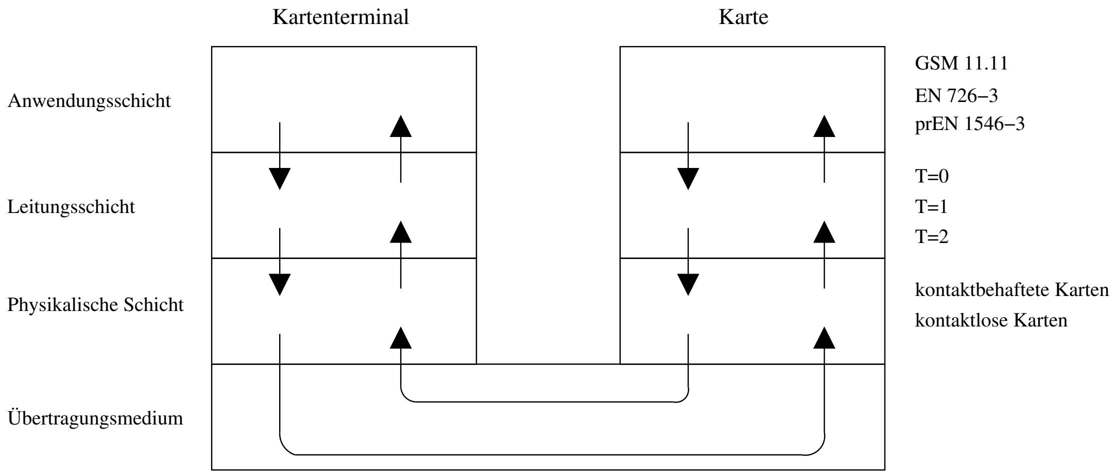

Abbildung 12: Das OSI-Modell der Kommunikation zwischen Kartenterminal und Chipkarte. Dargestellt sind die 3 Schichten und das Übertragungsmedium mit den Datenflüssen zwischen den Schichten. Zu den Schichten sind einige Protokolle angegeben (nach [Rankl 99, S.325]).

Die unterste Schicht, die physikalische Schicht, ist für den Austausch von Bitströmen zuständig. Es gibt zwei unterschiedliche Protokolle, die genormt sind: Kontaktbehaftete Karten (ISO/IEC 7816-3) und kontaktlose Karten (ISO/IEC 10 536-3).

Schicht 2 ist die Leitungsschicht. Ihre Aufgaben sind die Übermittlung von Dateneinheiten (byte- oder blockweise), die Bereitstellung einer fehlerfreien Verbindung (Fehlerkorrektur) sowie die Zugriffssteuerung (halb- oder vollduplex). Die am meisten verwendeten Protokolle der Leitungsschicht sind das einfache T=0 (sprich „Transport Protocol 0“) nach ISO-Norm 7816-3, T=1 und T=14, das ein Standard der Telekom AG ist und nur in Deutschland Verwendung findet. Für Details wird auf [Rankl 99, S. 354ff.] verwiesen.

Die oberste Schicht ist die Anwendungsschicht, in der Protokolle der Anwendungsdomäne verwendet werden. Das ist beispielsweise das GSM-Protokoll („Global System for Mobile Communications“), das im Bereich des Mobilfunks benutzt wird.

Chipkarten-Kommandos

Es gibt eine Vielzahl möglicher Kommandos, die ein Chipkartenterminal an eine Chipkarte senden kann. Die Anzahl der Kommandos ist sogar derart umfangreich, dass es keine Chipkarte gibt, die alle Kommandos verarbeiten kann; das ist vom dafür nötigen Speicheraufwand schon nicht möglich. Stattdessen sind die Kommandos, die eine Chipkarte versteht, vom Anwendungsgebiet abhängig. So gibt es für den Zahlungsverkehr oder für die Telekommunikation unterschiedliche Kommandos, die nur die entsprechenden Chipkarten verarbeiten können.

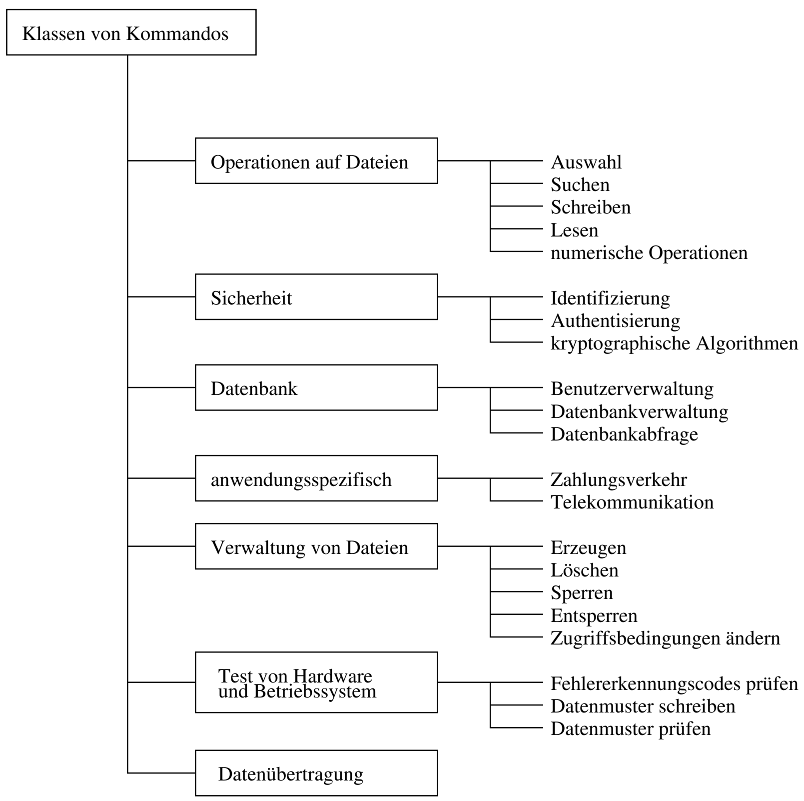

Abbildung 13: Klassifizierungsbaum der Kommandos an Chipkarten (nach [Rankl 99, S.396]).

Für Chipkarten-Kommandos gibt es diverse Normen und Standards, beispielsweise EMV („Europay, Mastercard, VISA“ – die Marktführer) im Zahlungsverkehr oder GSM im Bereich der Telekommunikation.

Einen Überblick über die Chipkartenkommandos liefert der Klassifizierungbaum der Kommandos in Abb. 13. Für diese Studienarbeit sind die Kommandos zur Erzeugung von Zufallszahlen und zur Authentifikation von größtem Interesse. Sie sollen daher nun vorgestellt werden (nach [Rankl 99, S. 415ff.].

- **GET CHALLENGE**: Fordert die Karte auf, eine Zufallszahl an das Terminal zu senden.
- **INTERNAL AUTHENTICATE**: Authentifikation der Chipkarte, das heißt die Chipkarte muss sich dem Terminal gegenüber authentifizieren. Das Terminal gibt eine Zahl als Parameter an und fordert die Karte auf, diese Zahl mit einem zuvor vereinbarten Schlüssel zu verschlüsseln. Diese verschlüsselten Daten sendet die Karte an das Terminal, das sie mit den von ihm selbst verschlüsselten Daten vergleicht. Wenn die beiden verschlüsselten Daten gleich sind, hat sich die Karte authentifiziert.
- **EXTERNAL AUTHENTICATE**: Authentifikation des Terminals gegenüber der Chipkarte. Man beachte, dass dieses Kommando vom Terminal aus gesendet wird, weil eben alle Kommandos vom Terminal gesendet werden müssen (vgl. Abb. 11). Die Chipkarte muss also so programmiert sein, dass sie erst dann Zugriff auf die Daten gewährt, wenn das Terminal sich gegenüber der Chipkarte authentifiziert, das heißt EXTERNAL AUTHENTICATE aufgerufen hat.

  Das Terminal sendet als Parameter eine mit einem zuvor vereinbarten Schlüssel verschlüsselte Zufallszahl, die es mit GET CHALLENGE von der Chipkarte angefordert hat. Die Chipkarte führt dieselbe Verschlüsselung durch und vergleicht die beiden verschlüsselten Informationen. Wenn sie gleich sind, hat sich das Terminal authentifiziert, und die Karte gibt die Benutzung der Datenkommandos frei.

4 Bewertungskriterien für Zufallsgeneratoren

4.1 Überblick über Anforderungen an Zufallsgeneratoren

In diesem Abschnitt wird vorgestellt, welche Eigenschaften ein Generator für Pseudozufallszahlen besitzen soll. Diese Eigenschaften spiegeln die Qualität des Generators wider. Dabei werden zunächst allgemeine Kriterien dargelegt, und dann folgen einige Kriterien, die sich dadurch ergeben, dass die Zufallsgeneratoren auf Chipkarten implementiert werden sollen.

Allgemeine Anforderungen an einen Zufallszahlengenerator

Ein guter Zufallszahlengenerator muss die folgenden Eigenschaften erfüllen (nach [Härtel 94, S. 27f.]):

- Die erzeugten Zufallszahlen sind *gleichverteilt*, das heißt sie treten alle mit derselben Wahrscheinlichkeit auf, z.B. 0 und 1 beide jeweils mit einer Wahrscheinlichkeit von 50%.
- Die erzeugten Zufallszahlen sind *unkorreliert*. Das bedeutet, sie sind alle voneinander stochastisch unabhängig. Es ist jedoch so, dass die Zufallszahlengeneratoren die nächsten Zahlen auf zuvor erzeugte Zahlen beziehen, also dass der aktuelle interne Zustand des Generators von den zuvor erzeugten Zahlen abhängt. Dadurch sind die Ergebniswerte in *jedem* Zufallsgenerator korreliert. Die Forderung muss demnach genauer lauten: Die Korrelation der erzeugten Zufallszahlen ist sehr klein, so dass sie mit statistischen Untersuchungen nicht gefunden wird.
- Der Generator ist *effizient*. Er muss einen geringen Platz- und Zeitbedarf besitzen, so dass er möglichst mit minimalem Speicheraufwand sehr viele Zufallszahlen in kurzer Zeit erzeugt.
- Der Generator besitzt eine *große Periode*. Es ist unmöglich, einen Zufallsgenerator zu entwerfen, dessen Folge der erzeugten Zufallszahlen sich nicht wiederholt, der also eine unendliche Periode besitzt. Daher wird zumindest eine große Periode gefordert.

  Die Periode ist ausreichend groß, wenn sie größer ist als die Anzahl der Zahlen, die mit dem Zufallsgenerator erzeugt werden sollen.
- Der Generator ist *einfach implementierbar*. Das bezieht sich sowohl auf Hardware- als auch Software-Implementierungen. Je nach Generator kann es allerdings aufwändiger sein, ihn entweder in Hardware oder in Software umzusetzen.
- Das Verfahren darf *öffentlich bekannt* sein. Das ist keine Einschränkung an den Zufallszahlengenerator, weil er trotzdem Zufallszahlen erzeugen kann: Geheim gehalten werden, soweit es erforderlich ist, der Startwert des Generators und damit sein interner Zustand.

Anforderungen an Zufallsgeneratoren auf Chipkarten

Für die Implementierung von Zufallszahlengeneratoren auf Chipkarten, die eines der Hauptthemen dieser Studienarbeit ist, gelten besondere Anforderungen. Sie sollen nun vorgestellt werden.

- Der Generator muss *besonders schnell* sein. Der Grund dafür ist, dass die Antwortzeit der Chipkarte möglichst gering sein soll, um Realzeitanforderungen zu genügen. Inklusive der Kartenansteuerung, der Authentifikation und dem Kommunikationsoverhead (siehe Abschnitt 3.4.3) soll die Antwortzeit nicht mehr als 0,3 Sekunden betragen, damit der gesamte Vorgang vom Menschen als augenblicklich angesehen wird.

  Die notwendige Antwortzeit eines Zufallsgenerators ist allerdings sehr stark von der Implementierung abhängig. Deshalb kann der Zeitaufwand lediglich von der Größenordnung her zwischen verschiedenen Generatoren verglichen werden.

- Der Generator muss eine *geringe elektrische Leistungsaufnahme* haben. Sie ist abhängig vom Hardwareaufwand, weil jeder Transistor elektrische Leistung aufnimmt und in Wärmeleistung umsetzt.

  Die Leistungsaufnahme kann auch dadurch minimiert werden, dass sich in zwei aufeinander folgenden erzeugten Zufallszahlen möglichst wenige Bits ändern, das heißt von 0 auf 1 bzw. umgekehrt wechseln. Der Grund dafür ist die inhärente Eigenschaft von CMOS-Schaltkreisen („Complementary Metal Oxide Semiconductor“ – eine Schaltkreisfamilie), dass die Leistungsaufnahme beim Wechsel von Low- und High-Spannungspegel sehr groß ist, während sie im Fall des konstant bleibenden Pegels minimal ist.

- Der Zufallsgenerator muss mit *geringem Hardwareaufwand* umsetzbar sein. Die Gründe dafür sind die schon genannte Verlustleistung und die begrenzte zur Verfügung stehende Chipfläche.

4.2 Komplexität

Zeitkomplexität und Antwortzeit

Ein wichtiges Kriterium zur Beurteilung der Qualität eines Zufallszahlengenerators ist die Zeitkomplexität bei der Erzeugung der Zufallszahlen. Wie bereits erwähnt, sind hier 0,3 Sekunden die obere Grenze für die Antwortzeit der Chipkarte auf eine Anfrage nach einer Zufallszahl. Die Antwortzeit setzt sich aus der Zeitdauer für die Kommunikation mit dem Terminal und der Zeitdauer für die Erzeugung der Zufallszahl zusammen.

Nach Abb. 11 (Seite 27) ist vor dem Anfragekommando, welches die Chipkarte zur Erzeugung einer Zufallszahl auffordert, noch der Reset auszuführen. Dies geschieht immer, wenn die Chipkarte in das Terminal eingesteckt wird und damit die Versorgungsspannung, der Takt und das Resetsignal angelegt werden (siehe Kapitel 3.4.3). Die Karte antwortet innerhalb von 400 bis 40 000 Taktzyklen mit einem ATR (siehe [Rankl 99, S. 330] oder Kapitel 3.4.3). Die PTS-Anfrage (siehe Kapitel 3.4.3) wird hier zeitlich nicht in Betracht gezogen, da bei den in dieser Arbeit vorgestellten Anforderungen davon ausgegangen werden kann, dass die Chipkarten auf die Terminalanforderungen abgestimmt sind, so dass eine Änderung des Protokolls nicht erforderlich ist. Für die Übertragungsraten gebräuchlicher Kommandos werden in [Rankl 99, S. 359ff.] für das Protokoll T=0 etwa 70 000 bis 134 000 Taktzyklen angegeben. Das bedeutet, dass allein für die Kommunikation zwischen Terminal und Chipkarte bereits 308 000 Taktzyklen benötigt werden könnten. Unter Benutzung der Standardfrequenz von 3,5712 MHz sind so bereits 0,086 Sekunden verstrichen, und für die Generierung der Zufallszahl stehen nur noch 0,218 Sekunden zur Verfügung, um die obere Grenze von 0,3 Sekunden einzuhalten.

Für die einzelnen Generatoren wird in Abhängigkeit vom Takt die Anzahl der erzeugten Zufallsbits angegeben. Um daraus einen besser zu interpretierenden und vergleichbareren Wert zwischen den verschiedenen Generatoren zu erhalten, wird der Wert unter der Annahme, dass die Karte mit 3,5712 MHz getaktet wird, in einen Wert umgerechnet, der die Zeitkomplexität in Bits pro Sekunde angibt. So lassen sich die Generatoren einfach vergleichen, und es wird offensichtlich, ob die Generatoren unterhalb der Zeitschranke von 0,218 Sekunden bleiben. Es lässt sich dann sogar leicht angeben, welche Anzahl von Bits innerhalb der Zeitgrenze erzeugt werden könnten.

Liefert als Beispiel ein Generator in 640 Takten genau 2 Zufallsbits, dann erzeugt er bei 3,5712 Millionen Verarbeitungsschritten 11 160 Zufallsbits. Die Leistung dieses Generators liegt also bei 11 160 Bit/s. Innerhalb von 0,2 Sekunden könnte der Generator also 2 232 Bit erzeugen.

Speicherbedarf

Ein weiteres wichtiges und restriktives Kriterium ist der Speicherbedarf, den der Generator auf der Chipkarte erfordert, da die Chipfläche begrenzt ist. Hierbei werden zwei Kriterien unterschieden. Einmal der Speicherplatz, angegeben in Anzahl der Flipflops, die die Operanden in Abhängigkeit ihrer Größe benötigen. Hierbei wird einmal der benötigte Registerplatz für die Operanden im statischen Zustand angeben, da die Möglichkeit eines Akzelerators auf der Chipkarte besteht, der dann für den benötigten dynamischen Speicher zuständig wäre. Die benötigten Register während der Berechnung der Zufallszahlen durch Hilfsvariablen werden zusätzlich angeben, sind aber bei Vorhandensein eines Akzelerators unerheblich; ist dieser nicht vorhanden, so gilt diese Aussage nicht.

Als zweites Kriterium wird durch das Programm `AMS SYNOPSYS` der Speicherzellenbedarf in verschiedenen Abhängigkeiten ermittelt. Die benutzten Einstellungen, die auch die Speicherzellenangabe beeinflussen, sind:

Input	: CSX,	
AMS Process	: CSA/CSD	CMOS,
Strukturgröße	: 0,35 μm.	

4.3 Hardware-Kriterien

Akzeleratoren

Ein Akzelerator ist eine Funktionseinheit, die bestimmte komplexere Berechnungen optimiert ausführen kann (siehe Kapitel 3.4.1, Abschnitt „Mikroprozessorkarten“). Ist auf einer Chipkarte ein Akzelerator vorhanden, der einen Verarbeitungsschritt optimiert ausführt, der auch von dem Zufallszahlengenerator benutzt wird, dann ist es nahe liegend, den Akzelerator auch für den Generator zu benutzen. Bei den vorgestellten Generatoren betrifft das den BBS-Generator (siehe Kapitel 5.2), der die Quadrierung von Zahlen benutzt und den EC-Generator, der die Punktaddition einer elliptischen Kurve benutzt. Geht man vom Vorhandensein eines passenden Akzelerators aus, dann schneiden die beiden Generatoren nicht mehr so schlecht im Vergleich zu RC4 und LFSR bezüglich des Speicherzellenbedarfs ab, da die aufwändige Funktionseinheit zur Quadrierung bzw. der Punktaddition schon auf der Karte vorhanden ist. Die Zufallsgeneratoren werden detailliert in Kapitel 5 verglichen.

Leistungsaufnahme

Die Leistungsaufnahme sollte aus den technischen Gründen der Umsetzung in Wärmeleistung und der Dimensionierung der einzelnen dotierten Gebiete auf der Chipfläche so gering wie möglich sein. Auch im Hinblick auf kontaktlose Chipkarten, die ihre Stromversorgung nicht durch das Kartenlesegerät, sondern durch induktive Kopplung erhalten, ist eine minimale Leistungsaufnahme ein entscheidendes Qualitätsmerkmal. Das hier benutze Maß wird keine konkrete Leistungsangabe, sondern die Angabe der kippenden Bits sein, da diese in erster Linie für die Leistungsaufnahme verantwortlich sind. Dazu werden in Abhängigkeit von der Erzeugung einer Zufallszahl die Anzahl der Wechsel der Flipflopinhalte von 0 auf 1 und umgekehrt angegeben werden.

4.4 Test auf Gleichverteilung

Die von einem Zufallsgenerator erzeugten Zahlen sollen, wie bereits in Abschnitt 4.1 gesagt, gleichverteilt sein, das heißt die erzeugten Zahlen sollen alle mit gleicher Wahrscheinlichkeit auftreten. Da die Zufallsgeneratoren in dieser Arbeit auf einer Chipkarte implementiert werden sollen, wird angenommen, dass die erzeugten Zufallszahlen als Binärzahlen vorliegen, also mit den Ziffern 0 und 1 dargestellt werden.

Die Anzahl der Binärziffern der Zufallszahl werde im Folgenden mit k bezeichnet, und n sei die Anzahl der vorliegenden Zufallszahlen. Es wird davon ausgegangen, dass eine Stichprobe $x_1, x_2, \ldots, x_n$ von Zufallszahlen von einem Zufallsgenerator erzeugt wurde.

Es gibt grundsätzlich zwei unterschiedliche Ansätze zum Testen auf Gleichverteilung: Entweder wird getestet, ob in jeder Zufallszahl durchschnittlich gleich viele Einsen und Nullen auftreten, oder es wird getestet, ob jede mögliche Zufallszahl mit der gleichen Wahrscheinlichkeit auftritt. Beide Testalternativen werden in den beiden folgenden Abschnitten vorgestellt[6].

[6]Bei idealen Zufallsgeneratoren würde es ausreichen, nur einen der beiden Tests durchzuführen. In Kapitel 5 werden aber Unterschiede zwischen beiden Tests festgestellt werden, und deshalb werden hier beide Verfahren vorgestellt.

Prüfen der bitweisen Wahrscheinlichkeit

Es sollen nun zwei Methoden dargestellt werden, um zu prüfen, ob Einsen und Nullen jeweils mit der Wahrscheinlichkeit $\frac{1}{2}$ auftreten, also die bitweise Wahrscheinlichkeit $\frac{1}{2}$ ist. Ein erster Ansatz dazu könnte sein, die erzeugten Zufallsbits als Sequenz anzusehen und zu prüfen, ob gleich viele Einsen und Nullen auftreten. Dieses Verfahren entspräche dem Verfahren im folgenden Abschnitt „Prüfen der blockweisen Wahrscheinlichkeit“ mit der Blocklänge 1. Die von den Zufallsgeneratoren erzeugten Zahlen sind allerdings immer k Bit lang, das heißt es werden immer Blöcke der Länge k ausgegeben.

Um zu testen, ob in jedem k Bit langen Bitblock Einsen und Nullen jeweils mit der Wahrscheinlichkeit $\frac{1}{2}$ auftreten, müssen die Einsen in jedem Block gezählt werden. Die Nullen brauchen nicht gezählt zu werden, weil sie sich ergeben, indem k minus Anzahl der Einsen gerechnet wird. Es können 0 bis k Einsen in jeder Zufallszahl auftreten, weil die Länge einer jeden Zufallszahl k Stellen ist. Es ist allerdings *nicht* so, dass jede Anzahl von Einsen mit gleicher Wahrscheinlichkeit auftritt. Vielmehr ergibt sich eine Binomialverteilung, weil das Erzeugen von aufeinander folgenden Zufallsbits als Bernoulli-Experiment aufgefasst werden muss. Die Wahrscheinlichkeit für das Auftreten von i Einsen $(1 \leq i \leq k)$ ist

$$P(X = i) = \binom{k}{i} p^i (1-p)^{k-i}. \tag{4}$$

Ob diese theoretische Verteilung zutrifft, wird im Folgenden mit zwei Verfahren überprüft: mit dem Chi-Quadrat-Test und der Anpassung an eine Normalverteilung.

Mithilfe des Chi-Quadrat-Tests, der in Kapitel 3.3 vorgestellt wurde, kann getestet werden, ob die Anzahlen der Einsen, die in den Zufallszahlen gezählt werden, binomialverteilt sind.

Die zu untersuchende Stichprobe seien die vorliegenden n Zufallszahlen $x_1, x_2, \ldots, x_n$, und k sei die Anzahl der Bits dieser Zufallszahlen. k ist gleichsam die Anzahl der Freiheitsgrade, also die möglichen Anzahlen an Einsen in den Zufallszahlen. Das Signifikanzniveau, das heißt das Sicherheitsniveau, wird auf $\alpha = 0,95$ festgelegt. Die Einsen der Zufallszahlen werden gezählt, und die Häufigkeiten werden mit H_i bezeichnet.

Mit diesen Angaben kann die Testgröße χ^2 mithilfe von Formel 1 aus Kapitel 3.3 berechnet werden:

$$\chi^2 = \sum_{i=1}^{k} \frac{1}{n \cdot p_i} (H_i - n \cdot p_i)^2,$$

mit der Wahrscheinlichkeit aus Gleichung 4 (mit p_1 sei die Wahrscheinlichkeit für das Auftreten einer 1 bezeichnet; es soll $p_1 = 0,5$ sein)

$$p_i = \binom{k}{i} p_1^i (1-p_1)^{k-i} = \binom{k}{i} 0,5^i (1-0,5)^{k-i} = \binom{k}{i} 0,5^k$$

und ineinander eingesetzt also

$$\chi^2 = \sum_{i=1}^{k} \frac{1}{n \cdot \binom{k}{i} 0,5^k} \left(H_i - n \cdot \binom{k}{i} 0,5^k \right)^2.$$

Der Wert χ^2 wird wie in Kapitel 3.3 dargestellt ausgewertet. Für den Fall, dass der Chi-Quadrat-Test ergibt, dass die Verteilung der Einsen nicht einer Binomialverteilung genügt, ist ein weiteres Prüfverfahren zum Testen der theoretischen Verteilung notwendig, das nun vorgestellt wird.

Ein weiterer Test, ob die Einsen der Zufallszahlen binomialverteilt sind, ist die Annäherung einer Normalverteilung an die Verteilung der Einsen. Das Verfahren, das dabei angewendet wird, ist in Kapitel 3.3 (Abschnitt „Intervallschätzung für die Normalverteilung“) beschrieben. Bei der Anpassung werden das Stichprobenmittel (siehe Formel 2, S. 22) und die Stichprobenstandardabweichung (siehe Formel 3, S. 22) berechnet.

Zum Vergleich mit der theoretisch vermuteten Binomialverteilung müssen ihr Erwartungswert EN und ihre Standardabweichung σ_N berechnet werden. Das geschieht mit den Formeln

$$EN = \sum_{i=1}^{k} k \cdot \binom{k}{i} 0,5^k \tag{5}$$

und

$$\sigma_N = \sqrt{\frac{1}{n} \cdot \sum_{i=1}^{n} (p_i - EN)^2} = \sqrt{\frac{1}{n} \cdot \sum_{i=1}^{n} (\tbinom{k}{i} 0,5^k - EN)^2}. \tag{6}$$

Wichtig ist, dass in der Formel zur Berechnung von σ_N der Bruch $\frac{1}{n}$ auftritt und nicht $\frac{1}{n-1}$ wie in Formel 3. Der Grund dafür ist, dass σ_N nicht geschätzt werden muss, sondern die exakte Standardabweichung ist.

Es können nun die theoretische und die angenäherte Verteilung verglichen werden. Wenn der Erwartungswert EN der theoretischen Verteilung im Konfidenzintervall um μ der angenäherten Verteilung liegt und wenn die Standardabweichungen σ_N und σ übereinstimmen, hat der Zufallsgenerator den Test bestanden.

Prüfen der blockweisen Wahrscheinlichkeit

Mit dem folgenden Verfahren soll getestet werden, ob alle n Zufallszahlen der Länge k mit der gleichen Wahrscheinlichkeit auftreten. Es gibt 2^k mögliche Bitkombinationen der Länge k , die der Zufallsgenerator ausgeben kann.

Es muss die Häufigkeit H_i $(1 \leq i \leq n)$ jeder ausgegebenen Zahl gezählt und mit der theoretischen Gleichverteilung verglichen werden. Die Wahrscheinlichkeit jeder Zahl ist bei einer Gleichverteilung

$$p_i = \frac{n}{2^k}.$$

Der Vergleich zwischen der Häufigkeit und der Gleichverteilung geschieht wiederum mit dem Chi-Quadrat-Test. Durch Einsetzen der Gleichverteilung in Formel 1 aus Kapitel 3.3 ergibt sich:

$$\chi^2 = \sum_{i=1}^{n} \frac{2^k}{n^2} \left(H_i - \frac{n^2}{2^k} \right)^2.$$

Der Wert χ^2 wird wiederum ausgewertet wie in Kapitel 3.3 dargestellt.

lfd. Nr.	Zufallszahl
1	0111
2	0111
3	0100
4	0011
5	0101
6	1000
7	1100
8	1011
9	0101
10	0011

Tabelle 2: Beispiel der Ausgabe von 10 4-Bit-Zufallszahlen eines Zufallsgenerators

Durchführung der Tests mit Rechnerunterstützung

Um von einem Zufallsgenerator erzeugte Zufallszahlen auf die Gleichverteilung hin zu prüfen, wurde im Rahmen dieser Arbeit ein Programm entwickelt, das Unterstützung bei der Durchführung der in diesem Abschnitt vorgestellten Tests bietet. Es ist auf der beiliegenden CD zu finden (siehe Anhänge A und B.2). An dieser Stelle soll aber nicht auf die technischen Details eingegangen werden, sondern das Programm soll anhand eines Beispiels vorgestellt werden.

In Tabelle 2 sind 10 Zahlen angegeben, die von einem Zufallsgenerator erzeugt wurden, der 4-Bit-Zahlen ausgibt[7].

Das Testprogramm untersucht zunächst die bitweise Wahrscheinlichkeit. Es liefert folgende Ausgabe, wenn es auf die Beispielzahlen aus Tabelle 2 angewandt wird:

[7] Die Zahlen wurden mit dem RC4-Generator (vgl. Kap. 5.3) unter Benutzung des Standardschlüssels erzeugt.

```
Anzahl | theoretische | Häufigkeit | Differenz |   approx.   |     Abweichung
Einsen |  Häufigkeit  |            |           | Normalvert.| Häuf.-Normalvert.
-------|--------------|------------|-----------|------------|------------------
   0   |        0.625 |      0     |     0.625 |      0.094 |   -0.094
   1   |        2.500 |      2     |     0.500 |      1.780 |    0.220
   2   |        3.750 |      5     |    -1.250 |      5.357 |   -0.357
   3   |        2.500 |      3     |    -0.500 |      2.570 |    0.430
   4   |        0.625 |      0     |     0.625 |      0.196 |   -0.196
-------|--------------|------------|-----------|------------|------------------
 SUMME |       10.000 |     10     |     0.000 |      9.997 |    0.003

Chi-Quadrat-Test:
Chi^2 =   1.867 mit n=5 Freiheitsgraden

Theoretische Verteilung:
Erwartungswert     = 2.0000
Standardabweichung = 1.0000

Approximierte Normalverteilung (d.h. tatsächliche Verteilung):
Stichprobenmittel              = 2.1000 +/- 0.7679 Einsen
Stichproben-Standardabweichung = 0.7379 Einsen
```

Diese Ausgabe ist wie folgt zu interpretieren: Die Zeilen der Tabelle spiegeln die Anzahlen der Einsen wider, die in den Zufallszahlen auftreten können. In diesem Fall gibt es 5 Zeilen, weil in jeder 4 Bit langen Zufallszahlen 0 bis 4 Einsen vorhanden sein können.

In der ersten Spalte ist die Anzahl der Einsen angegeben, auf die sich die jeweilige Zeile bezieht. Die Spalte 2 gibt die theoretischen Häufigkeiten an. Sie sind, wie oben dargelegt, binomialverteilt (siehe Formel 4). In der folgenden 3. Spalte sind die Einsen pro Zufallszahl gezählt. Die Anzahlen können mithilfe der Tabelle 2 nachvollzogen werden. Spalte 4 zeigt die Differenzen zwischen der vorliegenden und der hypothetischen Verteilung.

Mit den Daten aus den Spalten 2 bis 4 wird der Wert von χ^2 nach Formel 1 berechnet. Aus einer Tabelle für die Chi-Quadrat-Quantile (beispielsweise [Hübner 96, S. 197]) muss der Vergleichswert c_α ausgelesen werden (vgl. Kapitel 3.3); beispielsweise $c_{0,95} = 15,09$ für $n = 5$ Freiheitsgrade.

Aus den gezählten Anzahlen wird, wie oben beschrieben, eine Normalverteilung an die vorliegende Verteilung gelegt. Das Stichprobenmittel nach Formel 2 und die Stichprobenstandardabweichung nach Formel 3 werden ausgegeben, und in der Tabellenspalte 5 werden die aus der geschätzten Normalverteilung resultierenden Anzahlen angegeben. Spalte 6 zeigt zusätzlich die Differenz zwischen der tatsächlichen Verteilung und der berechneten Normalverteilung. Zum Vergleich der theoretischen und der Normalverteilung werden die Standardabweichung und der Erwartungswert der theoretischen Verteilung berechnet (nach den Formeln 5 und 6).

Das Testprogramm prüft auch die blockweisen Wahrscheinlichkeiten, das heißt es testet, wie oft welche Zufallszahl erzeugt wird. Die Ausgabe sieht wie folgt aus (falls die Tabelle zu lang wird, gibt das Programm sie nicht aus, sondern nur das Ergebnis des Chi-Quadrat-Tests):

```
   Bitkombination | theoret. Häuf. |   Häufigkeit   |    Differenz
------------------|----------------|----------------|---------------
             0000 |          0.625 |              0 |          0.625
             0001 |          0.625 |              0 |          0.625
             0010 |          0.625 |              0 |          0.625
             0011 |          0.625 |              2 |         -1.375
             0100 |          0.625 |              1 |         -0.375
             0101 |          0.625 |              2 |         -1.375
             0110 |          0.625 |              0 |          0.625
```

Bitkombination	theoret. Häuf.	Häufigkeit	Differenz
0111	0.625	2	-1.375
1000	0.625	1	-0.375
1001	0.625	0	0.625
1010	0.625	0	0.625
1011	0.625	1	-0.375
1100	0.625	1	-0.375
1101	0.625	0	0.625
1110	0.625	0	0.625
1111	0.625	0	0.625
SUMME	10	10	0

```
Chi-Quadrat-Test:
Chi^2 =  15.600 mit n=16 Freiheitsgraden
```

In der ersten Tabellenspalte sind alle theoretisch möglichen Zufallszahlen angegeben, die 4 Bit lang sind. In der folgenden Spalte sind die Häufigkeiten der Gleichverteilung angegeben, denen die erzeugten Zufallszahlen im Idealfall genügen. Die dritte Spalte zeigt das Ergebnis des Auszählens der Zahlen. Das Ergebnis kann einfach mithilfe von Tabelle 2 verifiziert werden. Die letzte Tabellenspalte gibt die Abweichungen zwischen theoretischer und vorliegender Häufigkeit an.

Das Ergebnis des Chi-Quadrat-Tests (siehe obiger Abschnitt „Prüfen der blockweisen Wahrscheinlichkeit“) ist unter der Tabelle angegeben.

4.5 Test auf Unabhängigkeit

Die vom Zufallsgenerator erzeugten Zahlen müssen auf Unabhängigkeit geprüft werden. Das bedeutet, dass getestet werden muss, ob sie hinreichend zufällig sind. Leider gibt es kein eindeutiges, quantifizierbares Merkmal einer zufälligen Zahlenfolge (nach [Härtel 94, S. 29]), so dass mehrere Tests notwendig sind, um die Unabhängigkeitseigenschaft zu prüfen.

Die zu testende Hypothese ist beim Test auf Unabhängigkeit:

$$\mathcal{H}_0 : \quad U_i \text{ unabhängig von } U_j \quad \forall i \geq j,$$

das heißt aufeinander folgende Zahlen sind voneinander unabhängig.

Zum Testen auf Unabhängigkeit gibt es eine Vielzahl von Testverfahren, beispielsweise den Iterationstest, den Lückentest oder den Pokertest, um nur einige zu nennen. Eine detaillierte Vorstellung der Verfahren ist in [Härtel 94, S. 32ff.] zu finden. Es gibt ein fertiges Programmpaket namens `DIE HARD`[8], das 15 Testverfahren auf eine Datei mit erzeugten Zufallszahlen anwendet.

Statt viele Verfahren anzuwenden, ist es auch möglich, den universellen statistischen Test aus [Maurer 90] zu verwenden. Dieser Test misst die Entropie der erzeugten Zufallsbits und vergleicht sie mit einem Referenzwert. Universal ist er deshalb, weil sich laut [Maurer 90] alle Tests auf den Universaltest zurückführen lassen. Schneier stellt in [Schneier 96, S. 483] fest, dass sich aus dem universellen Test ergibt, dass eine Zufallsfolge, die komprimierbar ist, nicht wirklich zufällig ist.

Der universelle Test kann also durch eine Kompression ersetzt werden: Wenn die erzeugten Zufallszahlen sich nicht durch die Verwendung redundanzreduzierender Codes, beispielsweise dem Huffman-Code, komprimieren lassen, dann sind sie unabhängig.

Zum Testen der Zufallszahlen kann also ein gängiges Kompressionsprogramm, zum Beispiel `gzip` oder `zip` verwendet werden. Dieses Testverfahren ist allerdings nur zum groben Einschätzen der Unabhängigkeit der Zufallszahlen geeignet, weil die Packprogramme keine Abhängigkeiten prüfen, die sich ergeben, wenn die Reihenfolge der zufälligen Bitfolge geändert wird.

[8] zu finden unter `http://stat.fsu.edu/~geo/diehard.html`

5 Vorstellung und Analyse ausgewählter Zufallsgeneratoren

5.1 LFSR

Ein *rückgekoppeltes Schieberegister* (siehe [Schneier 96, S. 429ff.]) setzt sich aus zwei Teilen zusammen, einem Schieberegister und einer Rückkopplungsfunktion. Das Schieberegister besteht aus linear geordeneten Flipflops. Die *Länge* eines Schieberegisters wird in Bit angegeben und bezeichnet die Anzahl der Register, bei einer Länge von n Bit spricht man von einem n-Bit-Schieberegister.

Wenn ein Bit in der Zufallszahlensequenz benötigt wird, dann werden die in den Flipflops gespeicherten Bits nach links in das benachbarte Flipflop geschoben, das heißt der Inhalt des Flipflops b_i wird durch den Wert des Flipflops b_{i-1} ersetzt. Das Bit b_1 wird in Abhängigkeit der anderen Bits in den Flipflops durch die Rückkopplungsfunktion berechnet. Als Ausgabe wird meist das niederwertigste Bit benutzt. Die einfachste Version eines Schieberegisters ist das linear rückgekoppelte Schieberegister, kurz LFSR („Linear Feedback Shift Register“, siehe Abb. 14).

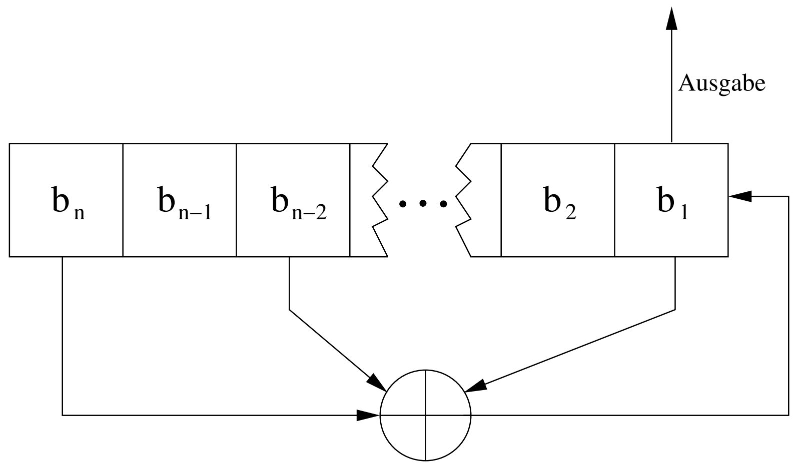

Abbildung 14: Schieberegister mit linearer Rückkopplung

Hier ist die Rückkopplungsfunktion durch eine XOR-Verknüpfung bestimmter Bits des Registers gegeben. Die Liste dieser Bits heißt *Tap Sequence* oder *Fibonacci-Konfiguration.* Ein 4-Bit-Schieberegister mit der Tap Sequence 1001, das heißt es werden die Bits b_4 und b_1 für die Rückkopplungsfunktion XOR-verknüpft, mit der Initialisierung 1111 durchläuft dann folgende Zustände: 1111, 1110, 1101, 1010, 0101, 1011, 0110, 1100, 1001, 0010, 0100, 1000, 0001, 0011 und 0111. Danach wiederholen sich die Zustände, weil der nächste Zustand wieder den Anfangszustand ergibt. Die Zufallszahlensequenz, bestehend aus den niederwertigsten Bits, lautet für dieses Beispiel: 101011001000111 . . . .

Ein LFSR mit n Bits kann 2^n Zustände annehmen. Der Zustand, der aus lauter Nullen besteht, ist hier nicht brauchbar, da ein LFSR mit Nullen in den Flipflops als einzigen Wert die Null in der Ausgabesequenz liefert. Die maximal erreichbare Periode ist also $2^n - 1$. Um diese zu erreichen, muss das Polynom, welches die Tap Sequence plus der Konstanten 1 beschreibt, ein primitives und irreduzibles Polynom modulo 2 sein.

Primitiv ist in multiplikativen Gruppen endlicher Körper eine andere Bezeichnung für ein erzeugendes Element. Das heißt ein solches Polynom erzeugt in seinen Potenzen sämtliche Elemente der Gruppe. Zusätzlich ist jede multiplikative Gruppe eines endlichen Körpers zyklisch (siehe [Biggs 89, S. 350ff.]).

Irreduzibel bedeutet, das Polynom lässt sich nicht faktorisieren. Faktorisieren bedeutet, dass das Polynom eine Nullstelle x_1 besitzt und das Polynom sich nach einer Folgerung aus dem Fundamentalsatz der Algebra (siehe [Bartsch 97, S. 107]) von $a_nx^n + a_{n-1}x^{n-1} + a_{n-2}x^{n-2} + a_{n-3}x^{n-3} + \cdots + a_1x + a_0$ in $(x - x_1) \cdot (a_nx^{n-1} + (a_nx_1 + a_{n-1})x^{n-2} + (a_nx_1^2 + a_{n-1}x_1 + a_{n-2})x^{n-3} + \cdots + (a_nx_1^{n-2} + a_{n-1}x_1^{n-3} + \cdots + a_3x_1 + a_2)x + (a_nx_1^{n-1} + a_{n-1}x_1^{n-2} + \cdots + a_2x_1 + a_1))$ umformen lässt.

Der Grad des Polynoms entspricht der Länge des Schieberegisters. Das Beispiel mit der Tap Sequence 1001 steht für das Polynom $x^4 + x + 1$. Für die Bestimmung primitiver Polynome gibt es keine allgemeine einfache

Möglichkeit. Weitere Einzelheiten dazu finden sich in [Rabin 68]. Schwach besetzte Polynome stellen nach [Schneier 96, S. 433] einen Nachteil dar, weil sie möglicherweise kryptographischen Anforderungen nicht genügen.

Der Vorteil von LFSRs ist die einfache Implementation in Hard- und Software. Der Nachteil ist, dass ein LFSR mit unbekannter Rückkopplung der Länge n mit dem effizienten *Berlekamp-Massey-Algorithmus* [Massey 69] aus nur $2n$ Ausgabebits ermittelt werden kann. Nicht ins Gewicht fällt dieser Nachteil, wenn die Anzahl der Zufallszahlen, die der Zufallsgenerator jemals erzeugen soll, kleiner als $2n$ ist, weil dann der Berlekamp-Massey-Algorithmus nicht anwendbar ist.

In diesem Zusammenhang wird für Generatoren von Zufallssequenzen auf Basis von LFSRs ein wichtiges Komplexitätsmaß angegeben. Die *lineare Komplexität* eines Generators ist definiert als die Länge des kürzesten LFSRs, das die Ausgabe des Generators simuliert. Jede von einem *endlichen Automaten* über einem *endlichen Körper* erzeugte Folge hat nach [Massey 69] endliche lineare Komplexität.

Untersuchung der erzeugten Zufallszahlen auf Gleichverteilung

Im Folgenden werden die Zufallszahlen, die vom LFSR generiert werden, auf Gleichverteilung untersucht, dabei wird wie in Kapitel 4.4 beschrieben vorgegangen. Zum Erzeugen der Zufallszahlen wird die in Kapitel 6.2 vorgestelle Implementation des LFSR in der Programmiersprache C benutzt. Das Ergebnis der Untersuchungen ist in Tabelle 3 zusammengefasst.

Bitzahl	Anzahl	χ^2 (Bit)	Ok?	EN	σ_N	$\overline{X}$	σ_S	Ok?	χ^2 (Block)	Ok?
4	100	3,587	√	2	1,000	2,070 ± 0,311	0,946	√	18,720	√
4	10 000	9,011	√	2	1,000	2,017 ± 0,033	0,957	√	17,370	√
4	1 000 000	3,603	√	2	1,000	2,000 ± 0,003	1,001	√	9,134	√
4	10 000 000	3,931	√	2	1,000	2,000 ± 0,001	0,995	√	9,498	√
4	50 000 000	0,284	√	2	1,000	2,000 ± 0,001	1,000	√	10,539	√
8	100	12,649	√	4	1,414	4,010 ± 0,471	1,432	√	243,040	√
8	10 000	4,502	√	4	1,414	4,010 ± 0,046	1,410	√	234,880	√
8	1 000 000	9,513	√	4	1,414	4,002 ± 0,005	1,416	√	271,349	√
8	10 000 000	6,603	√	4	1,414	4,001 ± 0,002	1,414	√	206,951	√
8	50 000 000	7,550	√	4	1,414	4,000 ± 0,001	1,414	√	247,467	√
12	100	5,571	√	6	1,732	5,950 ± 0,570	1,731	√	4 077,920	√
12	10 000	16,652	√	6	1,732	5,994 ± 0,057	1,732	√	4 147,584	√
12	1 000 000	9,509	√	6	1,732	6,002 ± 0,006	1,732	√	4 159,705	√
12	10 000 000	6,504	√	6	1,732	6,001 ± 0,002	1,732	√	4 187,401	√
12	50 000 000	6,730	√	6	1,732	6,000 ± 0,001	1,732	√	3 992,683	√
16	100	9,627	√	8	2,000	7,920 ± 0,634	1,926	√	65 436,000	√
16	10 000	13,173	√	8	2,000	8,008 ± 0,066	2,012	√	66 192,154	√
16	1 000 000	14,033	√	8	2,000	8,002 ± 0,007	2,001	√	65 253,863	√
16	10 000 000	10,671	√	8	2,000	8,001 ± 0,002	2,000	√	64 850,466	√
16	50 000 000	14,716	√	8	2,000	8,000 ± 0,001	2,000	√	62 427,195	√
20	100	5,403	√	10	2,236	10,040 ± 0,706	2,146	√	1 048 576,000	√
20	10 000	17,503	√	10	2,236	10,003 ± 0,073	2,228	√	1 050 110,336	√
20	1 000 000	19,623	√	10	2,236	10,003 ± 0,007	2,235	√	1 046 281,384	√
20	10 000 000	10,523	√	10	2,236	10,000 ± 0,002	2,236	√	1 038 999,393	√
20	50 000 000	7,972	√	10	2,236	10,000 ± 0,001	2,236	√	998 053,322	√

Tabelle 3: Untersuchung von Zufallszahlen, die mit dem LFSR-Generator mit einem Schieberegister der Bitbreite 30 erzeugt wurden. Als Polynom wurde $x^{30} + x^6 + x^4 + x^1 + x^0$ *benutzt, und der Startwert des Registers war* $(2AAA\ AAAA)_{16}$*. In den Tests wurden 4, 8, 12, 16 und 20 Bits aus dem Register ausgegeben. Es wurden immer die höchstwertigsten Bits genommen, bei 4 Ausgabebits also die Bits 30, 29, 28 und 27. Das Schieberegister wurde immer so oft geschoben wie Bits ausgegeben wurden, zum Beispiel wurde vor jedem Ausgeben von 20 Bit 20 mal geschoben. Das Signifikanzniveau der* χ^2*-Quadrat-Verteilung wurde auf* $\alpha = 0,95$ *festgelegt; die Quantile der* χ^2*-Verteilung wurden [Hübner 96, S. 197] entnommen.*

Vom Schieberegister wurden je nach Bitzahl entsprechend viele Flipflops genommen und jeweils vor der Ausgabe einer Zufallszahl ebenso oft geschoben. Für alle Ausgabelängen 4, 8, 12, 16 und 20 Bit wurden jeweils 100, 10 000, 1 000 000, 10 000 000 und 50 000 000 Zufallszahlen erzeugt (Tabellenspalten 1 und 2). Die erzeugten Zufallszahlen wurden den Statistiktests unterzogen, deren Ergebnisse in den Spalten der Tabelle 3 angegeben sind. Der Chi-Quadrat-Test zur Untersuchung auf Gleichverteilung der bitweisen Wahrscheinlichkeit ist in den Spalten 3 und 4 gezeigt; dabei gibt Spalte 3 den Wert von χ^2 an und Spalte 4, ob der Test mit dem berechneten Wert χ^2 bestanden wurde ($\surd$ bedeutet „bestanden“). Der Anpassungstest der bitweisen Wahrscheinlichkeit an eine Normalverteilung ist in den Spalten 5 bis 9 zu finden. Angegeben sind dort der Erwartungswert EN und die Standardabweichung σ_N der theoretischen Binomialverteilung, das empirische Stichprobenmittel $\overline{X}$ und die empirische Stichprobenstandardabweichung σ_S der Verteilung der Zufallszahlen sowie wieder eine Angabe, ob der Test bestanden worden ist. Das Ergebnis des Chi-Quadrat-Tests der blockweisen Wahrscheinlichkeiten auf Gleichverteilung ist in den Spalten 10 und 11 abgedruckt; Spalte 10 zeigt den Wert von χ^2, und in Spalte 11 ist wiederum angegeben, ob der Test bestanden wurde.

Sämtliche vom LFSR für den Test erzeugten Zufallszahlen bestehen die Statistiktests. Damit eignen sich diese Zufallszahlen mit Sicherheit für Low-Security-Anwendungen, insbesondere auch für einen Einsatz zur Erzeugung von Zufallszahlen innerhalb geschlossener Systeme ohne direkte Außenwirkung des LFSRs, beispielsweise für Initialisierungen anderer Generatoren oder zur Steuerung von Zufallsprozessen. Für einen Einsatz in Bereichen, in denen es um nichttriviale Werte geht, etwa der Absicherung von Geld- und Kreditkarten, ist von einem Einsatz eines LFSR abzusehen, da mit dem *Berlekamp-Massey-Algorithmus* [Massey 69] aus nur $2n$ Ausgabebits (n ist der Polynomgrad) der Generator vorhersagbar ist und damit ein etwaiger Kopierschutz von Geld- und Kreditkarten kein Hindernis mehr darstellt. Vor einem Einsatz von einem LFSR ist also genau zu überlegen, wie die sicherheitstechnischen Randbedingungen aussehen.

Untersuchung der erzeugten Zufallszahlen auf Unabhängigkeit

Die vom LFSR erzeugten Zufallszahlen werden nun auf ihre Unabhängigkeit überprüft. Dazu wird versucht, wie in Kapitel 4.5 beschrieben, sie mit den Packprogrammen `gzip` und `zip` zu komprimieren.

Es wurden dieselben Parameter des LFSR-Generators gewählt wie zuvor bei der Untersuchung auf Gleichverteilung (siehe voriger Abschnitt), und es wurden wiederum jeweils 100, 10 000, 1 000 000, 10 000 000 und 50 000 000 Zahlen der Längen 4, 8, 12, 16 und 20 Bit erzeugt. Das Ergebnis der Untersuchung dieser Zahlen ist in Tabelle 4 dargestellt. Die Spalten 1 und 2 geben die eben genannte Bitlänge der Zahlen sowie die Anzahl der Zufallszahlen an. In Spalte 3 ist die unkomprimierte Größe der Zufallszahlen angegeben, die in einer Datei abgespeichert wurden. Diese Größe berechnet aus $\frac{\text{Bitzahl}\cdot\text{Anzahl}}{8}$. In den Spalten 4 und 5 ist die Größe der mit `zip` bzw. `gzip` komprimierten Daten angegeben, das heißt nicht die Größe der komprimierten Datei, sondern nur die der komprimierten Daten (ohne Overhead, der sich aus den Dateiformaten ergibt).

Bitzahl	Anzahl	unkompr. [Byte]	`zip` [Byte]	`gzip` [Byte]	Ok?
4	100	50	50	81	$\surd$
4	10 000	5 000	5 000	5 031	$\surd$
4	1 000 000	500 000	500 080	500 106	$\surd$
4	10 000 000	5 000 000	5 000 765	5 000 791	$\surd$
4	50 000 000	25 000 000	25 003 815	25 003 841	$\surd$
8	100	100	100	131	$\surd$
8	10 000	10 000	10 000	10 031	$\surd$
8	1 000 000	1 000 000	1 000 155	1 000 181	$\surd$
8	10 000 000	10 000 000	10 001 530	10 001 556	$\surd$
8	50 000 000	50 000 000	50 007 630	50 007 656	$\surd$
12	100	150	150	181	$\surd$
12	10 000	15 000	15 000	15 031	$\surd$
12	1 000 000	1 500 000	1 500 230	1 500 256	$\surd$
12	10 000 000	15 000 000	15 002 290	15 002 316	$\surd$
12	50 000 000	75 000 000	75 012 070	75 012 096	$\surd$
16	100	200	200	231	$\surd$
16	10 000	20 000	20 000	20 031	$\surd$
16	1 000 000	2 000 000	2 000 310	2 000 336	$\surd$
16	10 000 000	20 000 000	20 003 055	20 003 081	$\surd$
16	50 000 000	100 000 000	100 015 917	100 015 943	$\surd$
20	100	250	250	281	$\surd$
20	10 000	25 000	25 000	25 031	$\surd$
20	1 000 000	2 500 000	2 500 385	2 500 411	$\surd$
20	10 000 000	25 000 000	25 003 815	25 003 841	$\surd$
20	50 000 000	125 000 000	125 019 695	125 019 721	$\surd$

Tabelle 4: Untersuchung der Kompressionsraten.

Bei der Betrachtung von Tabelle 4 fällt auf, dass die Informationsmenge der komprimierten Zufallszahlen nie kleiner als die der unkomprimierten Zahlen ist. Das bedeutet, dass sich die vom LFSR erzeugten Zufallszahlen nicht komprimieren lassen. Also haben die Zufallszahlen den Test auf Unabhängigkeit bestanden.

5.2 BBS

Einer der einfachsten und sichersten Zufallszahlengeneratoren ist der Blum-Blum-Shub-Generator, hier abkürzend als *BBS-Generator* bezeichnet. Er wurde 1982 auf der Crypto 82 vorgestellt und 1986 in [Blum 86] veröffentlicht und nach seinen Erfindern benannt, eine andere gebräuchliche Bezeichnung ist *quadratischer Restegenerator*.

Für den Generator benötigt man zwei spezielle und verschiedene Primzahlen p und q. Eine Primzahl p ist dann von der gewünschten Form, wenn $p = 2p_1 + 1$ und $p_1 = p_2 + 1$ ungerade Primzahlen sind und p_2 prim ist (insbesondere ergibt sich aus diesen Forderungen, dass p und q kongruent $3 \bmod 4$ sind). Diese Bedingung ermöglicht das Erreichen der maximalen Periode von $2p_2q_2$. Für den Modulus n bei der Berechnung der Zufallszahlen wird dann das Produkt der beiden Zahlen p und q benutzt. Weiterhin wählt man eine Zahl x, bei der sich für $x \bmod p$ bzw. $x \bmod q$ weder 0, 1 noch $p - 1$ bzw. $q - 1$ ergibt. Diese Einschränkungen[9] dienen ebenfalls einer langen Periode. Weitere Einzelheiten dazu sind in [Blum 86, S. 376ff.] zu finden.

Der Startwert des Generators berechnet sich mit

$$x_0 = x^2 \bmod n.$$

Jetzt kann man mit der fortgesetzten Berechnung der Rekurrenzgleichung

$$x_{i+1} = x_i^2 \bmod n$$

die Bits der Zufallszahlensequenz erzeugen, indem man das niederwertigste Bit der erzeugten x_i benutzt.

Wählen wir beispielsweise die Zahlen 23 und 47 für p und q und für x die Zahl 42, dann ergeben sich folgende Berechnungsschritte:

$$\begin{aligned}
x_0 &= 42^2 \bmod 1081 = 683 \\
x_1 &= 683^2 \bmod 1081 = 578 \\
x_2 &= 578^2 \bmod 1081 = 55 \\
x_3 &= 55^2 \bmod 1081 = 863 \\
&\vdots \\
x_{109} &= 347^2 \bmod 1081 = 418 \\
x_{110} &= 418^2 \bmod 1081 = 683
\end{aligned}$$

In diesem Beispiel wurde die maximale Periode von $2p_2q_2 = 2 \cdot 5 \cdot 11 = 110$ erreicht. Die Zahlen $p = 23 = 2 \cdot 11 + 1$, $p_1 = 11 = 2 \cdot 5 + 1$, $p_2 = 5$, $q = 47 = 2 \cdot 23 + 1$, $q_1 = 23 = 2 \cdot 11 + 1$, $q_2 = 11$ sind hier alle ungerade Primzahlen, und $x = 42$ ist kongruent $19 \bmod 23$ bzw. $42 \bmod 47$. Die Zufallszahlensequenz, aus den niederwertigsten Bits bestehend, beginnt für dieses Beispiel mit $1011\ldots$.

In dieser Version hat der Generator die Eigenschaft, dass die Bits der Sequenz weder nach links noch nach rechts vorhersagbar sind. Diese Eigenschaft verliert man, wenn man mehr als ein Bit der x_i für die Zufallszahlensequenz benutzen möchte. Um die Eigenschaft der Nichtvorhersagbarkeit der zu generierenden Bits zu behalten, gilt nach [Vazirani 84] als Grenze der möglichen zu benutzenden Bits $\log_2 m$, dabei ist m die Bitlänge der x_i.

Eine weitere vorteilhafte Eigenschaft des Generators ist, dass man das i-te Bit berechnen kann, ohne die vorherigen $i - 1$ Bits berechnen zu müssen. Das bedeutet, dass man wahlfreien Zugriff auf die einzelnen Bits der Zufallszahlensequenz hat. Dafür müssen aber die Werte für p und q bekannt sein. Die Berechnung erfolgt dann mit

$$x_i = x_0^{2^i \bmod ((p-1)(q-1))}.$$

Die Sicherheit des Verfahrens liegt in der Schwierigkeit der Faktorisierung von n. Ist das Primzahlprodukt groß genug, so ist die Zufallszahlensequenz nicht vorhersagbar, auch wenn n veröffentlicht wird. So kann

[9] http://www.contestcen.com/crypto001.htm

jeder, der n kennt, mit dem Generator Zufallszahlen erzeugen und muss nur den Startwert x geheimhalten. Nachteilig an dem Verfahren ist die Ineffizienz des BBS-Generators, denn das Quadrieren der großen Zahlen bei der Erzeugung des nächsten Zufallsbits ist aufwändig.

Untersuchung der erzeugten Zufallszahlen auf Gleichverteilung

In der Tabelle 5 sind die Ergebnisse der Untersuchung der vom BBS-Generator erzeugten Zufallszahlen auf Gleichverteilung dargestellt. Bei der Untersuchung wurde wie in Kapitel 4.4 beschrieben vorgegangen. Zum Erzeugen der Zufallszahlen wird die in Kapitel 6.3 vorgestelle Implementation von BBS in der Programmiersprache C benutzt. Der Generator erzeugt jeweils pro Quadrierung eine neue Zahl, von dieser wurden die niederwertigsten Bits entsprechend der Bitzahl entnommen, also wie in Spalte 1 angegeben 1, 2, 4, 8, 12, 16 und 20 Bit. Für alle Ausgabelängen wurden jeweils 100, 10 000 und 1 000 000 Zufallszahlen erzeugt (Tabellenspalten 1 und 2). Die erzeugten Zufallszahlen wurden den Statistiktests unterzogen, deren Ergebnisse in den Spalten der Tabelle 5 angegeben sind. Der Chi-Quadrat-Test zur Untersuchung auf Gleichverteilung der bitweisen Wahrscheinlichkeit ist in den Spalten 3 und 4 gezeigt; dabei gibt Spalte 3 den Wert von χ^2 an und Spalte 4, ob der Test mit dem berechneten Wert χ^2 bestanden wurde ($\surd$ bedeutet „bestanden“, $\times$ bedeutet „nicht bestanden“). Der Anpassungstest der bitweisen Wahrscheinlichkeit an eine Normalverteilung ist in den Spalten 5 bis 9 zu finden. Angegeben sind dort der Erwartungswert EN und die Standardabweichung σ_N der theoretischen Binomialverteilung, das empirische Stichprobenmittel $\overline{X}$ und die empirische Stichprobenstandardabweichung σ_S der Verteilung der Zufallszahlen sowie wieder eine Angabe, ob der Test bestanden worden ist. Das Ergebnis des Chi-Quadrat-Tests der blockweisen Wahrscheinlichkeiten auf Gleichverteilung ist in den Spalten 10 und 11 abgedruckt; Spalte 10 zeigt den Wert von χ^2, und in Spalte 11 ist wiederum angegeben, ob der Test bestanden wurde.

Bitzahl	Anzahl	χ^2 (Bit)	Ok?	EN	σ_N	$\overline{X}$	σ_S	Ok?	χ^2 (Block)	Ok?
1	100	3,115	$\surd$	0,5	0,500	0,414 ± 0,495	0,495	$\surd$	3,115	$\surd$
1	10 000	2,822	$\surd$	0,5	0,500	0,492 ± 0,500	0,500	$\surd$	2,822	$\surd$
1	1 000 000	0,264	$\surd$	0,5	0,500	0,500 ± 0,500	0,500	$\surd$	0,264	$\surd$
2	100	1,440	$\surd$	1,0	0,707	0,920 ± 0,720	0,720	$\surd$	2,160	$\surd$
2	10 000	1,109	$\surd$	1,0	0,707	0,995 ± 0,710	0,710	$\surd$	3,754	$\surd$
2	1 000 000	1,040	$\surd$	1,0	0,707	0,999 ± 0,707	0,707	$\surd$	1,087	$\surd$
4	100	5,000	$\surd$	2,0	1,000	1,870 ± 0,981	0,981	$\surd$	14,240	$\surd$
4	10 000	4,926	$\surd$	2,0	1,000	1,983 ± 0,999	0,999	$\surd$	17,245	$\surd$
4	1 000 000	2,017	$\surd$	2,0	1,000	1,999 ± 1,000	1,000	$\surd$	5,032	$\surd$
8	100	28,695	$\times$	4,0	1,141	3,800 ± 1,706	1,706	$\surd$	273,760	$\surd$
8	10 000	7,120	$\surd$	4,0	1,141	3,991 ± 1,428	1,428	$\surd$	298,061	$\surd$
8	1 000 000	3,109	$\surd$	4,0	1,141	3,998 ± 1,414	1,414	$\surd$	173,423	$\surd$
12	100	196,466	$\times$	6,0	1,732	5,600 ± 2,123	2,123	$\surd$	4 077,920	$\surd$
12	10 000	8,355	$\surd$	6,0	1,732	5,981 ± 1,747	1,747	$\surd$	4 059,738	$\surd$
12	1 000 000	4,504	$\surd$	6,0	1,732	5,999 ± 1,733	1,733	$\surd$	2 401,530	$\surd$
16	100	828,881	$\times$	8,0	2,000	7,680 ± 2,449	2,449	$\surd$	65 436,000	$\surd$
16	10 000	14,047	$\surd$	8,0	2,000	7,987 ± 2,014	2,014	$\surd$	65 720,294	$\surd$
16	1 000 000	38,504	$\times$	8,0	2,000	7,998 ± 2,000	2,000	$\surd$	65 213,493	$\surd$
20	100	4 741,185	$\times$	10,0	2,236	9,630 ± 2,834	2,834	$\surd$	1 048 476,000	$\surd$
20	10 000	49,923	$\times$	10,0	2,236	9,992 ± 2,249	2,249	$\surd$	1 047 803,469	$\surd$
20	1 000 000	64,489	$\times$	10,0	2,236	9,991 ± 2,259	2,259	$\surd$	996 180,423	$\surd$

Tabelle 5: Untersuchung von Zufallszahlen, die mit dem BBS-Generator erzeugt wurden. Als Parameter wurden $n = 4079 \cdot 4127 = 16\ 834\ 033$ *und* $x = 2$ *gewählt; daraus ergibt sich eine Periode von* $2\ 101\ 178$. *Ausgegeben wurden jeweils die niedrigsten 1, 2, 4, 8, 12, 16 bzw. 20 Bits. Das Signifikanzniveau der* χ^2*-Quadrat-Verteilung wurde auf* $\alpha = 0,95$ *festgelegt; die Quantile der* χ^2*-Verteilung wurden [Hübner 96, S. 197] entnommen.*

In Tabelle 5 fällt auf, dass der BBS-Generator den Test auf die blockweise Gleichverteilung und den Anpassungstest der bitweisen Wahrscheinlichkeiten an die Normalverteilung besteht, aber den Chi-Quadrat-Test der bitweisen Gleichverteilung bei den Bitzahlen größer als 4 nicht. Das hat zwei Gründe: Es werden pro erzeugtem quadratischen Rest zu viele Bits ausgegeben, und bei der Erzeugung von nur 100 Zufallszahlen tritt nicht jedes mögliche Versuchsergebnis 5 mal auf. Erst wenn alle Ergebnisse 5 mal vorkommen, ist der Chi-Quadrat-Test anwendbar und liefert brauchbare Ergebnisse. Der Modulus beträgt in diesem Beispiel 16 834 033, der größte quadratische Rest, der auftreten kann, ist also kleiner als 16 834 033. In dualer Darstellung sind höchstens $\lceil \log_2(16\,834\,032) \rceil = 25$ Bit erforderlich. Um die Sicherheit des Generators nicht zu gefährden, dürfen von den quadratischen Resten nicht mehr als $\log_2 m$ ausgegeben werden, in diesem Beispiel also nicht mehr als $\lfloor \log_2 25 \rfloor = 4$ Bit ausgegeben werden. Weiterhin sind die quadratischen Reste über das gesamte Intervall von 2 bis 16 834 033 verteilt, und somit sind die 4 Bit auch nicht in allen Fällen nutzbar, tatsächlich liegt die durchschnittliche nutzbare Anzahl der Bits unter diesem Wert. Insofern sind die Ergebnisse vom Chi-Quadrat-Test für größere Bitzahlen kein unerwartetes Ergebnis. Es zeigt deutlich die Grenzen der nutzbaren Bits, und es ist offensichtlich, dass man für größere Bitzahlen, entweder die Bits von aufeinanderfolgenden Zufallszahlen zusammensetzen oder eben einen deutlich größeren Modulus wählen muss. Eine weitere Schwierigkeit ist die jeweilige Ermittlung der Anzahl der nutzbaren Bits, die ja von der Größe der verschiedenen quadratischen Reste abhängt. Da hier die Implementierung auf Chipkarten im Vordergrund steht, werden diese Aspekte, die mit weiteren Aufwand verbunden sind, nicht weiter verfolgt.

Untersuchung der erzeugten Zufallszahlen auf Unabhängigkeit

Die vom BBS-Generator erzeugten Zufallszahlen werden nun auf ihre Unabhängigkeit geprüft. Dazu wird versucht, wie in Kapitel 4.5 beschrieben, sie mit den Packprogrammen `gzip` und `zip` zu komprimieren.

Es wurden dieselben Parameter wie bei der Untersuchung des BBS-Generators auf Gleichverteilung gewählt (siehe voriger Abschnitt), und es wurden wiederum jeweils 100, 10 000 und 1 000 000 Zahlen der Längen 1, 2, 4, 8, 12, 16 und 20 Bit erzeugt. Das Ergebnis der Untersuchung dieser Zahlen ist in Tabelle 6 dargestellt. Die Spalten 1 und 2 geben die eben genannte Bitlänge der Zahlen sowie die Anzahl der Zufallszahlen an. In Spalte 3 ist die unkomprimierte Größe der Zufallszahlen angegeben, die in einer Datei abgespeichert wurden. Diese Größe berechnet aus $\frac{\text{Bitzahl} \cdot \text{Anzahl}}{8}$. In den Spalten 4 und 5 ist die Größe der mit `zip` bzw. `gzip` komprimierten Daten angegeben, das heißt nicht die Größe der komprimierten Datei, sondern nur die der komprimierten Daten (ohne Overhead, der sich aus den Dateiformaten ergibt).

Bitzahl	Anzahl	unkompr. [Byte]	`zip` [Byte]	`gzip` [Byte]	Ok?
1	100	13	13	40	$\surd$
1	10 000	1 250	1 250	1 280	$\surd$
1	1 000 000	125 000	125 020	125 045	$\surd$
2	100	25	25	53	$\surd$
2	10 000	2 500	2 500	2 530	$\surd$
2	1 000 000	250 000	250 040	250 065	$\surd$
4	100	50	50	80	$\surd$
4	10 000	5 000	5 000	5 030	$\surd$
4	1 000 000	500 000	500 080	500 105	$\surd$
8	100	100	100	130	$\surd$
8	10 000	10 000	10 000	10 030	$\surd$
8	1 000 000	1 000 000	1 000 155	1 000 180	$\surd$
12	100	150	150	180	$\surd$
12	10 000	15 000	15 000	15 030	$\surd$
12	1 000 000	1 500 000	1 500 230	1 500 255	$\surd$
16	100	200	200	230	$\surd$
16	10 000	20 000	20 000	20 030	$\surd$
16	1 000 000	2 000 000	2 000 310	2 000 335	$\surd$
20	100	250	250	280	$\surd$
20	10 000	25 000	25 000	25 030	$\surd$
20	1 000 000	2 500 000	2 500 385	2 500 410	$\surd$

Tabelle 6: Untersuchung der Kompressionsraten der von BBS erzeugten Zufallszahlen.

Bei der Betrachtung von Tabelle 6 fällt auf, dass die Informationsmenge der komprimierten Zufallszahlen nie kleiner als die der unkomprimierten Zahlen ist. Das bedeutet, dass sich die von BBS erzeugten Zufallszahlen nicht komprimieren lassen. Also haben die Zufallszahlen den Test auf Unabhängigkeit bestanden.

5.3 RC4

RC4 ist ein Zufallszahlengenerator, der von Ron Rivest[10] entwickelt wurde[11]. Die Erzeugung von Zufallszahlen mit RC4 ist sehr effizient, und deshalb ist dieser Generator zur Stromchiffrierung (vgl. Abschnitt 2.4) geeignet. RC4 steht für „Ron's Code 4“ oder „Rivest's Cipher 4“, wobei die 4 andeutet, dass es noch weitere Stromchiffrierungen von Ron Rivest gibt.

Im Folgenden wird die Darstellung aus [Schneier 96, S. 455f.] auf eine variable Ausgabelänge verallgemeinert. Diese variable Ausgabelänge n, mit der der Zufallsgenerator arbeitet, kann beliebig groß sein. Sie gibt die Anzahl der Bits an, die der Zufallsgenerator bei der Ausführung parallel ausgibt. Üblich sind 8 oder 16 Bit, allerdings sind für die Implementierung auf Chipkarten 4 Bit besser geeignet, wie später gezeigt werden wird.

Alle Werte, die zur Berechnung einer Zufallszahl benötigt werden, sind aus der Gruppe $\mathcal{GF}(2^n)$. Als Rechenoperation der Gruppe wird die Addition benutzt.

Beschreibung des Verfahrens

Für die Initialisierung von RC4 wird ein Schlüssel benötigt. Dieser Schlüssel besteht aus 2^n Werten K_0, $K_1, \ldots, K_{2^n-1}$, die Werte aus $\mathcal{GF}(2^n)$ annehmen dürfen. Der Algorithmus benutzt zur Erzeugung eine Substitutions-Box (S-Box), die die 2^n Werte $S_0, S_1, \ldots, S_{2^n-1}$ enthält, die ebenfalls aus $\mathcal{GF}(2^n)$ sind. Sie sei anfangs mit den Werten $S_0 = 0, S_1 = 1, \ldots, S_{2^n-1} = 2^n - 1$ gefüllt. Zusätzlich gebe es die Indizes i und j.

Dann werden folgende Instruktionen ausgeführt:

$j \leftarrow 0$
for $i = 0$ **to** $2^n - 1$ **do**
  $j \leftarrow (j + S_i + K_i) \bmod 2^n$
  vertausche S_i und S_j
end for
$i \leftarrow 0$
$j \leftarrow 0$

Durch diese Schleife werden die Elemente der S-Box 2^n-mal permutiert. Jedes Element der S-Box ist mindestens einmal an einer Permutation beteiligt, weil in der Vertauschoperation der Schleifenzähler i als Index S_i benutzt wird.

Für die Erzeugung der ersten Zufallszahl müssen die Indizes i und j mit 0 initialisiert werden, was in den letzten beiden Programmzeilen geschieht.

Um eine Zufallszahl der Länge n zu erzeugen, werden folgende Rechenoperationen durchgeführt:

$i \leftarrow (i + 1) \bmod 2^n$
$j \leftarrow (j + S_i) \bmod 2^n$
vertausche S_i und S_j
$t \leftarrow (S_i + S_j) \bmod 2^n$
gib aus S_t

Die ersten 3 Operationen entsprechen der Permutation bei der Initialisierung, allerdings mit zwei Unterschieden: Erstens wird der Schleifenzähler i beim Erreichen von 2^n wieder auf 0 gesetzt (Inkrement in $\mathcal{GF}(2^n)$). Zweitens wird der Schlüssel K_i nicht mehr berücksichtigt.

[10]Ron Rivest ist neben Adi Shamir und Leonard Adleman einer der Entwickler des RSA-Verschlüsselungsverfahrens.
[11]RC4 ist von RSA Data Security Inc. patentiert, und die Benutzung in eigenen Programmen bedarf der Lizenzierung.

Die letzten beiden Operationen wählen mithilfe der Variablen t einen Wert aus der S-Box aus und geben diesen als Zufallszahl aus. Der ausgegebene Wert ist also n Bit lang. Da die S-Box alle Werte 0 bis 2^{n-1} genau einmal enthält, sind alle 2^n Kombinationen aus n Bit als Ausgabe möglich.

Parametrisierung des Verfahrens für Chipkarten

Für die Implementierung von RC4 auf Chipkarten muss die Ausgabelänge n sinnvoll festgelegt werden. Es muss ein Kompromiss zwischen notwendigem Speicherbedarf und Sicherheit des Zufallsgenerators gefunden werden.

Der Speicherbedarf hängt von der Größe der S-Box ab. Er ist $P = n \cdot (2^n + 3)$ Bit, denn es werden 2^n Werte in der S-Box gespeichert, es wird Platz für die Variablen i, j und t benötigt, und jeder dieser Werte ist n Bit lang. Die Sicherheit, also die Periode des Zufallsgenerators, hängt nicht von der Wahl des Schlüssels ab, denn RC4 besitzt nach [Schneier 96, S. 456] unabhängig von der Wahl des Schlüssels keine kleinen Zyklen. Aus demselben Grund ist die Periode in der Grössenordnung der Anzahl der Zustände des Zufallsgenerators. Der Generator kann genau $Z = 2^n! \cdot (n^2)^2$ verschiedene Zustände annehmen, denn es gibt $2^n!$ viele Permutationen der S-Box und jeweils n^2 Werte, die die Variablen i und j annehmen können.

n	Speicherbedarf [Bit]	Anzahl Zustände
1	5	2
2	14	384
3	33	3 265 920
4	76	5 356 234 211 328 000
5	175	$\approx 3 \cdot 10^{38}$
6	402	$\approx 5 \cdot 10^{92}$
7	917	$\approx 4 \cdot 10^{215}$
8	2 072	$\approx 1 \cdot 10^{512}$

Tabelle 7: Speicherbedarf und Anzahl der Zustände des RC4-Generators in Abhängigkeit der Ausgabebreite n

In Tabelle 7 sind für die Ausgabebreiten 1 bis 8 Bit Speicherbedarf und Anzahl der Zustände angegeben. Zur Implementation auf Chipkarten ist eine Ausgabebreite von 4 Bit am sinnvollsten, weil nur 76 Bit (9,5 Bytes) Speicherplatz benötigt werden, aber trotzdem eine sehr große Periode gewährleistet ist. Ein Test der Periode ergab, dass sie mindestens 3 500 000 000 000 beträgt[12].

In der Praxis ist allerdings gefordert, dass die Zufallszahlen länger als 4 Bit sind. Um das zu erreichen, wird der RC4-Generator mehrfach gestartet, und die 4-Bit-Zahlen, die er erzeugt, werden zu einer längeren Zufallszahl zusammengesetzt. Am einfachsten ist es, Vielfache von 4 als Breite der zusammengesetzten Zufallszahl zu wählen, weil dann die Ausgabe sehr einfach umgesetzt werden kann.

Untersuchung der erzeugten Zufallszahlen auf Gleichverteilung

Es sollen nun die Zufallszahlen, die vom RC4-Generator erzeugt werden, wie in Kapitel 4.4 beschrieben auf Gleichverteilung untersucht werden. Zum Erzeugen der Zufallszahlen wird die in Kapitel 6.4 vorgestelle Implementation von RC4 in der Programmiersprache C benutzt. Das Ergebnis der Untersuchungen ist in Tabelle 8 (Seite 45) zusammengefasst. Der Generator erzeugt jeweils 4 Bit lange Zufallszahlen; längere Zufallszahlen (8, 12, 16 und 20 Bit) wurden durch Aneinanderreihen mehrerer 4-Bit-Zahlen erzeugt. Für alle Ausgabelängen 4, 8, 12, 16 und 20 Bit wurden jeweils 100, 10 000, 1 000 000, 10 000 000 und 50 000 000 Zufallszahlen erzeugt (Tabellenspalten 1 und 2). Die erzeugten Zufallszahlen wurden den Statistiktests unterzogen, deren Ergebnisse in den Spalten der Tabelle 8 angegeben sind. Der Chi-Quadrat-Test zur Untersuchung auf Gleichverteilung der bitweisen Wahrscheinlichkeit ist in den Spalten 3 und 4 gezeigt; dabei gibt Spalte 3 den Wert von χ^2 an und Spalte 4, ob der Test mit dem berechneten Wert χ^2 bestanden wurde ($\surd$ bedeutet „bestanden“, $\times$ bedeutet „nicht bestanden“). Der Anpassungstest der bitweisen Wahrscheinlichkeit an eine Normalverteilung ist in den Spalten 5 bis 9 zu finden. Angegeben sind dort der Erwartungswert EN und die Standardabweichung σ_N der theoretischen Binomialverteilung, das empirische Stichprobenmittel $\overline{X}$ und die

[12] Um das festzustellen, rechnete ein 500 MHz-Rechner 50 Stunden lang; um alle Zustände des Generators zu durchlaufen, hätte er etwa 88 000 Stunden rechnen müssen.

empirische Stichprobenstandardabweichung σ_S der Verteilung der Zufallszahlen sowie wieder eine Angabe, ob der Test bestanden worden ist. Das Ergebnis des Chi-Quadrat-Tests der blockweisen Wahrscheinlichkeiten auf Gleichverteilung ist in den Spalten 10 und 11 abgedruckt; Spalte 10 zeigt den Wert von χ^2, und in Spalte 11 ist wiederum angegeben, ob der Test bestanden wurde.

Bitzahl	Anzahl	χ^2 (Bit)	Ok?	EN	σ_N	$\overline{X}$	σ_S	Ok?	χ^2 (Block)	Ok?
4	100	3,867	✓	2	1,000	1,920 ± 0,326	0,992	✓	7,520	✓
4	10 000	2,079	✓	2	1,000	2,007 ± 0,033	0,999	✓	18,362	✓
4	1 000 000	3,017	✓	2	1,000	2,001 ± 0,003	1,000	✓	15,678	✓
4	10 000 000	3,272	✓	2	1,000	2,001 ± 0,001	1,000	✓	10,809	✓
4	50 000 000	1,208	✓	2	1,000	2,000 ± 0,001	1,000	✓	15,950	✓
8	100	1,723	✓	4	1,414	4,060 ± 0,477	1,448	✓	263,520	✓
8	10 000	8,743	✓	4	1,414	4,010 ± 0,046	1,388	✓	276,608	✓
8	1 000 000	30,047	×	4	1,414	4,003 ± 0,005	1,414	✓	267,371	✓
8	10 000 000	239,157	×	4	1,414	4,001 ± 0,002	1,415	✓	698,327	✓
8	50 000 000	1258,277	×	4	1,414	4,000 ± 0,001	1,415	✓	2 820,727	✓
12	100	15,077	✓	6	1,732	6,130 ± 0,602	1,829	✓	4 405,600	✓
12	10 000	24,129	×	6	1,732	6,019 ± 0,057	1,727	✓	4 009,139	✓
12	1 000 000	37,232	×	6	1,732	6,005 ± 0,006	1,733	✓	4 190,958	✓
12	10 000 000	211,857	×	6	1,732	6,001 ± 0,002	1,733	✓	5 196,444	✓
12	50 000 000	930,820	×	6	1,732	6,000 ± 0,001	1,733	✓	9 369,948	✓
16	100	9,423	✓	8	2,000	8,200 ± 0,678	2,060	✓	65 436,000	✓
16	10 000	9,524	✓	8	2,000	8,022 ± 0,065	1,975	✓	65 548,150	✓
16	1 000 000	29,103	×	8	2,000	8,004 ± 0,007	2,002	✓	66 068,345	✓
16	10 000 000	123,543	×	8	2,000	8,001 ± 0,002	2,000	✓	67 909,306	✓
16	50 000 000	572,012	×	8	2,000	8,000 ± 0,001	2,000	✓	77 487,208	✓
20	100	6,516	✓	10	2,236	10,220 ± 0,676	2,053	✓	1 048 576,000	✓
20	10 000	10,904	✓	10	2,236	10,028 ± 0,073	2,230	✓	1 046 545,178	✓
20	1 000 000	45,481	×	10	2,236	10,006 ± 0,007	2,238	✓	1 046 807,769	✓
20	10 000 000	140,678	×	10	2,236	10,001 ± 0,002	2,237	✓	1 050 521,985	✓
20	50 000 000	553,078	×	10	2,236	10,000 ± 0,001	2,237	✓	1 060 015,985	✓

Tabelle 8: Untersuchung von Zufallszahlen, die mit dem RC4-Generator mit der Ausgabebreite 4 Bit erzeugt wurden. Zum Erzeugen längerer Zufallszahlen wurde der Generator mehrfach nacheinander ausgeführt. Es wurde immer der Standardschlüssel $K_0 = 0$, $K_1 = 1, \ldots, K_{15} = 15$ verwendet. Das Signifikanzniveau der χ^2-Quadrat-Verteilung wurde auf $\alpha = 0,95$ festgelegt; die Quantile der χ^2-Verteilung wurden [Hübner 96, S. 197] entnommen.

In Tabelle 8 fällt auf, dass der RC4-Generator den Test auf die blockweise Gleichverteilung und den Anpassungstest der bitweisen Wahrscheinlichkeiten an die Normalverteilung besteht, aber den Chi-Quadrat-Test der bitweisen Gleichverteilung nicht. Wenn die Ausgabelänge größer als 4 ist und mehr als 10 000 Zufallszahlen erzeugt werden, wird der Wert von χ^2 größer als das entsprechende Quantil der Chi-Quadrat-Verteilung. Da in dieser Arbeit nur Untersuchungen im Low-Security-Bereich betrachtet werden, das heißt Anwendungen, bei denen nur wenige Zufallszahlen erzeugt werden, fällt das Nichtbestehen des Tests ab 10 000 Zufallszahlen nicht ins Gewicht.

Um diesen Mangel am RC4-Generator dennoch zu beheben, ist eine Modifikation notwendig, die kurz erläutert werden soll. Der Test ergibt, dass die ausgegebenen Zufallszahlen nicht voneinander unabhängig sind, sobald mehrere nacheinander erzeugte 4-Bit-Zufallszahlen zu einer längeren Zufallszahl zusammengesetzt werden. Weiterhin ist aus Tabelle 8 zu entnehmen, daß der Test für die Ausgabelänge 4 immer bestanden wird. Also ist es ungünstig, mehrere 4-Bit-Zahlen, die nacheinander erzeugt wurden, zu einer neuen zusammenzusetzen. Die Lösung ist, mehrere RC4-Generatoren, die mit verschiedenen Schlüsseln initialisiert sind, parallel mehrere 4-Bit-Zufallszahlen erzeugen zu lassen und diese Zahlen zu einer langen Zufallszahl zusammenzusetzen. Dadurch ist gewährleistet, dass die Zahlen unabhängig voneinander sind. Der Nachteil ist, dass der Speicherbedarf auf ein Vielfaches ansteigt. Das Verfahren paralleler RC4-Generatoren wird in dieser Arbeit nicht weiter verfolgt, weil sie sich auf Low-Security-Anwendungen beschränkt.

Untersuchung der erzeugten Zufallszahlen auf Unabhängigkeit

Die von RC4 erzeugten Zufallszahlen werden nun auf ihre Unabhängigkeit geprüft. Dazu wird versucht, wie in Kapitel 4.5 beschrieben, sie mit den Packprogrammen `gzip` und `zip` zu komprimieren.

Es wurden dieselben Parameter des RC4-Generators gewählt wie zuvor bei der Untersuchung auf Gleichverteilung (siehe voriger Abschnitt), und es wurden wiederum jeweils 100, 10 000, 1 000 000, 10 000 000 und 50 000 000 Zahlen der Längen 4, 8, 12, 16 und 20 Bit erzeugt. Das Ergebnis der Untersuchung dieser Zahlen ist in Tabelle 9 dargestellt. Die Spalten 1 und 2 geben die eben genannte Bitlänge der Zahlen sowie die Anzahl der Zufallszahlen an. In Spalte 3 ist die unkomprimierte Größe der Zufallszahlen angegeben, die in einer Datei abgespeichert wurden. Diese Größe berechnet aus $\frac{\text{Bitzahl}\cdot\text{Anzahl}}{8}$. In den Spalten 4 und 5 ist die Größe der mit `zip` bzw. `gzip` komprimierten Daten angegeben, das heißt nicht die Größe der komprimierten Datei, sondern nur die der komprimierten Daten (ohne Overhead, der sich aus den Dateiformaten ergibt).

Bitzahl	Anzahl	unkompr. [Byte]	`zip` [Byte]	`gzip` [Byte]	Ok?
4	100	50	50	80	√
4	10 000	5 000	5 000	5 030	√
4	1 000 000	500 000	500 080	500 105	√
4	10 000 000	5 000 000	5 000 765	5 000 790	√
4	50 000 000	25 000 000	25 003 815	25 003 840	√
8	100	100	100	130	√
8	10 000	10 000	10 000	10 030	√
8	1 000 000	1 000 000	1 000 155	1 000 180	√
8	10 000 000	10 000 000	10 001 530	10 001 555	√
8	50 000 000	50 000 000	50 007 630	50 007 655	√
12	100	150	150	180	√
12	10 000	15 000	15 000	15 030	√
12	1 000 000	1 500 000	1 500 230	1 500 255	√
12	10 000 000	15 000 000	15 002 290	15 002 315	√
12	50 000 000	75 000 000	75 012 046	75 012 071	√
16	100	200	200	230	√
16	10 000	20 000	20 000	20 030	√
16	1 000 000	2 000 000	2 000 310	2 000 335	√
16	10 000 000	20 000 000	20 003 055	20 003 080	√
16	50 000 000	100 000 000	100 015 861	100 015 886	√
20	100	250	250	280	√
20	10 000	25 000	25 000	25 030	√
20	1 000 000	2 500 000	2 500 385	2 500 410	√
20	10 000 000	25 000 000	25 003 815	25 003 840	√
20	50 000 000	125 000 000	125 019 671	125 019 696	√

Tabelle 9: Untersuchung der Kompressionsraten beim Komprimieren von Zufallszahlen, die mit dem RC4-Generator der Ausgabebreite 4 Bit erzeugt wurden. Zum Erzeugen längerer Zufallszahlen wurde der Generator mehrfach nacheinander ausgeführt. Es wurde immer der Standardschlüssel $K_0 = 0$*,* $K_1 = 1, \ldots,$ $K_{15} = 15$ *verwendet.*

Bei der Betrachtung von Tabelle 9 fällt auf, dass die Informationsmenge der komprimierten Zufallszahlen nie kleiner als die der unkomprimierten Zahlen ist. Das bedeutet, dass sich die von RC4 erzeugten Zufallszahlen nicht komprimieren lassen. Also haben die Zufallszahlen den Test auf Unabhängigkeit bestanden.

5.4 Elliptische Kurven

Dieser Zufallszahlengenerator, abgekürzt als EC-Generator bezeichnet, nutzt die in Kapitel 3.2.2 beschriebenen elliptischen Kurven der Form $y^2 = x^3 + ax + b$ über $\mathcal{GF}(p)$. Die Berechnungen werden in der projektiven Koordinatendarstellung durchgeführt, um die Division zu vermeiden. Als Startwert wird ein Punkt $P_0(x_0, y_0, z_0)$ der elliptischen Kurve gewählt. Dabei ist $z_0 = 1$, weil mit einem Punkt in affiner Darstellung initialisiert wird.

Die rekursive Definition der fortgesetzten Punktverdopplung bzw. der Addition eines Punktes zu sich selbst lautet dann:

$$\begin{aligned} (x_1, y_1, z_1) &= 2 \cdot (x_0, y_0, z_0) \\ (x_n, y_n, z_n) &= 2 \cdot (x_{n-1}, y_{n-1}, z_{n-1}) \quad \text{mit } n > 0 \end{aligned}$$

Die Zufallszahlensequenz besteht dann aus der Folge der x_n oder der Folge von Bitblöcken einer festen Breite w, die aus den x_n erzeugt werden. Sei Beispielsweise $x_n = 10111011$ und die Breite des Bitblocks mit $w = 5$ definiert, dann werden die 5 niederwertigsten Bits benutzt.

Problematisch ist bei diesem Verfahren die Beurteilung der Periode, die hier auftritt. Da statt der fortgesetzten Punktaddition die Punktverdopplung benutzt wird, gilt der Satz von Langrange (siehe Kapitel 3.2.1) nicht mehr, und es ist in Abhängigkeit von der Kurvenordnung keine einfache Aussage mehr zu treffen, welche Größe die Periode besitzt. Auch wird diese Untersuchung erschwert, da hier mit einer uneindeutigen Darstellung gearbeitet wird. Das bedeutet, selbst wenn der Ausgangspunkt erneut erreicht wird, ist damit noch keine Periode erkennbar, da die ausgegebenen Werte sich unterscheiden, auch wenn sie den gleichen Punkt beschreiben. Da hier von den drei Werten jeweils nur einer zur Ausgabe benutzt wird und die anderen unbekannt sind und zusätzlich eine nichteindeutige Darstellung benutzt wird, ist es im Low-Security-Bereich eine hinnehmbare Einschränkung, dass hier die theoretischen Grundlagen nicht vorliegen. Insofern wäre auch eine numerische Untersuchung durch ein Trustcenter denkbar, welches dann für gestiegene Anforderungen die Qualität der Zufallszahlen garantiert.

Untersuchung der erzeugten Zufallszahlen auf Gleichverteilung

Es sollen nun die Zufallszahlen, die vom EC-Generator erzeugt werden, wie in Kapitel 4.4 beschrieben auf Gleichverteilung untersucht werden. Zum Erzeugen der Zufallszahlen wird die in Kapitel 6.5 vorgestelle Implementation von EC in der Programmiersprache C benutzt.

Bitzahl	Anzahl	χ^2 (Bit)	Ok?	EN	σ_N	$\overline{X}$	σ_S	Ok?	χ^2 (Block)	Ok?
1	100	0,154	√	0,5	0,500	0,481 ± 0,162	0,502	√	0,154	√
1	1 000	0,144	√	0,5	0,500	0,494 ± 0,052	0,500	√	0,144	√
1	1 847	0,009	√	0,5	0,500	0,499 ± 0,038	0,500	√	0,009	√
2	100	0,380	√	1,0	0,707	0,990 ± 0,227	0,689	√	0,400	√
2	1 000	2,136	√	1,0	0,707	0,974 ± 0,073	0,697	√	2,528	√
2	1 847	1,216	√	1,0	0,707	0,990 ± 0,054	0,700	√	1,429	√
4	100	5,360	√	2,0	1,000	2,140 ± 0,307	0,932	√	18,080	√
4	1 000	3,848	√	2,0	1,000	2,043 ± 0,102	0,978	√	13,440	√
4	1 847	1,833	√	2,0	1,000	1,971 ± 0,077	1,007	√	12,970	√
8	100	6,478	√	4,0	1,141	4,180 ± 0,457	1,388	√	222,560	√
8	1 000	8,557	√	4,0	1,141	3,955 ± 0,143	1,377	√	263,616	√
8	1 847	3,703	√	4,0	1,141	3,959 ± 0,108	1,416	√	235,235	√
12	100	8,912	√	6,0	1,732	6,100 ± 0,574	1,744	√	3 996,000	√
12	1 000	17,444	√	6,0	1,732	5,933 ± 0,182	1,745	√	4 144,576	√
12	1 847	19,176	√	6,0	1,732	5,935 ± 0,135	1,765	√	4 102,332	√

Tabelle 10: Untersuchung von Zufallszahlen, die mit dem EC-Generator durch Ausgabe der x-Koordinate *erzeugt wurden. Als Kurve wurde* $y^2 = x^3 + 2155 \cdot x + 16008$ *über* $\mathcal{GF}(33\,013)$ *mit dem Startpunkt* $P_0(x_0; y_0) = (3; 150)$ *gewählt. Die Kurve besitzt die prime Ordnung 33247, und mit dem Startpunkt wird eine Periode von 1847 erreicht. Das Signifikanzniveau der* χ^2*-Quadrat-Verteilung wurde auf* $\alpha = 0,95$ *festgelegt; die Quantile der* χ^2*-Verteilung wurden [Hübner 96, S. 197] entnommen.*

Das Ergebnis der Untersuchungen ist in den Tabellen 10 und 11 zusammengefasst. Der Generator erzeugt Zufallszahlen der Länge 1, 2, 4, 8 und 12 Bit. Für alle Ausgabelängen 1, 2, 4, 8 und 12 Bit wurden jeweils 100, 1 000 und 1 847 Zufallszahlen erzeugt (Tabellenspalten 1 und 2). Es wurde als obere Grenze 1 847 Zahlen für den Statistiktest gewählt, da dann wieder der Ausgangspunkt erreicht wird. Dazu wurden für die Tabelle 10 die x-Koordinaten und für die Tabelle 11 die y-Koordinaten benutzt. Die erzeugten Zufallszahlen wurden den Statistiktests unterzogen, deren Ergebnisse in den Spalten der Tabellen 10 und 11 angegeben sind. Der Chi-Quadrat-Test zur Untersuchung auf Gleichverteilung der bitweisen Wahrscheinlichkeit ist in den Spalten 3 und 4 gezeigt; dabei gibt Spalte 3 den Wert von χ^2 an und Spalte 4, ob der Test mit dem berechneten Wert χ^2 bestanden wurde (√ bedeutet „bestanden“, × bedeutet „nicht bestanden“). Der Anpassungstest der bitweisen Wahrscheinlichkeit an eine Normalverteilung ist in den Spalten 5 bis 9 zu finden. Angegeben sind dort der Erwartungswert EN und die Standardabweichung σ_N der theoretischen Binomialverteilung, das empirische Stichprobenmittel $\overline{X}$ und die empirische Stichprobenstandardabweichung σ_S der Verteilung der Zufallszahlen sowie wieder eine Angabe, ob der Test bestanden worden ist. Das Ergebnis des Chi-Quadrat-Tests der blockweisen Wahrscheinlichkeiten auf Gleichverteilung ist in den Spalten 10 und 11 abgedruckt; Spalte 10 zeigt den Wert von χ^2, und in Spalte 11 ist wiederum angegeben, ob der Test bestanden wurde.

Die erzeugten Zufallszahlen, die mit den x-Koordinaten erzeugt wurden, bestehen in allen benutzten Ausgabebreiten den Statistiktest. Bei den Zufallszahlen, die mittels der y-Koordinate generiert wurden, fällt auf, dass die Zahlen bei der 1-Bit-Ausgabe bei einer Anzahl von 1 000 erzeugten Zufallszahlen durchfallen.

Bitzahl	Anzahl	χ^2 (Bit)	Ok?	EN	σ_N	$\overline{X}$	σ_S	Ok?	χ^2 (Block)	Ok?
1	100	0,154	$\surd$	0,5	0,500	0,519 ± 0,162	0,502	$\surd$	0,154	$\surd$
1	1 000	7,744	×	0,5	0,500	0,544 ± 0,052	0,498	$\surd$	7,744	×
1	1 847	4,987	$\surd$	0,5	0,500	0,526 ± 0,038	0,500	$\surd$	4,987	$\surd$
2	100	2,720	$\surd$	1,0	0,707	0,920 ± 0,246	0,748	$\surd$	7,840	$\surd$
2	1 000	3,776	$\surd$	1,0	0,707	1,040 ± 0,073	0,698	$\surd$	8,384	$\surd$
2	1 847	2,566	$\surd$	1,0	0,707	1,025 ± 0,054	0,703	$\surd$	5,165	$\surd$
4	100	4,947	$\surd$	2,0	1,000	1,970 ± 0,308	0,937	$\surd$	25,440	$\surd$
4	1 000	3,743	$\surd$	2,0	1,000	1,958 ± 0,102	0,975	$\surd$	11,104	$\surd$
4	1 847	4,994	$\surd$	2,0	1,000	2,012 ± 0,075	0,973	$\surd$	17,489	$\surd$
8	100	12,146	$\surd$	4,0	1,141	4,180 ± 0,457	1,389	$\surd$	278,880	$\surd$
8	1 000	5,843	$\surd$	4,0	1,141	3,993 ± 0,148	1,421	$\surd$	234,432	$\surd$
8	1 847	3,473	$\surd$	4,0	1,141	3,989 ± 0,108	1,411	$\surd$	253,531	$\surd$
12	100	17,410	$\surd$	6,0	1,732	6,270 ± 0,569	1,728	$\surd$	4 159,840	$\surd$
12	1 000	14,650	$\surd$	6,0	1,732	6,002 ± 0,182	1,749	$\surd$	3 997,120	$\surd$
12	1 847	7,473	$\surd$	6,0	1,732	6,002 ± 0,133	1,734	$\surd$	3 974,332	$\surd$

Tabelle 11: Untersuchung von Zufallszahlen, die mit dem EC-Generator durch Ausgabe der y-Koordinate *erzeugt wurden. Als Kurve wurde* $y^2 = x^3+2155\cdot x+16008$ *über* $\mathcal{GF}(33\,013)$ *mit dem Startpunkt* $P_0(x_0; y_0) = (3; 150)$ *gewählt. Die Kurve besitzt die prime Ordnung 33 247, und mit dem Startpunkt wird eine Periode von 1847 erreicht. Das Signifikanzniveau der* χ^2*-Quadrat-Verteilung wurde auf* $\alpha = 0,95$ *festgelegt; die Quantile der* χ^2*-Verteilung wurden [Hübner 96, S. 197] entnommen.*

Bitzahl	Anzahl	unkompr. [Byte]	`zip` [Byte]	`gzip` [Byte]	Ok?
1	100	13	13	41	$\surd$
1	1 000	125	125	155	$\surd$
1	1 847	231	231	261	$\surd$
2	100	25	25	41	$\surd$
2	1 000	250	250	280	$\surd$
2	1 847	462	462	492	$\surd$
4	100	50	50	80	$\surd$
4	1 000	500	500	530	$\surd$
4	1 847	924	924	954	$\surd$
8	100	100	100	130	$\surd$
8	1 000	1 000	1 000	1 030	$\surd$
8	1 847	1 847	1 847	1 877	$\surd$
12	100	150	150	180	$\surd$
12	1 000	1 500	1 500	1 530	$\surd$
12	1 847	2 771	2 771	2 801	$\surd$

Tabelle 12: Untersuchung der Kompressionsraten beim EC-Generator. Da sich die Kompressionsraten bei der Ausgabe von x- und y-Koordinate an keiner Stelle unterscheiden, sind sie in einer einzigen Tabelle zusammengefasst.

Untersuchung der erzeugten Zufallszahlen auf Unabhängigkeit

Die vom EC-Generator erzeugten Zufallszahlen werden nun auf ihre Unabhängigkeit geprüft. Dazu wird versucht, wie in Kapitel 4.5 beschrieben, sie mit den Packprogrammen `gzip` und `zip` zu komprimieren.

Es wurden dieselben Parameter des EC-Generators gewählt wie zuvor bei der Untersuchung auf Gleichverteilung (siehe voriger Abschnitt), und es wurden wiederum jeweils 100, 1 000 und 1 847 Zahlen der Längen 1, 2, 4, 8 und 12 Bit erzeugt. Das Ergebnis der Untersuchung dieser Zahlen ist in Tabelle 12 dargestellt. Da sich die Werte für die Zahlen, die durch Ausgabe der x- bzw. der y-Koordinate erzeugt wurden, nicht unterscheiden, sind sie in einer Tabelle zusammengefasst. Die Spalten 1 und 2 geben die eben genannte Bitlänge der Zahlen sowie die Anzahl der Zufallszahlen an. In Spalte 3 ist die unkomprimierte Größe der Zufallszahlen angegeben, die in einer Datei abgespeichert wurden. Diese Größe berechnet aus $\frac{\text{Bitzahl}\cdot\text{Anzahl}}{8}$. In den Spalten 4 und 5 ist die Größe der mit `zip` bzw. `gzip` komprimierten Daten angegeben, das heißt nicht die Größe der komprimierten Datei, sondern nur die der komprimierten Daten (ohne Overhead, der sich aus den Dateiformaten ergibt).

Bei der Betrachtung von Tabelle 12 fällt auf, dass die Informationsmenge der komprimierten Zufallszahlen nie kleiner als die der unkomprimierten Zahlen ist. Das bedeutet, dass sich die vom EC-Generator erzeugten Zufallszahlen nicht komprimieren lassen. Also haben die Zufallszahlen den Test auf Unabhängigkeit bestanden.

6 Implementierung der Zufallsgeneratoren auf Chipkarten

In diesem Kapitel werden die Implementierungen der Zufallsgeneratoren aus Kapitel 5 vorgestellt. Jeder der Generatoren wurde in den Programmiersprachen C und VHDL programmiert. In Kapitel 6.1 werden die Eigenschaften vorgestellt, die die Implementationen der Zufallsgeneratoren gemeinsam vorweisen. In den darauf folgenden Kapiteln 6.2 bis 6.5 werden die verschiedenen Implementationen im einzelnen vorgestellt. Zusätzlich werden die Eigenschaften der Zufallsgeneratoren untersucht, die sich aus der Implementation ergeben. Das sind die Antwortzeit, der Speicherbedarf und die Leistungsaufnahme.

6.1 Gemeinsamkeiten der Implementierungen

Alle in Kapitel 5 vorgestellten Zufallsgeneratoren wurden zweimal implementiert: Einmal in der Programmiersprache C und einmal in VHDL[13]. Diese doppelte Implementierung wurde aus folgendem Grund durchgeführt: Zum ausführlichen Analysieren der Zufallsgeneratoren und zum Erstellen von Statistiken ist es günstig, die Generatoren auf einem Rechner in einer Hochsprache vorliegen zu haben. Andererseits sollen sie in Hardware umgesetzt werden, und deshalb ist die Implementierung in einer Hardwarebeschreibungssprache dringend notwendig. Erst dadurch ist es möglich, den Hardware- und Zeitaufwand der Zufallsgeneratoren abzuschätzen.

Die Implementierungen in C

Zum Analysieren der Zufallsgeneratoren wurde die Programmiersprache C gewählt, weil sie zum einen die notwendigen Bitoperationen unterstützt und zum anderen portabel ist.

Als C-Compiler wurde `gcc`, der GNU-C-Compiler, benutzt, und die Programme wurden unter den Betriebssystemen Linux, Solaris und AmigaOS erfolgreich kompiliert. Da auf graphische Benutzungsoberflächen verzichtet wurde und nur die Bibliotheken aus dem ANSI-C-Sprachstandard benutzt wurden, ist die Portierung zwischen verschiedenen Betriebssystemen problemlos möglich.

Bei der Programmierung wurde darauf geachtet, dass die Zufallsgeneratoren frei parametrisierbar sind. Das bedeutet, dass alle Eigenschaften – auch die Anzahl der Ausgabebits – einfach änderbar sind. Diese Möglichkeit zum Einstellen der Parameter ist zum Analysieren der Zufallszahlen sehr bedeutsam. Die Einstellungen werden jeweils direkt im Quellcode im Definitionsabschnitt des Zufallsgenerators vorgenommen. Dieses Vorgehen hat im Gegensatz zu eigenen Konfigurationsdateien den Vorteil, dass die Änderungen schneller und einfacher durchgeführt werden können.

Im Folgenden sollen die Parameter vorgestellt werden, die bei allen Zufallsgeneratoren eingestellt werden können.

- `#define ANZAHL 10` — Legt die Anzahl der Zufallszahlen fest, die erzeugt werden sollen.
- `#define TESTE_PERIODE true` — Boolescher Wert, der besagt, ob der Zufallsgenerator auf eine Periode untersucht werden soll. Wenn `true` eingestellt ist, wird nach jedem Erzeugen einer Zufallszahl getestet, ob der Zustand des Zufallsgenerators gleich dem Anfangszustand ist, und der Benutzer wird informiert, wenn eine Periode gefunden wird.
- `#define DATEIAUSGABE true` — Festlegung, ob die Zufallszahlen in einer Datei abgespeichert werden sollen. Die erzeugte Zufallszahlendatei kann mit dem in Kapitel 4.4 vorgestellten Statistikprogramm getestet werden, oder es kann versucht werden, sie wie in Kapitel 4.5 beschrieben mit einem Packprogramm zu komprimieren.
- `#define DATEINAME "datei.zz"` — Hier kann der Dateiname der zu erzeugenden Datei eingestellt werden.

[13] VHDL steht für „Very High Speed Integrated Circuit Hardware Description Language“ und ist eine Beschreibungssprache für Hardwarestrukturen.

➪ `#define KONSOLE false` — Einstellung, ob die Zufallszahlen auf der Konsole ausgegeben werden sollen. Das ist nur dann möglich, wenn auch die Ausgabe in eine Datei (siehe oben) eingestellt ist.

➪ `#define ZUSTANDLOG true` — Mit diesem Parameter wird eingestellt, ob nach jedem Rechenschritt der Zustand des Generators in eine Datei geschrieben werden soll. Das ist notwendig, um die kippenden Bits beim Erzeugen einer Zufallszahl zu analysieren. Das Dateiformat sieht für jeden Zufallsgenerator anders aus; es ist in den C-Programmen jeweils dokumentiert. Die Rechenschritte werden in C genauso durchgeführt wie in den Hardwarebeschreibungen, um die Aussagekraft der Tests kippender Bits zu gewährleisten.

➪ `#define LOGDATEINAME "zustand.log"` — Hier wird die Datei angegeben, in die die Zustände gespeichert werden sollen.

Die Implementierungen in VHDL

Die Implementierung in VHDL ist erforderlich, um die Zufallsgeneratoren in Hardware zu beschreiben. Diese Hardwarebeschreibung kann dazu benutzt werden, um die Generatoren auf Chipkarten zu verwenden. Im Rahmen dieser Studienarbeit wurden die in VHDL beschriebenen Zufallsgeneratoren erfolgreich auf ein FPGA („Field Programmable Gate Array“) abgebildet. Als Compiler und Simulator kam `SYNOPSYS` zum Einsatz, und die Untersuchung der Schaltung wurde mit `CADENCE` durchgeführt.

Die Zufallsgeneratoren wurden so implementiert, dass sie von der Ansteuerung alle das gleiche Verhalten aufweisen. Wie in Abb. 15 dargestellt, verfügen die Zufallsgeneratoren alle über den Eingang „nreset“ zum Initialisieren des Zufallsgenerators, den Takteingang „clk“ und das Eingangssignal „start“, um eine neue Zufallszahl anzufordern. Wenn eine Zufallszahl fertig berechnet ist, ist der Ausgang „zufallszahl_fertig“ für einen Takt auf High, und gleichzeitig liegt am Ausgang „zufallszahl“ die Zufallszahl. Zu allen anderen Zeitpunkten sind die Ausgänge auf Low, das heißt es nicht möglich, aus den Signalen an den Ausgängen Rückschlüsse über den Zustand des Zufallsgenerators zu ziehen.

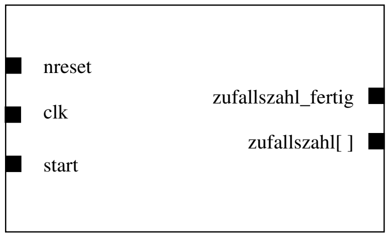

Abbildung 15: Anschlüsse, die die Implementationen der Zufallsgeneratoren in VHDL aufweisen. Links sind die Eingänge dargestellt und rechts die Ausgänge.

Die Realisierung der Zufallsgeneratoren ist so, wie in Abb. 15 dargestellt und wie im Rahmen dieser Studienarbeit in VHDL programmiert wurde, noch nicht direkt auf Chipkarten zu übernehmen. Der Grund dafür ist, dass alle Generatoren von einer dauerhaften externen Spannungsquelle abhängig sind. Wenn diese nicht gegeben ist, geht der im Generator gespeicherte Zustand verloren, und er lässt sich nur noch über das Reset-Signal in einen definierten Zustand überführen. Dadurch wird allerdings der Zufallsgenerator wieder auf den Anfangswert zurückgesetzt und liefert demzufolge dieselben Zufallszahlen wie bei der ersten Initialisierung. Abhilfe für diese Problematik ist, dass der Zustand des Zufallsgenerators in einem nichtflüchtigen Speicher, beispielsweise einem EEPROM (siehe Kapitel 3.4.1, Abschnitt „Mikroprozessorkarten“), gespeichert wird. Da der Aufbau einer Chipkarte je nach Chipkartentyp und Hersteller variiert, wurde in dieser Studienarbeit die allgemeine Variante nach dem oben beschriebenen Verfahren implementiert, die sich an verschiedene Chipkarten anpassen lässt.

Bei der Implementierung wurden für jeden Zufallsgenerator drei Dateien erstellt: `generator_pkg.vhd`, `generator.vhd` und `tb_generator.vhd`. Die erste Datei, `generator_pkg.vhd`, enthält die Parameter des entsprechenden Zufallsgenerators. Als einziger Parameter, der für alle Generatoren gleichsam einstellbar ist, ist „`constant TIMEBASE: time := 100 ns;`“, der die Zeitauflösung angibt, die bei der Simulation verwendet werden soll. Die Datei `generator.vhd` enthält die eigentliche Implementierung des Zufallsgenerators, und `tb_generator.vhd` ist die Testumgebung des Generators, die zur Validierung benutzt wird.

6.2 LFSR

In diesem Kapitel werden die LFSR-Implementierungen beschrieben, die Zeit- und Speicherkomplexität des LFSR-Generators untersucht sowie Aussagen zur Leistungsaufnahme getroffen.

Implementierung von LFSR in C

Die Implementierung des LFSR-Generators in C bereitet keine Probleme, da die notwendigen Operationen und Datentypen zur Verfügung stehen (das Schieberegister und das Polynom werden jeweils als „unsigned long" realisiert). Auch hinsichtlich der Wertebereiche gibt es keine Probleme. Es sind LFSRs mit einem Polynomgrad von bis zu $n = 32$ möglich (auf 32-Bit-Rechnern).

Neben den Standard-Parametern, die jeder Zufallsgenerator bietet (siehe Kapitel 6.1, Abschnitt „Die Implementierungen in C"), sind folgende Einstellungen für den LFSR-Generator zusätzlich wählbar:

- `#define BITBREITE` — Angabe der Breite, die das Schieberegister in Bits haben soll.
- `#define ANFANGSZUSTAND` — Angabe, mit welchem Schieberegisterinhalt initialisiert wird.
- `#define POLYNOM` — Angabe des Polynoms, welches das Rückkopplungsschema beschreibt.
- `#define AUSGABEHOCH, -NIEDRIG` — Hier wird das Intervall der nebeneinanderliegenden Bits angegeben, aus denen die auszugebenden Zufallsbits genommen werden.
- `#define EINMAL_SCHIEBEN false` — Auswahl, ob zum Erzeugen einer Zufallszahl ein einziges Mal geschoben werden soll oder so oft wie die Zufallszahl breit ist.

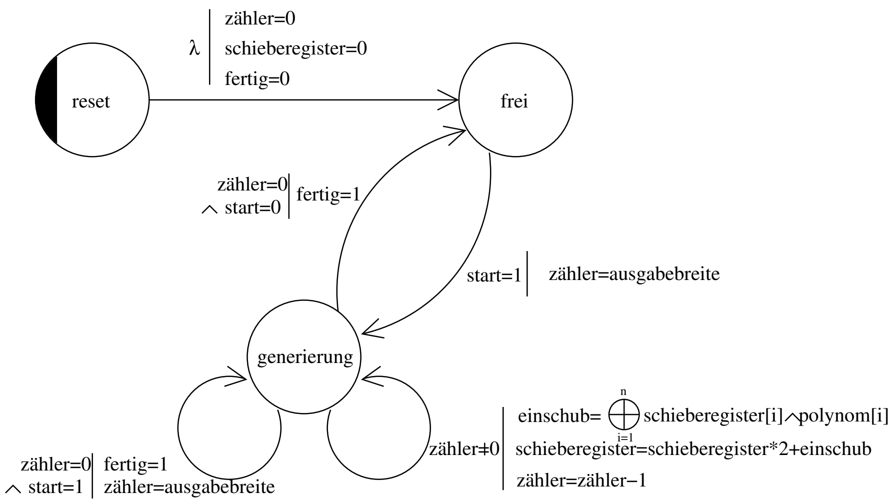

Abbildung 16: Zustandsdiagramm des Automaten, der in der VHDL-Implementation von LFSR beschrieben wird. Jede Transition ist mit der Bedingung für den Übergang (λ bei spontanem Übergang) und den Ausgabe-Wertzuweisungen, die beim Zustandswechsel durchgeführt werden, beschriftet.

Implementierung von LFSR in VHDL

Der LFSR-Generator ist auch in der VHDL-Implementation frei konfigurierbar. Die Einstellungen, die vorgenommen werden können, entsprechen denen der C-Implementierung, nur die Einstellmöglichkeit `EINMAL_SCHIEBEN` wurde weggelassen, weil die Tests ergaben, dass die Qualität der Zufallszahlen bei nur einmaligem Schieben zum Erzeugen einer Zufallszahl unbrauchbar ist. Zum Erzeugen einer n Bit langen Zufallszahl wird daher immer n-mal das Register geschoben.

Bei der Implemetation in VHDL wurden für das LFSR folgende Register definiert:

```
signal schieberegister:      std_logic_vector(BITBREITE-1 downto 0);
signal zaehler:              std_logic_vector(AUSGABEBREITELOG-1 downto 0);
signal fertig:               std_logic_vector(AUSGABEBREITE-1 downto 0);
```

Dabei ist das Schieberegister neben dem durch das Polynom beschriebene Rückkopplungsschema das zentrale Element zur Erzeugung der Zufallszahlen. Der Zähler dient zur Bestimmung, wie oft geschoben werden muss, bevor die Zufallszahl ausgegeben werden kann. Die Zufallszahl wird dabei direkt dem Schieberegister entnommen, so dass kein weiteres Ausgaberegister notwendig ist. Mit „fertig“ wird signalisiert, ob die Zufallszahl ausgegeben werden kann.

Der LFSR-Generator wurde – wie die anderen Zufallsgeneratoren auch – als Mealy-Automat realisiert. Das Zustandsdiagramm dieses endlichen Automaten ist in Abb. 16 dargestellt. Nach dem Reset geht der Generator in den „frei“-Zustand über, initialisiert das Schieberegister mit dem Startwert und wartet auf das „start“-Signal. Sobald das „start“-Signal anliegt, wechselt der Generator in den Zustand „generierung“ und setzt den „zaehler“ auf die „ausgabebreite“. Dann werden, solange der „zaehler“ ungleich Null ist, das neue Bit „einschub“ berechnet, die Bits im Schieberegister um eine Stelle verschoben und das neue Bit in das LFSR geschrieben. Sind der „zaehler“ und „start“ gleich Null, dann wird „fertig“ auf Eins gesetzt, und der Generator geht wieder in den Zustand „frei“ zurück. Liegt aber schon wieder ein „start“-Signal an, dann bleibt der Generator im Zustand „generierung“, der „zaehler“ wird auf die „ausgabebreite“ gesetzt, und es wird wiederum eine Zufallszahl erzeugt, ohne erst in den „frei“-Zustand zurückzuwechseln.

In Abb. 17 sind die Signale des LFSR-Generators bei der Initialisierung und dem Erzeugen einer Zufallszahl dargestellt.

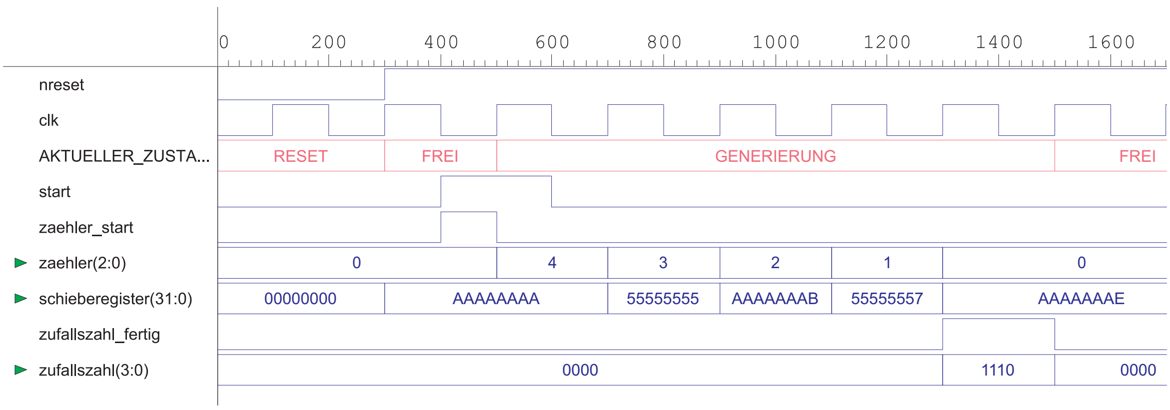

Abbildung 17: Signale des LFSR-Generators. Gezeigt sind die Initialisierung und das Erzeugen einer Zufallszahl.

Leistungsaufnahme

Zur Analyse der Leistungsaufnahme der VHDL-Implementation des LFSR-Generators wurden, wie in Kapitel 4.3 beschrieben, die kippenden Bits bei der Erzeugung einer neuen Zufallszahl untersucht.

Untersucht wurden dazu die Bits, die durch das Schieberegister geschoben werden. Da hierzu das Polynom $x^{30} + x^6 + x^4 + x^1 + x^0$ zur Beschreibung des LFSRs benutzt wurde, sind genau 30 wechselnde Bits als Höchstgrenze möglich. Als Startwert wurde $(2AAA\ AAAA)_{16}$ gewählt. Nach jeweils einem Takt, also nach dem Verschieben der Bits um eine Stelle, wurde die Anzahl der gewechselten Bits protokolliert. Insgesamt wurde dieser Vorgang 10 000 000 mal durchgeführt. Das Ergebnis der Untersuchung ist in Tabelle 13 dargestellt.

Die Anzahl der Bitwechsel befindet sich in der ersten Spalte, in der zweiten steht die gezählte Häufigkeit, und in der dritten Spalte stehen die Werte einer Normalverteilung, die an die Verteilung der gezählten Häufigkeiten angepasst ist.

Das Ergebnis der Untersuchung zeigt, dass die Anzahl der kippenden Bits zwischen 2 und 28 liegt. Im Mittelwert kippen $\overline{X} = 14.9990$ Bit (die Schätzgenauigkeit von $\overline{X}$ ist $0,0029$ Bit), und die empirische Standardabweichung ist $\sigma_S = 2.7389$ kippende Bits.

Für die Leistungsaufnahme der Hardware ist das Ergebnis, dass pro erzeugtem Bit im Mittel 14,999 Bits kippen, ein recht gutes Ergebnis. Für das Generieren von 20-Bit-Zufallszahlen ist unter Annahme der Unabhängigkeit die Anzahl der kippenden Bits also $20 \cdot 14,999 = 299,98$ Bits. Das LFSR eignet sich damit unter dem Aspekt der Leistungsaufnahme ohne weiteres für eine Implementierung auf einer Chipkarte.

Um diesen Wert noch zu verbessern, überlegt man, was passiert, wenn man ein kürzeres Schieberegister wählt. Der Mittelwert der kippenden Bits $\overline{X}$ ist offensichtlich direkt vom Polynomgrad n abhängig. Es gilt $\overline{X} \approx \frac{n}{2}$. Insofern ist es eine Überlegung wert, von welchem Grad das eingesetzte Polynom sein soll. Wenn der Polynomgrad n kleiner gewählt wird, dann sinkt auch die Leistungsaufnahme. Bei der Erzeugung einer 20-Bit-Zufallszahl mit einem LFSR, welches auf Grundlage eines Polynoms vom Grad $n = 20$ konstruiert wurde, sind im Mittel 200 kippende Bits zu erwarten.

Anzahl Wechsel	Häufigkeit	approx. Normalvert.
0	0	0.448
1	0	3.095
2	1	18.711
3	30	99.017
4	231	458.593
5	1279	1858.881
6	5421	6594.493
7	18861	20474.706
8	54862	55636.548
9	132946	132314.964
10	279483	275400.062
11	509491	501678.379
12	805648	799821.395
13	1118259	1116007.031
14	1356400	1362847.073
15	1444098	1456577.517
16	1351139	1362468.153
17	1113028	1115386.537
18	805378	799154.443
19	508876	501120.673
20	280336	275017.419
21	133528	132094.388
22	54527	55528.355
23	19023	20429.208
24	5536	6578.010
25	1324	1853.719
26	259	457.193
27	33	98.687
28	3	18.644
29	0	3.083
30	0	0.446

Tabelle 13: Untersuchung der kippenden Bits beim Erzeugen einer Zufallszahl. Erzeugt wurden 10 000 000 Zufallszahlen mit dem Polynom $x^{30}+x^6+x^4+x^1+x^0$ und dem Startwert $(2AAA\ AAAA)_{16}$ Das Stichprobenmittel ist $\overline{X} = 14.9990 \pm 0.0029$ Einsen, die empirische Stichprobenstandardabweichung ist $\sigma_S = 2.7389$ Einsen.

Komplexität

Die benötigten Flipflops entsprechen genau dem Grad des benutzten Polynoms und sind damit direkt von der gewünschten Periode abhängig. Weitere Register für temporäre Variablen sind nicht nötig. Auch Register zum Speichern der Zufallsbits sind nicht erforderlich, da die Zufallsbits direkt den Schieberegistern entnommen werden können. Damit hat das LFSR hinsichtlich des Speicherbedarfs den besten Wert unter den vorgestellten Generatoren.

Ein Akzelerator ist bei dem einfachen Aufbau des LFSRs nicht erforderlich.

Soll die vom Generator erzeugte Zufallszahlensequenz einen Wert m als Periode nicht unterschreiten, so hat das optimal gewählte Schieberegister eine Länge von $n = \lceil \log_2(m+1) \rceil$ Bits, das heißt das Polynom ist von diesem Grad, und auch die Anzahl der Flipflops im Register entspricht diesem Wert.

Das LFSR erzeugt halb so schnell wie der RC4-Generator (der schnellste der hier vorgestellten Generatoren) die Zufallsbits, da bei jedem Takt ein Bit erzeugt wird. Unter Annahme der Standardfrequenz von 3,5712 MHz produziert das LFSR also 3,5712 Millionen Zufallsbits und in 0,2 Sekunden genau 714 240 Zufallsbits und ist somit sehr schnell und unproblematisch bezüglich der Zeitschranke.

Polynomgrad n	Speicherzellen
8	70
16	86
24	102
32	118
40	134
48	150
56	176
64	192

Tabelle 14: Abhängigkeit des Speicherzellenbedarfs vom Polynomgrad n bei konstanten Werten $k = 1$ (Besetzungszahl) und $b = 4$ (Ausgabebitbreite).

Um eine allgemeine Formel für den Speicherzellenbedarf zu erhalten, wurde mit dem Programm `AMS SYNOPSYS` [14] der Speicherzellenbedarf in Abhängigkeit des Grades des benutzten Polynoms, der Besetzung des Polynoms und der Ausgabebreite der Zufallszahlen ermittelt. Mit einem höheren Polynomgrad n steigt der Zellenbedarf, da die Anzahl der Register genau dem Polynomgrad entspricht. Ist die Besetzung k des Polynoms dichter, so steigt auch der Speicherzellenbedarf, da entsprechend mehr XOR-Verknüpfungen ausgeführt werden müssen. Eine größere Ausgabebitbreite b erfordert mehr Speicherzellen, da für den Zähler zusätzliche Register erforderlich sind. In den Tabellen 14, 15 und 16 sind die abhängigen Werte dargestellt.

Besetzung k	Speicherzellen
1	118
2	119
3	120
...	...
31	148
32	149

Tabelle 15: Abhängigkeit des Speicherzellenbedarfs von der Besetzung k des Polynoms bei konstanten Werten $n = 32$ (Polynomgrad) und $b = 4$ (Ausgabebitbreite).

Bei Betrachtung der Werte für den Polynomgrad n und dem Speicherzellenbedarf in Tabelle 14 ergibt sich eine lineare Abhängigkeit der Form $f(n) = 2n + x$. Die Konstante x wird hier durch die Werte von k und b bestimmt und hat hier den Wert $x = 54$ (das ergibt sich z.B. aus der ersten Zeile von Tabelle 14: $70 = 2 \cdot 8 + x$).

Bei Betrachtung der Werte für die Besetzung k des Polynoms und dem Speicherzellenbedarf (Tabelle 15) ergibt sich eine lineare Abhängigkeit der Form $f(k) = k + y$. Die Konstante y wird hier durch die Werte von n und b bestimmt und hat hier den Wert $y = 117$.

Bei Betrachtung der Werte für die Ausgabebitbreite b und den Speicherzellenbedarf (Tabelle 16) ergibt sich eine linearlogarithmische Abhängigkeit der Form $f(b) = 2 \cdot \lfloor \log_2(b) \rfloor + b + z$. Die Konstante z wird hier durch die Werte von n und k bestimmt und hat hier den Wert $z = 110$.
Bei Betrachtung der drei Abhängigkeiten ergibt sich nach einfacher Rechnung die folgende Formel für den Speicherzellenbedarf:

$$f(n, k, b) = 2n + k + 2 \cdot \lfloor \log_2(b) \rfloor + b + 45$$

Ausgabebitbreite b	Speicherzellen
1	111
2	114
3	115
4	118
5	119
6	120
7	121
8	124
9	125
10	126

Tabelle 16: Abhängigkeit des Speicherzellenbedarfs von der Ausgabebitbreite b bei konstanten Werten $n = 32$ (Polynomgrad) und $k = 1$ (Besetzungszahl des Polynoms).

Ist nun beispielsweise der Speicherzellenbedarf für ein LFSR gefragt, welches durch das Polynom $x^{54} + x^8 + x^6 + x^3 + 1$ beschrieben wird und nach vier Takten eine Zufallszahl der Breite vier ausgeben soll, dann sind die drei Werte $n = 54$, $k = 4$ und $b = 4$. Beim Einsetzten in die Formel ergibt sich:

$$f(54; 4; 4) = 2 \cdot 54 + 4 + 2 \cdot \lfloor \log_2(4) \rfloor + 4 + 45 = 165$$

Dieser Wert wird durch `AMS SYNOPSYS` bestätigt.

[14] Die Programmeinstellungen sind wie in Kapitel 4.2 beschrieben.

6.3 BBS

In diesem Kapitel werden die Implementationen des BBS-Generators vorgestellt, die Komplexität des Zeit- und Speicheraufwandes zur Berechnung von $x_{i+1} = x_i^2 \bmod n$ sowie die Leistungsaufnahme untersucht.

Implementierung von BBS in C

Die Implementierung des BBS-Generators in C liegt in zwei Versionen vor. Einmal die „int"-Version, die mit Integer-Werten arbeitet. Diese Implementierung ist wegen der Verarbeitung ganzzahliger Werten besser für eine Chipkartenimplementierung geeignet. Allerdings führt die Begrenzung der Wortbreite auf 32 Bit auf den meisten Rechnern dazu, dass bei einem Überlauf der Generator nur noch Nullen erzeugt. Dieser Fall tritt auf, wenn beim Quadrieren ein Wert größer als 2^{32} erzeugt wird. Die zweite Version der BBS-Implementierung ist die „double"-Version, die intern mit Gleitkommazahlen rechnet und daher einen größeren Wertebereich bietet. Eine Implementation, die Gleitkommazahlen verwendet, ist wegen des höheren Aufwands nicht für Chipkarten geeignet.

Neben den Standard-Parametern, die jeder Zufallsgenerator bietet (siehe Kapitel 6.1, Abschnitt „Die Implementierungen in C"), sind in der Implementation vom BBS-Generator folgende Einstellungen zusätzlich wählbar:

- `#define PRODUKT` — Angabe des Modulus, der für den Generator verwendet werden soll.
- `#define START` — Angabe des Startwertes, mit dem der Generator initialisiert werden soll.
- `#define BITBREITE` — Angabe der Anzahl der Bits, die pro Quadrierung des quadratischen Restes ausgegeben werden sollen. Problemlos ist die Ausgabe von bis zu zwei Bit. Bei der Ausgabe von mehr als zwei Bit sind gewisse Einschränkungen bezüglich der Qualität der Zufallszahlen zu erwarten (siehe Kapitel 5.2).

Implementierung von BBS in VHDL

Der BBS-Generator ist auch in der VHDL-Implementation frei konfigurierbar. Die Parameter entsprechen der C-Implementation. Zusätzlich muss in der VHDL-Implementation in der Datei `bbs_pkg.vhd` die Konstante `PRODUKTLOG` angegeben werden, das ist der auf eine ganze Zahl aufgerundete Logarithmus des verwendeten Modulus'.

Folgende Register wurden für den BBS-Generator in der VHDL-Implementation definiert:

```
signal wert:                std_logic_vector(PRODUKTLOG-1 downto 0);
signal rechenregister:      std_logic_vector(PRODUKTLOG*2-1 downto 0);
signal multregister:        std_logic_vector(PRODUKTLOG-1 downto 0);
signal ergebnisregister:    std_logic_vector(PRODUKTLOG*2+1-1 downto 0);
signal fertig:              std_logic_vector(BITBREITE-1 downto 0);
```

Dabei sind alle Register vom Typ „std_logic_vector". Im „wert"-Register steht der aktuelle quadratische Rest des Generators. Das „rechenregister" dient zur Berechnung des Produktes bei der Quadrierung, das „multregister" dient dabei als Hilfsregister für die Multiplikation. Im „ergebnisregister" steht dann das Ergebnis der Multiplikation bzw. der Moduloreduktion. Das Register „fertig" dient zum Signalisieren, ob die Berechnung der Zufallsbits fertig ist.

BBS wurde – wie die anderen Zufallsgeneratoren auch – als Mealy-Automat realisiert. Das Zustandsdiagramm dieses endlichen Automaten ist in Abb. 18 dargestellt. Der Automat soll hier nicht in allen Einzelheiten beschrieben werden, Details lassen sich der Abb. 18 und dem VHDL-Code (auf der beiliegenden CD, siehe Anhang A) entnehmen. Nach dem Reset geht der Generator in den „frei"-Zustand über und setzt das „wert"-Register auf den Startwert des Generators und alle anderen Register auf Null und wartet auf das „start"-Signal. Beim Anliegen des „start"-Signals wechselt der Generator in den „quad"-Zustand, in diesem wird der aktuelle „wert" quadriert. Ist die Quadrierung beendet, geht der Generator in den „redux"-Zustand über, in dem dann aus dem Quadrat mittels Reduktion durch den Modulus der neue quadratische Rest gebildet wird. Ist die Reduktion beendet, steht im „ergebnisregister“ der neue quadratische Rest, der auch in das „wert“-Register geschrieben wird. Beim Übergang in den „frei“-Zustand wird dann das Signal „fertig“ gesetzt.

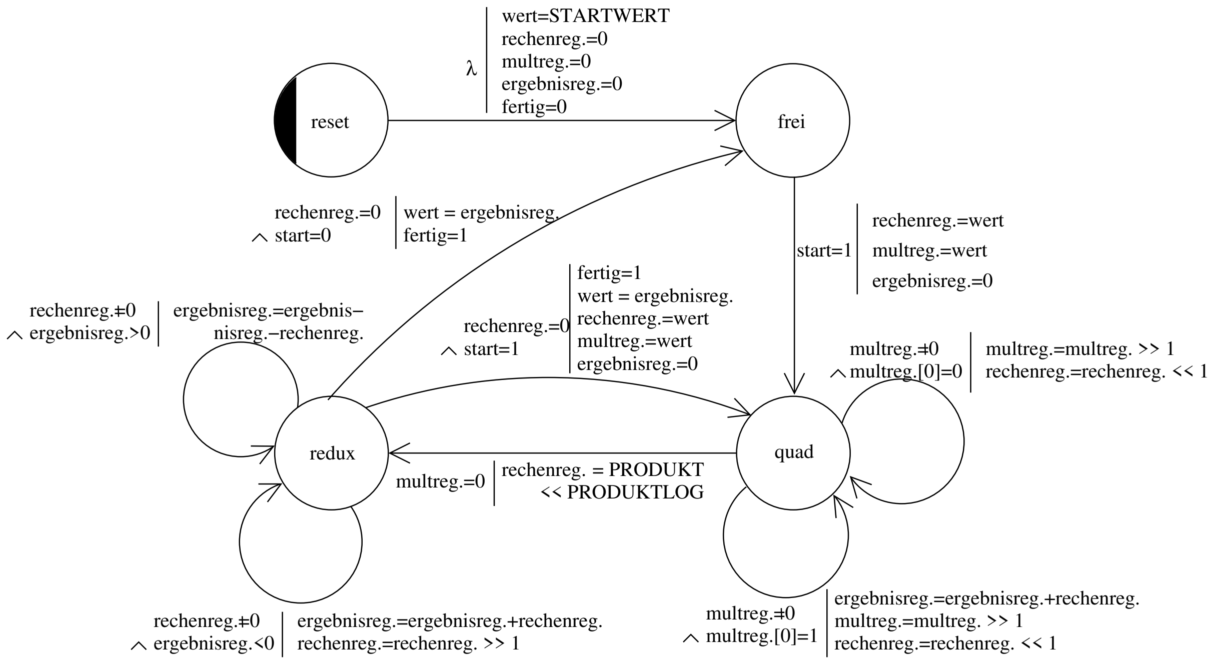

Abbildung 18: Zustandsdiagramm des Automaten, der in der VHDL-Implementation des BBS-Zufallsgenerators beschrieben wird. Jede Transition ist mit der Bedingung für den Übergang (λ bei spontanem Übergang) und den Wertzuweisungen, die beim Zustandswechsel durchgeführt werden, beschriftet. Wenn ein Wert beim Zustandsübergang nicht explizit zugewiesen wird, bedeutet das, dass er den Wert beibehält. Zur Übersichtlichkeit ist das Setzen von „fertig=0“ nicht immer explizit angegeben.

Komplexität

Der BBS-Generator ist erheblich aufwändiger und langsamer als das LFSR. Bei den Ergebnissen, die hier dargestellt werden, ist aber nicht zu vergessen, dass die Möglichkeit eines Akzelerators besteht. Ist ein solcher vorhanden, der die Quadrierung übernimmt, dann sind die Ergebnisse im Vergleich zu LFSR und RC4 nicht mehr ganz so schlecht.

Im Folgenden soll nun untersucht werden, welche Zeit- und Speicherplatzkomplexität der BBS-Generator in Abhängigkeit vom gewählten Modulus besitzt.

Um eine allgemeine Formel für den Speicherzellenbedarf zu erhalten, wurde mit dem Programm `AMS SYNOPSYS` [15] der Speicherzellenbedarf in Abhängigkeit von der Größe des Modulus’ und der Ausgabebitbreite untersucht. Für veränderlichen Modulus m und konstante Ausgabebitbreite $b = 1$ ergeben sich die in Tabelle 17 dargestellten Werte.

Modulus m	Speicherzellen
3	92
4	104
5	104
6	104
7	104
8	116
…	…
15	116
16	128
…	…
31	128
32	140

Tabelle 17: Abhängigkeit des Speicherzellenbedarfs vom Modulus m bei der Ausgabebreite $b = 1$.

Der logarithmische Anstieg der Speicherzellen in Abhängigkeit des Modulus’ m ergibt sich aus der binären Speicherung der Operanden. Bei Betrachtung der Werte der Tabelle 17 ergibt sich eine Abhängigkeit der Form $f(m) = 12 \cdot (\lfloor \log_2(m) \rfloor + 1) + 68$.

Bei einer Änderung der Ausgabebreite b und konstantem Modulus $m = 128$ ergeben sich die Werte, die in Tabelle 18 zu finden sind. Natürlich sind die gewählten Werte zu klein für einen praktischen Einsatz, da die Ausgabebreite viel zu hoch gewählt und der Modulus viel zu klein gewählt wurde, diese Wahlen zur Ermittlung der Abhängigkeiten aber nützlich sind.

[15] Die Programmeinstellungen sind wie in Kapitel 4.2 beschrieben.

Mit diesen Werten ergibt sich für den Speicherzellenbedarf eine Abhängigkeit von der Ausgabebreite in der Form $f(b) = 3b + 161$. Setzt man nun diese beiden ermittelten Abhängigkeiten zusammen, dann ergibt sich für den BBS-Generator folgende Gleichung:

$$f(m, b) = 12 \cdot (\lfloor \log_2(m) \rfloor + 1) + 3b + 65$$

Ist nun beispielsweise der Speicherzellenbedarf eines BBS-Generators mit dem Modulus 1 081 und einer 2 Bit-Ausgabe gefragt, dann ergibt sich damit:

$$f(1\,081; 2) = 12 \cdot (\lfloor \log_2(1\,081) \rfloor + 1) + 3 \cdot 2 + 65 = 203$$

Dieser Wert wird durch `AMS SYNOPSYS` bestätigt.

Ausgabebreite b	Speicherzellen
1	164
2	167
3	170
4	173
5	176
6	179
7	182
8	185

Tabelle 18: Abhängigkeit des Speicherzellenbedarfs von der Ausgabebreite b und konstantem Modulus $m = 128$.

Gefragt ist nun nach einer Abhängigkeit zwischen Periode und Speicherzellenbedarf. Hier wird dabei die Einschränkung vorgenommen, dass die beiden Primzahlen p und q, aus denen sich der Modulus mit $m = p \cdot q$ errechnet, in etwa gleich groß sind, da sich ansonsten bei großen unterschiedlichen Werten von p und q keine einfachen Aussagen mehr treffen lassen. Für optimal gewählte Werte von p und q ergibt sich eine Periode von $2p_2q_2$, dabei sind $p_2 = \frac{p-3}{4}$ und $q_2 = \frac{q-3}{4}$. Werden 2 Bits pro Quadrierung ausgegeben, so erhält man eine Periode von $4p_2q_2$. Unter der Annahme, dass $p_2 \approx q_2$ gilt, ergibt sich nach einfacher Rechnung für die Periode die Näherung $\frac{1}{16}\left(m^2 - \frac{3}{4} + \frac{9}{4}\right)$. Soll nun ein Wert k als Periode nicht unterschritten werden, dann ergibt sich nach Lösung der quadratischen Gleichung, dass m größer als $\frac{1}{8}(3 + 32\sqrt{k - 135})$ zu wählen ist. Eingesetzt in die Gleichung für die Abhängigkeit der Speicherzellen von dem Modulus ergibt sich somit ein Zusammenhang von Periode und Speicherbedarf von

$$f(k, 2) = 12 \cdot \left\lfloor \log_2(\frac{1}{8}(3 + 32\sqrt{k - 135}) + 1) \right\rfloor + 71.$$

Eine weitere wichtige Frage ist, wie schnell der BBS-Generator die Zufallsbits erzeugen kann. Hier wird wiederum in Abhängigkeit des variablen Modulus' und fester Ausgabebreite $b = 2$ ausgegangen und wegen etwas komplizierterer durchschnittlicher logarithmischer Größen der quadratischen Reste die Abschätzung für den schlechtesten Fall vorgenommen. Das Quadrieren der Zahlen nimmt so viele Takte in Anspruch wie diese in ihrer binären Darstellung an Bits benötigen. Die größte Zahl, die bei einem festen Modulus m quadriert werden kann, ist $m - 1$, also sind dafür im schlechtesten Fall $\lceil \log_2(m - 1) \rceil$ Takte notwendig. Weiterer Zeitaufwand ergibt sich beim Reduzieren der quadrierten Zahlen mit dem Modulus. Der Modulus m benötigt $\lceil \log_2 m \rceil$ Stellen. Die größtmögliche Quadratzahl kann $(m-1)^2$ ergeben und benötigt damit $\lceil \log_2(m-1)^2 \rceil$ bzw. $2 \cdot \lceil \log_2(m - 1) \rceil$ Stellen. Der größte zeitliche Aufwand ergibt sich, wenn an einer einzigen Stelle der quadrierten Zahlen ein Bit gesetzt ist, welches höherwertiger als das höchstwertigste Bit des Modulus ist. Dann werden für alle Stellen der Quadratzahl, die höherwertiger als das höchstwertigste Bit des Modulus und nicht gesetzt sind, 2 Takte für Subtraktion und Korrekturaddition benötigt, und für das eine gesetzte Bit ist ein Takt erforderlich. Damit ergeben sich im schlechtesten Fall weitere $\lceil \log_2(m - 1)^2 \rceil - \lceil \log_2 m \rceil$ Takte. Zusammengefasst sind das für zwei Zufallsbits genau $\lceil \log_2(m - 1) \rceil + \lceil \log_2(m - 1)^2 \rceil - \lceil \log_2 m \rceil$ Takte. Vereinfacht ergibt sich für die Anzahl der Takte für zwei Bit in Abhängigkeit des Modulus' folgende Gleichung:

$$t(m) = \left\lceil \log_2 \frac{(m - 1)^3}{\sqrt{m}} \right\rceil.$$

Sei nun $m = 16\,000\,000$, dann sind für 2 Bit höchstens $\left\lceil \log_2 \frac{(16\,000\,000-1)^3}{\sqrt{16\,000\,000}} \right\rceil = 60$ Takte notwendig. Unter Annahme der Standardfrequenz von 3,5712 MHz erzeugt der BBS-Generator in diesem Beispiel also mindestens 119 040 Zufallsbits pro Sekunde. Innerhalb von 0,2 Sekunden sind das 23 808 Zufallsbits. Der BBS-Generator ist damit zwar erheblich langsamer als LFSR und RC4, liefert aber die Zufallsbits schnell genug. Auch hier stellt also die Zeitschranke kein Problem dar.

Leistungsaufnahme

Zur Analyse der Leistungsaufnahme der VHDL-Implementation des BBS-Generators wurden, wie in Kapitel 4.3 beschrieben, die kippenden Bits der Zustandswechsel bei der Erzeugung der Zufallsbits untersucht.

Das Ergebnis der statistischen Analyse ist in Tabelle 19 gezeigt. Betrachtet wurden dabei die Register „wert" und „multregister", die hier bei diesem Beispiel eine Größe von 15 Bits benötigen sowie die Register „rechenregister" und „ergebnisregister", die bei diesem Beispiel eine Größe von 30 Bits benötigen. Das sind insgesamt 90 mögliche Bits, die wechseln können.[16] Für jede mögliche Anzahl von Bits, die sich ändern können, gibt es eine Tabellenzeile. Die Anzahlen der wechselnden Bits sind in der ersten Spalte zu finden, die gezählten Häufigkeiten in der zweiten, und in der letzten Spalte stehen die Werte einer Normalverteilung, die an die Verteilung der gezählten kippenden Bits angepasst ist.

Das Ergebnis der Untersuchung zeigt, dass die Anzahl der kippenden Bits zwischen 1 und 44 liegt. Untersucht wurden dabei die kippenden Bits nach jedem Zustandswechsel, das heißt bei jedem Durchlaufen einer Kante im Zustandsdiagramm des Automaten (siehe Abb. 18). Im Mittelwert kippen $\overline{X} = 13,9470$ Bit (die Schätzgenauigkeit von $\overline{X}$ ist 0.0675 Bit), und die empirische Standardabweichung ist $\sigma_S = 6,7348$ kippende Bits.

Erzeugt wurden 1 969 Zufallszahlen mit 107 907 Zustandswechseln. Bei einer sicheren Ausgabe von 2 Bits pro erzeugtem quadratischen Rest ergeben sich somit $10 \cdot 13,947 \cdot \frac{107\ 907}{1\ 969} = 7\ 643,367$ kippende Bits bei der Erzeugung von 20 Zufallsbits. Dieses Ergebnis ist unter dem Aspekt der Leistungsaufnahme schlecht und lässt den BBS-Generator für einen Einsatz bei kontaktlosen Chipkarten als nicht geeignet erscheinen.

Anzahl Wechsel	Häufigkeit	approx. Normalvert.
0	0	748.832
1	2	1007.251
2	466	1325.306
3	3435	1705.767
4	4714	2147.576
5	4383	2644.859
6	4507	3186.264
7	4124	3754.793
8	5140	4328.281
9	3532	4880.564
10	6333	5383.314
11	4037	5808.374
12	7016	6130.341
13	4369	6329.068
14	6803	6391.755
15	4882	6314.305
16	6614	6101.775
17	5110	5767.823
18	5516	5333.261
19	4323	4823.907
20	4312	4268.056
21	3868	3693.912
22	3297	3127.289
23	2507	2589.851
24	2023	2098.005
25	1524	1662.507
26	1109	1288.681
27	801	977.131
28	547	724.745
29	478	525.827
30	514	373.186
31	442	259.080
32	365	175.941
33	306	116.876
34	199	75.947
35	127	48.274
36	82	30.016
37	45	18.256
38	23	10.862
39	17	6.321
40	6	3.599
41	3	2.004
42	3	1.092
43	2	0.582
44	1	0.303
45	0	0.155
46	0	0.077
47	0	0.038
48	0	0.018
49	0	0.008
50	0	0.004
51	0	0.002
52	0	0.001
53	0	0.000
…	…	…
120	0	0.000

Tabelle 19: Untersuchung der kippenden Bits beim Erzeugen von Zufallszahlen. Es wurden der Modulus $n = 47 \cdot 719 = 33793$ und die Zahl zum Erzeugen des Startwertes $x = 2$ (vgl. Kap 5.2) gewählt.

[16] Bei der Implementation des Statistiktests wurden zur einfacheren Implementation auch „rechenregister" und „ergebnisregister" als 30 Bit breite Zahlen angenommen, deren obere 15 Bit immer 0 sind, die also nicht kippen. Bei der Statistikausgabe werden daher maximal 120 Bitwechsel angegeben, von denen aber nur maximal 90 tatsächlich auftreten können.

6.4 RC4

Im Folgenden werden die Implementationen des RC4-Zufallsgenerators vorgestellt und die Analysen der Hardware-Kriterien (Leistungsaufnahme und Aufwand) zusammengefasst.

Implementierung von RC4 in C

Die Implementierung des RC4-Zufallsgenerators ist in C problemlos möglich, weil die erforderlichen Datentypen in C zur Verfügung stehen (die S-Box, i und j sind als „unsigned integer“ realisiert) und der Wertebereich der Typen hinreichend groß ist. Für 4-Bit-RC4 sind die maximalen Werte für i,j und die S-Box-Werte $2^4 - 1 = 15$, während der Datentyp „unsigned integer“ bis zu $2^{32} - 1$ zulässt.

Neben den Standard-Parametern, die jeder Zufallsgenerator bietet (siehe Kapitel 6.1, Abschnitt „Die Implementierungen in C“), sind in der Implementation von RC4 folgende Einstellungen zusätzlich wählbar:

- `#define BITBREITE 4` — Anzahl der Bits, die jede Zufallszahl breit sein soll. Dieser Wert beeinflusst die Größe der S-Box und den Wertebereich der Variablen i und j (siehe Kapitel 5.3). Wenn mehrere Zufallszahlen zu einer längeren zusammengesetzt werden sollen, wie es bei der Analyse in Kapitel 5.3 getan wurde, so muss die Bitbreite beibehalten und die Anzahl der Zufallszahlen entsprechend erhöht werden. Wenn beispielsweise die mit 4-Bit-RC4 erzeugten 4-Bit-Zufallszahlen zu 20-Bit-Zufallszahlen zusammengesetzt werden sollen, ist keine Änderung der Bitbreite nötig, weil die 4-Bit-Zufallszahlen aufeinanderfolgend in die Ausgabedatei geschrieben werden und 5 4-Bit-Zahlen vom Statistikprogramm als 20-Bit-Zahlen eingelesen werden können. Um die Verminderung der Anzahl an Zufallszahlen durch das Zusammenfassen auszugleichen, muss die Anzahl der 4-Bit-Zufallszahlen im RC4-Generator erhöht werden.

- `#define STANDARDSCHLUESSEL true` — Auswahl, ob der Standardschlüssel $K_0 = 0$, $K_1 = 1, \ldots, K_{2^{\text{BITBREITE}}-1} = 2^{\text{BITBREITE}} - 1$ verwendet werden soll.

- `unsigned int Schluessel [SBOX_GROESSE]=...` — Hier kann der Schlüssel angegeben werden, wenn der Standardschlüssel nicht verwendet werden soll.

Implementierung von RC4 in VHDL

Der RC4-Zufallsgenerator ist auch in der VHDL-Implementation frei konfigurierbar. Die Parameter entsprechen denen des C-Programms, es gibt aber einige Unterschiede. Aus diesem Grund werden die in der Datei `rc4_pkg.vhd` einstellbaren Parameter kurz erläutert:

- `constant BITBREITE: integer := 4;` — Diese Einstellung entspricht der des C-Programms.

- `constant SBOX_GROESSE: integer := 16;` — Dieser Wert muss auf $2^{\text{BITBREITE}}$ gesetzt werden.

- `constant AUSGABEBREITE: integer := 20;` — Im Gegensatz zum C-Programm kann hier explizit angegeben werden, wie viele Bits der Zufallsgenerator ausgeben soll. Die Ausgabebreite muss ein Vielfaches der Bitbreite sein. Wenn beispielsweise die Bitbreite 4 und die Ausgabebreite 20 ist, dann erzeugt die Hardware 5 4-Bit-Zufallszahlen, setzt sie zu 20 Bit zusammen und gibt diese 20 Bit aus.

- `constant RUNDENZAHLLOG: integer := 3;` — Die Rundenzahl ist die Anzahl der Zufallszahlen, die zu einer auszugebenden Zahl zusammengesetzt werden, im obigen Beispiel also 5. Hier muss der aufgerundete duale Logarithmus der Rundenzahl angegeben werden, also wiederum im Beispiel $\lceil log_2(5) \rceil = 3$.

➩ `constant Schluessel: BOX := ...` — Hier wird der Schlüssel angegeben. Einen Standardschlüssel gibt es nicht.

Bei der Implementation in VHDL werden für RC4 folgende Register definiert:

```
signal SBox:    array (0 to SBOX_GROESSE-1) of integer range 0 to SBOX_GROESSE-1;
signal i:       integer range 0 to SBOX_GROESSE-1;
signal j:       integer range 0 to SBOX_GROESSE-1;
signal zaehler: std_logic_vector(RUNDENZAHLLOG-1 downto 0);
signal ausgabe: std_logic_vector(AUSGABEBREITE-1 downto 0);
signal fertig:  std_logic_vector(AUSGABEBREITE-1 downto 0);
```

Dabei sind die S-Box, i und j, wie in Kapitel 5.3 angegeben, die Variablen, die für die Erzeugung einer Zufallszahl benötigt werden. Sie sind in VHDL als „integer“ und nicht als „std_logic_vector“ realisiert, weil die Indizes zum Zugriff auf ein Arrayelement nur vom Typ „integer“ sein dürfen. Der Zähler wird zum Rückwärtszählen der Rundenzahl benötigt, „ausgabe“ dient zum Speichern der zusammengesetzten Zufallszahlen, bevor sie ausgegeben werden, und „fertig“ dient zum Signalisieren, ob die Berechnung fertig ist.

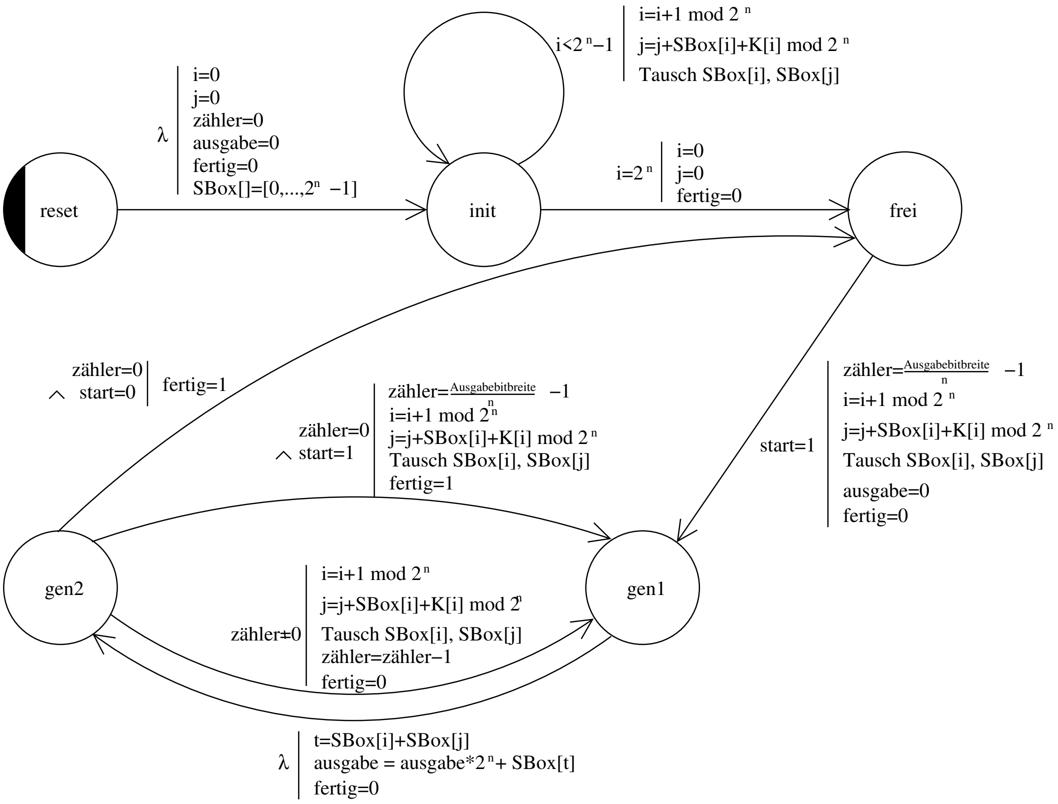

Abbildung 19: Zustandsdiagramm des Automaten, der in der VHDL-Implementation von RC4 beschrieben wird. Jede Transition ist mit der Bedingung für den Übergang (λ bei spontanem Übergang) und den Wertzuweisungen, die beim Zustandswechsel durchgeführt werden, beschriftet. Wenn ein Wert beim Zustandsübergang nicht explizit zugewiesen wird, bedeutet das, dass er den Wert beibehält.

RC4 wurde – wie die anderen Zufallsgeneratoren auch – als Mealy-Automat realisiert. Das Zustandsdiagramm dieses endlichen Automaten ist in Abb. 19 dargestellt. Nach dem Reset geht der RC4-Zufallsgenerator in den „init“-Zustand über, in dem die S-Box mithilfe des Schlüssels initialisiert wird. Danach wartet der Generator auf das „start“-Signal. Sobald es anliegt, wird der Zähler mit der Rundenzahl, also mit $\frac{\text{Ausgabebreite}}{\text{Bitbreite}} - 1$ initalisiert (weil bis 0 zurückgezählt wird, wird 1 subtrahiert). Die Zustände „gen1“ und „gen2“ werden so oft durchlaufen wie der Zähler angibt. Dabei wird beim Übergang in „gen1“ die S-Box permutiert, und beim

Übergang zu „gen2“ wird mithilfe des Registers t ein Wert aus der S-Box gewählt und in das Ausgaberegister geschoben. Wenn der Zähler im Zustand „gen2“ gleich 0 ist, wird das Ausgaberegister ausgegeben, das „fertig“-Signal gesetzt und abhängig vom „Start“-Signal in den Zustand „frei“ oder „gen1“ gewechselt.

Leistungsaufnahme

Zur Analyse der Leistungsaufnahme der VHDL-Implementation von RC4 wurden, wie in Kapitel 4.3 beschrieben, die kippenden Bits bei der Erzeugung einer neuen Zufallszahl untersucht.

In die Betrachtung einbezogen wurden dabei die Variablen i und j sowie die Werte S-Box. Das Ergebnis der Untersuchung von 1 000 000 Zufallszahlen der Länge 4 Bit ist in Tabelle 20 angegeben. Es sind bei der Bitbreite 4 maximal $2^4 \cdot 4 + 2 \cdot 4 = 72$ (für die S-Box $2^4 \cdot 4$ Bit und für i und j je 4 Bit) kippende Bits möglich. Für jede mögliche Anzahl an Wechseln gibt es eine Tabellenzeile. Die Anzahl ist in der ersten Spalte eingetragen; in der zweiten Spalte ist die gezählte Häufigkeit zu finden, und in der letzten Spalte stehen die Werte einer Normalverteilung, die an die Verteilung der gezählten Häufigkeiten angepasst ist.

Das Ergebnis der Untersuchung zeigt, dass die Anzahl der kippenden Bits zwischen 1 und 16 liegt. Im Mittelwert kippen $\overline{X} = 7,8764$ Bit (die Schätzgenauigkeit von $\overline{X}$ ist 0.0082 Bit), und die empirische Standardabweichung ist $\sigma_S = 2.4769$ kippende Bits.

Für die Leistungsaufnahme der Hardware ist das Ergebnis, dass pro 4-Bit-Zufallszahl im Mittel $7,8764$ Bit kippen, sehr gut. Für 20-Bit-Zufallszahlen ist die Anzahl der kippenden Bits also (unter der Annahme der Unabhängigkeit) $5 \cdot 7,8764 = 39,382$ Bit. Zur Verwendung auf Chipkarten ist diese Eigenschaft des RC4-Zufallsgenerators hervorragend, weil die Leistungsaufnahme minimal ist.

Anzahl Wechsel	Häufigkeit	approx. Normalvert.
0	0	1026.066
1	1941	3414.680
2	8823	9654.621
3	23915	23191.599
4	51497	47329.994
5	87155	82063.966
6	123695	120886.791
7	148855	151291.837
8	155327	160865.325
9	140063	145318.090
10	110263	111528.848
11	74219	72721.872
12	42640	40285.910
13	20721	18960.580
14	7883	7581.591
15	2511	2575.606
16	492	743.376
17	0	182.284
18	0	37.975
19	0	6.721
20	0	1.011
21	0	0.129
22	0	0.014
23	0	0.001
24	0	0.000
25	0	0.000
...	...	...
71	0	0.000
72	0	0.000

Tabelle 20: Untersuchung der kippenden Bits beim Erzeugen einer Zufallszahl. Erzeugt wurden 1 000 000 Zufallszahlen. Das Stichprobenmittel ist $\overline{X} = 7,8764 \pm 0.0082$ kippende Bits, die empirische Stichprobenstandardabweichung ist $\sigma_S = 2.4769$ kippende Bits.

Komplexität

Der Zeitbedarf zur Erzeugung einer Zufallszahl mit dem RC4-Zufallsgenerator ist 2 Takte zum Generieren einer 4-Bit-Zahl (bei Verwendung des 4-Bit-RC4). Diese Zahl ergibt sich aus dem Zustandsdiagramm (Abb. 19), weil zum Erzeugen einer 4-Bit-Zufallszahl die beiden Zustände „gen1“ und „gen2“ durchlaufen werden.

Allgemein ist der Zeitbedarf $t(m)$ zum Erzeugen einer n Bit breiten Zufallszahl mit 4-Bit-RC4

$$t(m) = 2 \cdot \frac{m}{4}.$$

Dabei muss m durch 4 teilbar sein, das heißt es sind nur Vielfache von 4 als Ausgabebreite zulässig. Der Zufallsgenerator geht beim Erzeugen einer Zufallszahl vor wie in Kapitel 5.3 beschrieben: Wenn mehr als 4 Bit angefordert werden, erzeugt er mehrere 4-Bit-Zahlen und setzt sie zu einer langen Zufallszahl zusammen.

Wenn der Takt die Standardfrequenz von Chipkarten ist (3,5712 MHz), dann erzeugt RC4 innerhalb von 0,2 Sekunden 7 142 400 Zufallsbits (je 4 Bit in 2 Takten). Die Anforderung nach kurzer Antwortzeit ist damit sehr gut erfüllt.

Zur Untersuchung des Hardwareaufwands, den die VHDL-Implementation von RC4 benötigt, wurde für verschiedene Ausgabebreiten b der Bedarf an Speicherzellen ermittelt. Das Ergebnis ist in Tabelle 21 dargestellt. Zwischen den beiden Größen ergibt sich induktiv die Beziehung

$$f(b) = 22 \cdot \frac{b}{4} + 561.$$

Ausgabebreite b	Speicherzellen
8	605
12	627
16	649
20	671

Tabelle 21: Abhängigkeit des Speicherzellenbedarfs von der Ausgabebreite b.

Die Anzahl der notwendigen Speicherzellen steigt also linear mit der Ausgabebreite an. Im Vergleich mit den zuvor untersuchten Zufallsgeneratoren LFSR und BBS werden mehr Zellen benötigt. Das liegt daran, dass für RC4 ein aufwändiges Permutationsnetzwerk zur Realisierung der Vertauschungen der S-Box gebraucht wird.

Die Periode ist beim 4-Bit-RC4-Generator unabhängig vom Speicherbedarf, weil die Periode eine Konstante ist, wie in Kapitel 5.3 (Abschnitt „Parametrisierung des Verfahrens für Chipkarten“) beschrieben wurde. Da die Periode aber konstant in der Größenordnung 10^{15} liegt und damit die Sicherheit des RC4-Generators im Low-Security-Bereich in jedem Fall gewährleistet ist, ist das kein Nachteil.

6.5 Elliptische Kurven

Im Folgenden werden die Implementierungen des EC-Generators vorgestellt, die Komplexität in Hinblick auf den Speicher- und den Zeitaufwand untersucht sowie Angaben zur Leistungsaufnahme gemacht.

Implementierung von EC in C

Die Implementierung des EC-Zufallsgenerators ist in C nicht ganz einfach, da die 32-Bit-Beschränkung auf den meisten Rechnern dazu führt, dass die eingesetzte Beispielkurve nicht korrekt benutzt werden kann, weil durch die auftretenden Multiplikationen die 32-Bit-Darstellung nicht ausreicht. Mit einigen Anpassungen ist die Einschränkung allerdings umgehbar. Die Einzelheiten sind ausführlich in der Datei `EC.c` (siehe Anhang A) dokumentiert. Sämtliche Variablen, sowohl die Punktkoordinaten als auch die Hilfsvariablen (siehe Kapitel 3.2.2), sind als „integer“ realisiert.

Neben den Standard-Parametern, die jeder Zufallsgenerator bietet (siehe Kapitel 6.1, Abschnitt „Die Implementierungen in C“), sind in der Implementation von EC folgende Einstellungen zusätzlich wählbar:

- `#define KURVE_A` — Angabe des Kurvenparameters a.
- `#define KURVE_B` — Angabe des Kurvenparameters b, dieser hat nur einen informativen Wert, da er nicht zur Berechnung benutzt wird. Der Wert ergibt sich implizit aus dem angegeben Startwert.
- `#define KURVE_X` — Angabe der x-Koordinate des Startpunktes P_0.
- `#define KURVE_Y` — Angabe der y-Koordinate des Startpunktes P_0.
- `#define KURVE_MODULUS` — Angabe des Modulus p der Gruppe $\mathcal{GF}(p)$.
- `#define KURVE_MODULUSLOG` — Angabe der Anzahl Bits zur Darstellung von `KURVE_MODULUS`.
- `#define AUSGABE_KOORDINATE` — Angabe, welche Koordinate zur Ausgabe benutzt werden soll, möglich sind x, y oder z.
- `#define BITBREITE` — Angabe der Anzahl der Bits, die der Ausgabekoordinate entnommen werden sollen.

Implementierung von EC in VHDL

Der EC-Zufallsgenerator ist auch in der VHDL-Implementation frei konfigurierbar. Die Parameter entsprechen denen des C-Programms.

Bei der Implementation in VHDL werden für den EC-Generator folgende Register definiert:

```
signal x:                 std_logic_vector(KURVE_MODULUSLOG-1 downto 0);
signal y:                 std_logic_vector(KURVE_MODULUSLOG-1 downto 0);
signal z:                 std_logic_vector(KURVE_MODULUSLOG-1 downto 0);
signal m:                 std_logic_vector(KURVE_MODULUSLOG-1 downto 0);
signal s:                 std_logic_vector(KURVE_MODULUSLOG-1 downto 0);
signal t:                 std_logic_vector(KURVE_MODULUSLOG-1 downto 0);
signal rechenregister:    std_logic_vector(KURVE_MODULUSLOG*2-1 downto 0);
signal fak1register:      std_logic_vector(KURVE_MODULUSLOG*2-1 downto 0);
signal fak2register:      std_logic_vector(KURVE_MODULUSLOG*2-1 downto 0);
signal ergebnisregister:  std_logic_vector(KURVE_MODULUSLOG*2+1-1 downto 0);
signal fertig:            std_logic_vector(BITBREITE-1 downto 0);
```

Für die Punktkoordinaten werden x, y und z benutzt, und m, s und t sind die Hilfsvariablen zur Berechnung der neuen Punktkoordinaten bei der Punktverdopplung (siehe Kapitel 3.2.2). Das „rechenregister" dient zur Produktberechnung, die Register „fak1register" und „fak2register" zur Multiplikation (die beiden Faktoren), das „ergebnisregister" für das Multiplikations- und das Endergebnis, und das Register „fertig" signalisiert das Ende der Berechnung.

EC wurde – wie die anderen Zufallsgeneratoren auch – als Mealy-Automat realisiert. Auf das Zustandsdiagramm dieses endlichen Automaten wird verzichtet, da der Automat 30 Zustände besitzt und eine grafische Darstellung bei der Größenordnung nichts mehr veranschaulicht. Auch auf eine detaillierte Beschreibung der Arbeitsweise des Generators in der VHDL-Implementation soll verzichtet werden, da diese zu umfangreich und auch recht redundant ausfallen würde. Die benutzten Grundoperationen wie die Multiplikation oder die Addition sind insoweit nicht erklärungsbedürftig, und der Punktverdopplungsalgorithmus findet sich in Kapitel 3.2.2.

Es ist aber ohne nähere Betrachtungen ersichtlich, dass der EC-Generator den größten Berechnungsaufwand verursacht.

Komplexität

Der EC-Generator ist hinsichtlich der Zeitkomplexität der schlechteste unter den hier vorgestellten Generatoren. Vom Speicherzellenbedarf liegt er zwischen dem BBS- und dem RC4-Generator.

Die Speicherkomplexität wurde wie bei den anderen Generatoren ermittelt. Aus Tabelle 22 ergibt sich offensichtlich in Abhängigkeit vom Kurvenmodulus m und der Ausgabebreite b folgende Formel für die Speicherzellen:

$$f(m, b) = 28 \cdot \lceil \log_2 m) \rceil + 3b + 247.$$

Für unsere Beispielkurve $y^2 = x^3 + 2155 \cdot x + 16008$ über $\mathcal{GF}(33\,013)$ mit einer 4-Bit-Ausgabe ergibt sich somit:

$$f(33\,013; 4) = 28 \cdot \lceil \log_2 33\,013 \rceil + 3 \cdot 4 + 247 = 707.$$

Dieser Wert wird durch `AMS SYNOPSYS` bestätigt.

Logarithmus des Modulus m	Bitbreite b	Speicherzellen
2	1	306
2	2	309
3	1	334
3	2	337
3	3	340
4	1	362
4	2	365
4	3	368
4	4	371
5	1	390
5	2	393
5	3	396
5	4	399
5	5	402
6	1	418

Tabelle 22: Abhängigkeit des Speicherzellenbedarfs vom Logarithmus des Modulus m bei der Ausgabebreite b.

Zur Zeitkomplexität wird in analoger Weise wie für den BBS-Generator vorgegangen. Es wird vom schlechtesten Fall ausgegangen, und es werden die beim BBS-Generator vorgestellten Ergebnisse benutzt (siehe

Kapitel 6.3). Bei der Punktverdopplung sind fünf Multiplikationen mit den Konstanten 2, 3, 4 und 8 erforderlich (siehe Kapitel 3.2.2). Das ist vergleichsweise vernachlässigbarer Rechenaufwand. Auch die drei Subtraktionen und die eine Addition lassen sich, wenn auch schon etwas aufwändiger, noch vergleichsweise vernachlässigen. Es sind weiterhin 13 Multiplikationen notwendig, bei denen die Faktoren bis zu $m - 1$ groß werden können. Nach den beim BBS-Generator durchgeführten Betrachtungen ergibt das in etwa folgende Abschätzung in Abhängigkeit des Modulus' m folgenden Zeitaufwand pro Punktverdopplung:

$$t(m) = 13 \cdot \left\lceil \log_2 \frac{(m-1)^3}{\sqrt{m}} \right\rceil$$

Sei nun $m = 16\ 000\ 000$, dann sind für 4 Bits in etwa höchstens $13 \cdot \left\lceil \log_2 \frac{(16\ 000\ 000 - 1)^3}{\sqrt{16\ 000\ 000}} \right\rceil = 778$ Takte notwendig. Unter Annahme der Standardfrequenz von 3,5712 MHz erzeugt der EC-Generator in diesem Beispiel also mindestens 18 360 Zufallsbits pro Sekunde. Innerhalb von 0,2 Sekunden sind das 3 672 Zufallsbits. Der EC-Generator ist damit zwar der langsamste unter den hier vorgestellten Zufallszahlengeneratoren, liefert aber die Zufallsbits schnell genug. Auch hier stellt also die Zeitschranke kein Problem dar. Der Modulus von 16 000 000 ist deshalb gewählt worden, da dies dem verwendeten BBS-Modulus entspricht. Praktischerweise wird wohl kaum eine solche große elliptische Kurve im Low-Security-Bereich eingesetzt werden. Für ein $m = 30\ 000$ liefert der Generator in 0,2 Sekunden immerhin schon 5 910 Bits.

Anzahl Wechsel	Häufigkeit	approx. Normalvert.	Anzahl Wechsel	Häufigkeit	approx. Normalvert.
0	2742	5343.719	35	2418	6883.772
1	1	7095.765	36	2602	5174.829
2	7	9270.330	37	1971	3827.419
3	30	11916.029	38	2221	2785.200
4	4208	15069.829	39	1962	1994.101
5	39471	18751.046	40	2072	1404.683
6	57416	22955.306	41	1861	973.531
7	55137	27649.104	42	1731	663.837
8	49744	32765.699	43	1443	445.363
9	37771	38203.065	44	1511	293.973
10	52416	43824.541	45	1366	190.915
11	43493	49462.601	46	1230	121.987
12	43667	54925.867	47	991	76.688
13	46928	60009.125	48	841	47.433
14	52922	64505.696	49	721	28.865
15	57912	68221.181	50	618	17.283
16	62340	70987.324	51	509	10.181
17	75804	72674.620	52	372	5.901
18	75708	73202.369	53	265	3.365
19	73570	72545.070	54	223	1.888
20	75857	70734.464	55	140	1.042
21	68379	67856.997	56	89	0.566
22	62832	64046.971	57	74	0.302
23	55524	59476.166	58	52	0.159
24	52201	54341.012	59	29	0.082
25	47673	48848.686	60	17	0.042
26	46172	43203.452	61	8	0.021
27	47961	37594.508	62	18	0.010
28	43631	32186.279	63	6	0.005
29	30339	27111.749	64	3	0.002
30	21793	22469.049	65	0	0.001
31	11552	18321.129	66	0	0.001
32	8198	14698.066	67	0	0.000
33	4412	11601.351	...	...	...
34	4047	9009.430	310	0	0.000

Tabelle 23: Untersuchung der kippenden Bits beim Erzeugen einer Zufallszahl. Erzeugt wurden 1 847 Zufallszahlen. Das Stichprobenmittel ist $\overline{X} = 17.9451 \pm 0.0215$ *kippende Bits, die empirische Stichprobenstandardabweichung ist* $\sigma_S = 7.8434$ *kippende Bits. Die Spalten 4, 5 und 6 sind die Fortsetzungen der Spalten 1, 2 und 3.*

Leistungsaufnahme

Zur Analyse der Leistungsaufnahme der VHDL-Implementation des EC-Generators wurden, wie in Kapitel 4.3 beschrieben, die kippenden Bits der Zustandswechsel bei der Erzeugung der Zufallsbits untersucht.

Das Ergebnis der statistischen Analyse ist in Tabelle 23 gezeigt. Betrachtet wurden dabei die Register der Variablen x, y, z, m, s und t sowie die Register „rechenregister“, „fak1register“, „fak2register“ und „ergebnisregister“. Jedes dieser zehn Register benötigt 31 Bits, da die Kurve $y^2 = x^3 + 2155 \cdot x + 16008$ über $\mathcal{GF}(33\,013)$ mit dem Startpunkt $P_0(x_0; y_0) = (3; 150)$ benutzt wurde. Insgesamt sind also 310 mögliche Bits vorhanden, die ihren Wert verändern können. Für jede mögliche Anzahl von Bits, die sich ändern können, gibt es eine Tabellenzeile. Die Anzahlen der wechselnden Bits sind in der ersten Spalte zu finden, die gezählten Häufigkeiten in der zweiten, und in der letzten Spalte stehen die Werte einer Normalverteilung, die an die Verteilung der gezählten kippenden Bits angepasst ist.

Das Ergebnis der Untersuchung zeigt, dass die Anzahl der kippenden Bits zwischen 0 und 66 liegt. Untersucht wurden dabei die kippenden Bits nach jedem Zustandswechsel. Im Mittelwert kippen $\overline{X} = 17.9451$ Bit (die Schätzgenauigkeit von $\overline{X}$ ist 0.0215 Bit), und die empirische Standardabweichung ist $\sigma_S = 7.8434$ kippende Bits.

Erzeugt wurden 1 847 Zufallszahlen mit 1 439 222 Zustandswechseln. Bei einer Ausgabe von 4 Bit[17] pro erzeugtem verdoppelten Punkt ergeben sich somit $5 \cdot 17.9451 \cdot \frac{1\,439\,222}{1\,847} = 69\,916,034$ kippende Bits bei der Ereugung von 20 Zufallsbits. Dieses Ergebnis ist unter dem Aspekt der Leistungsaufnahme sehr schlecht und lässt den EC-Generator für einen Einsatz bei Chipkarten nicht als geeignet erscheinen.

[17] 4 Bit wurden deshalb gewählt, weil der BBS-Generator 2 Bits problemlos sicher ausgeben kann, aber im Gegensatz zum EC-Generator nur mit einer Koordinate arbeitet, der EC-Generator in der affinen Darstellung aber mit zwei.

7 Schluss

7.1 Zusammenfassung

In diesem Kapitel werden die in der Studienarbeit für die vier untersuchten Zufallszahlengeneratoren erzielten Ergebnisse vergleichend vorgestellt. Dabei wird für die einzelnen Anforderungspunkte (siehe Kapitel 4), soweit möglich, eine Reihenfolge der Zufallszahlengeneratoren bezüglich ihrer Eigenschaften angegeben werden. Untersucht wurden die Generatoren LFSR, BBS, RC4 und EC. Die genauen Einzelheiten finden sich in den Unterpunkten der Kapitel 5 und 6. Zu beachten ist bei diesen Aufstellungen aber, dass die Ergebnisse implementationsabhängig und teilweise gar nicht absolut vergleichbar sind, weil mehrdimensionale Einstellmöglichkeiten eine einfache Reihenfolge gar nicht zulassen. Dies ist aber bei den einzelnen Punkten vermerkt. Insofern dient dieses Kapitel als Überblick.

Zeitkomplexität

Die zeitlichen Anforderungen an einen Zufallszahlengenerator sind gerade beim Chipkarteneinsatz sehr wichtig, da ein Warten auf die Chipkarte bzw. das Warten auf die so genannte Technik im Allgemeinen von den Benutzern als unangenehm empfunden wird. Nach Möglichkeit sollen diese Vorgänge vom Empfinden her als zeitlos angesehen werden. Nach den in Kapitel 4.2 beschriebenen Einzelheiten bestehen alle Generatoren die Anforderungen. Es ergibt sich hierbei die Reihenfolge:

1. RC4 $= \mathbf{7,1424 \cdot 10^6 \frac{Bit}{s}}$
2. LFSR $= \mathbf{3,5712 \cdot 10^6 \frac{Bit}{s}}$
3. BBS $= \mathbf{119\,040 \frac{Bit}{s}}$
4. EC $= \mathbf{5\,910 \frac{Bit}{s}}$

Der RC4-Generator liefert in der 4-Bit-Version 4 Bit pro Takt, bei Nutzung einer Version, die mehr Bits ausgibt, entsprechend mehr, allerdings mit stark steigendem Speicherbedarf. Es lässt sich aber ohne weiteres die Geschwindigkeit beschleunigen. Der LFSR-Generator liefert pro Takt ein Bit unabhängig von der Größe des benutzten Polynoms. Es besteht aber die Möglichkeit der Parallelisierung (siehe [Schneier 96, S.434]), die jedoch mit höherem Speicher- und Rechenaufwand verbunden ist. Bei dem BBS- bzw. EC-Generator entstammt der Wert einer Abschätzung nach oben und der Annahme eines Modulus von 16 000 000 für BBS und von 30 000 für EC. Die allgemeinen Formeln finden sich in den Kapiteln 6.3 und 6.5. Hier sei auf die Möglichkeit von Akzeleratoren hingewiesen, die aber nichts an der Reihenfolge der Generatoren ändern würden, aber die Generierung beschleunigen würden.

Speicherbedarf

Der Speicherbedarf, den ein Zufallszahlengenerator erfordert, ist ein Ausschlußkriterium für den Einsatz auf einer Chipkarte, da gerade hier die Chipfläche begrenzt ist. Ein Generator, dessen Speicherzellenanforderungen zu hoch sind, ist für eine Chipkartenimplementation nicht tragbar. Die Ermittlung des Speicherzellenbedarfs ist in Kapitel 4.2 beschrieben, und die Einzelheiten zu den unterschiedlichen Generatoren findet sich im Kapitel 6. Allerdings hängt der Speicherzellenbedarf von mehreren Faktoren ab, das sind sowohl das benutzte Programm, die Implementation und insbesondere die verschiedenen Parameter, die die einzelnen Generatoren beeinflussen. Die genauen Abhängigkeiten von der Periode, der Ausgabebreite, dem eingesetzten Modulus oder der Besetzungszahl finden sich in den Kapiteln 6.2 bis 6.5. Für die Reihenfolge wurden die Einstellungen benutzt, die auch für die Statistiktests benutzt wurden. Damit ergibt sich:

$$\begin{array}{lll}
\textbf{1. LFSR} = f(30;4;20) & = \textbf{117 Speicherzellen}\ ; (2^{30}-1) \\
\textbf{2. BBS} \ \ = f(16\,834\,033;2) & = \textbf{359 Speicherzellen}\ ; (2\,101\,178) \\
\textbf{3. RC4} \ \ = f(20) & = \textbf{671 Speicherzellen}\ ; (10^{15}) \\
\textbf{4. EC} \ \ \ \ = f(33\,013;4) & = \textbf{707 Speicherzellen}\ ; (7\,388)
\end{array}$$

In den Klammern hinter dem Speicherzellenbedarf ist die Periode angegeben, die die Bitsequenzen der Generatoren mit den entsprechenden Einstellungen haben. Die genauen Zusammenhänge zwischen Periode[18] und Speicherbedarf finden sich in den Kapiteln 6.2 bis 6.4. Für den EC-Generator liegen diesbezüglich keine Zusammenhänge vor. Die Einzelheiten zu den allgemeinen Formeln zum Ermitteln des Speicherbedarfs sind auch in den Kapiteln 6.2 bis 6.5 zu finden. Der Speicherbedarf beim LFSR hängt hauptsächlich linear vom Polynomgrad ab, bei BBS und EC logarithmisch vom Modulus. Bei RC4 hängt der Speicherbedarf hauptsächlich von der Größe des Permutationsnetzwerkes ab, die 5-Bit-Version ist erheblich aufwändiger als die hier benutzte 4-Bit-Version, ansonsten ist der Speicherbedarf hier linear abhängig von der Ausgabebreite. Bei den vielfältigen Einstellmöglichkeiten sollten die Angaben des Speicherzellenbedarfs nicht absolut betrachtet werden, aber sie zeigen die oben angegebene Reihenfolge auf.

Leistungsaufnahme

Ein weiteres wichtiges Kriterium für einen Einsatz auf Chipkarten ist die Leistungsaufnahme, die hier in Form von kippenden Bits dargestellt wird. Die Einzelheiten finden sich wiederum in den Kapiteln 4.2 und 6.2 bis 6.5. Für die Erzeugung von 20-Bit-Zufallszahlen ergeben sich für die Generatoren mit den Einstellungen, die auch in den Statistiktests benutzt wurden, folgende Ergebnisse:

1. RC4 = im Mittel 39 kippende Bits
2. LFSR = im Mittel 300 kippende Bits
3. BBS = im Mittel 7 643 kippende Bits
4. EC = im Mittel 69 916 kippende Bits

RC4 und LFSR genügen hier ohne Probleme den Anforderungen, BBS sieht schon nach einer recht hohen Leistungsaufnahme aus, ob aber EC überhaupt bei der zu erwartenden hohen Anzahl an kippenden Bits noch für den Chipkarteneinsatz taugt, müsste von den entsprechenden Chipherstellern überprüft werden.

Qualität der erzeugten Zufallszahlen

Für den Sicherheitsaspekt ist natürlich wichtig, wie zufällig die erzeugten Zahlen aussehen. Bedenklich ist der LFSR-Generator wegen des effizienten *Berlekamp-Massey-Algorithmus'* (siehe Kapitel 5.1). Der theoretisch beste Generator ist BBS (siehe Kapitel 5.1). Zu EC existieren in der hier vorgestelleten Form keine theoretischen Grundlagen. RC4 wurde von Ron Rivest bei RSA Data Security Inc. entwickelt, die Namen sprechen für sich, aber es ist nicht zu vergessen, dass es keine öffentlichen Texte zur Kryptoanalyse gibt (siehe [Schneier 96, S. 455f.]).

Allen Generatoren gemeinsam ist aber, dass sie die in Kapitel 3.3 beschriebenen Tests bestehen. Einzelheiten zu kritischen Werten sind in den Kapiteln 5.1 bis 5.4 zu finden.

[18]Die Periode beim BBS-Generator ist hier ein Fall der halben Periode, also p_2q_2 statt $2p_2q_2$.

Fazit

Abschließend lässt sich sagen, dass RC4 der für eine Chipkartenimplementation am besten geeignete Zufallszahlengenerator ist, auch wenn der Speicherbedarf das Gesamtbild etwas trübt und auch das Patent auf diesem Verfahren mit Kosten verbunden ist. *Am schlechtesten schneidet der EC-Generator ab.* Welcher Generator nun am besten für eine Implementation taugt, hängt aber trotzdem von den einzelnen Anforderungen ab, die ja sehr verschieden sein können. Nicht zu vergessen trotz der gerade mal durchschnittlichen Werte des BBS-Generators sind die Eigenschaften des wahlfreien Zugriffs auf die Zufallszahlensequenz und die hohe Sicherheit. Das LFSR ist für eine Kombination mit einem andern Generator oder innerhalb eines geschlossenen Systems eine Alternative.

7.2 Ausblick, Perspektiven

In diesem abschließendem Kapitel werden weiterführende Fragestellungen vorgestellt, die sich aus dieser Studienarbeit ergeben. Der Grund, weshalb diese Fragen nicht an dieser Stelle geklärt werden, ist, dass sie den Umfang der Studienarbeit sprengen würden und sich für weitere wissenschaftliche Arbeiten eignen.

Optimierung der Implementation

Diese Studienarbeit ist auf die Bewertung verschiedener Zufallsgeneratoren und die Realisierbarkeit dieser Generatoren auf Chipkarten fokussiert. Es wurde eine exemplarische Implementation durchgeführt, um die grundsätzliche Umsetzung zu untersuchen.

Bei der Implementierung wurde aber nicht untersucht, inwieweit sich die Algorithmen zur Erzeugung von Zufallszahlen bezüglich des Zeit- und Platzbedarfs optimieren lassen. Zeitoptimierungen sind vor allem durch Parallelisierung einzelner Berechnungsschritte möglich. Insbesondere trifft das bei dem Generator zu, der elliptische Kurven benutzt, weil dort viele voneinander unabhängige Rechnungen durchgeführt werden, die durch Nebenläufigkeit deutlich schneller als in der sequentiellen Implementation dieser Studienarbeit sind. Interessant ist in diesem Zusammenhang auch die Fragestellung, welcher Grad an paralleler Verarbeitung welchen Hardware- und Zeitaufwand verursacht.

Weiterhin wäre es wünschenswert, die Implementationen in der Programmiersprache C so zu verbessern, dass sie – wie die VHDL-Implementation – beliebige Wortlängen zulassen und nicht durch die Maschinenwortbreite (meist 32 Bit) beschränkt sind. Dazu könnte beispielsweise die GNU-MP-Library `gmp`[19] verwendet werden, die das Rechnen mit beliebig großen Zahlen gestattet.

Parametrisierbare Chipkarten

Es ist ebenfalls wünschenswert, die Zufallsgeneratoren so zu implementieren, dass sie sich für so genannte parametrisierbare Chipkarten eignen. Parametrisierbare Chipkarten sind Chipkarten, auf denen der Endanwender Einstellungen vornehmen kann, beispielsweise die Änderung des Startwerts oder das Überspringen einer beliebigen Anzahl an Zufallszahlen.

Weitere Zufallsgeneratoren mit Elliptischen Kurven

Es gibt verschiedene Möglichkeiten, mit elliptischen Kurven Zufallszahlen zu erzeugen. In dieser Studienarbeit wurde ein Zufallsgenerator vorgestellt, der einen Punkt auf der Kurve fortlaufend verdoppelt. Eine Untersuchung anderer Verfahren zur Erzeugung von Zufallszahlen mit elliptischen Kurven wäre interessant, ist aber an dieser Stelle zu umfangreich. Ein Algorithmus ist beispielsweise zu finden in [Gong 98].

[19] zu finden unter `http://www.swox.com/gmp`

A Inhalt der CD

Zu dieser Studienarbeit wurde eine CD-ROM erstellt, auf der die verschiedenen Implementationen, die zitierten Internetseiten und weitere Daten zu finden sind. Die CD-ROM ist so gestaltet, dass mit einem beliebigen Internetbrowser durch die CD navigiert werden kann. Dazu muss lediglich die Datei `index.html` mit einem Browser geöffnet werden (unter dem Betriebssystem „Windows“ geschieht das automatisch).

Zur Übersicht werden nun die Verzeichnisse mit den enthaltenen Dateien vorgestellt; die HTML-Dateien, die zur Navigation enthalten sind, werden der Einfachheit halber weggelassen:

- Verzeichnis `Beispiel` (siehe Anhang B.1)

`RC4.EXE`	Ausführbarer RC4-Zufallsgenerator (für „Windows“).
`RC4_Ausgabe.txt`	Bildschirmausgabe des RC4-Zufallsgenerators.
`STATISTIK.EXE`	Ausführbarer Statistik-Test des RC4-Zufallsgenerators.
`Statistik_Ausgabe.txt`	Bildschirmausgabe der RC4-Statistik-Untersuchung.
`STATISTIK-KIPPBIT.EXE`	Ausführbarer Test der kippenden Bits des RC4-Zufallsgenerators.
`Statistik-Kippbit_Ausgabe.txt`	Bildschirmausgabe der Untersuchung der kippenden Bits des RC4-Zufallsgenerators.

- Verzeichnis `C-Implementation`

`bbs-double.c`	Double-Variante des BBS-Generators.
`bbs-int.c`	Integer-Variante des BBS-Generators.
`BBS-Periode.txt`	Ergebnisse der Untersuchung der Periodengröße des BBS-Generators.
`BBS-Test.txt`	Exemplarischer Test der Periode des BBS-Generators.
`Bitblock.h`	Prozeduren zum Laden und Speichern einzelner Bits in Binärdateien.
`Blumzahlen.c`	Programm zur Erzeugung von Blumschen Zahlen. Diese werden für den BBS-Generator benötigt.
`blumzahlen.txt`	Liste aller Blumschen Zahlen bis zu 1 000 000.
`EC.c`	Der Zufallsgenerator mit elliptischen Kurven.
`LFSR.c`	Linear rückgekoppeltes Schieberegister als Zufallsgenerator.
`Makefile`	Datei mit den Übersetzungsregeln für den C-Compiler; wird vom `make`-Befehl benutzt.
`RC4.c`	RC4-Zufallsgenerator.
`RC4-periode.c`	Modifizierter RC4-Zufallsgenerator, der nach der Periode sucht.

`Statistik.c`	Programm zur statistischen Untersuchung von Zufallszahlen.
`Statistik-Kippbit.c`	Programm zur Untersuchung der kippenden Bits zwischen aufeinanderfolgenden Zuständen eines Zufallsgenerators.

➪ Verzeichnis `Text`

`Arbeit.pdf`	Studienarbeit im PDF-Format (Achtung: Einige Formatierungen sind dabei nicht korrekt!).
`Arbeit.ps`	Studienarbeit im Postscript-Format.
`LaTeX` (Verzeichnis)	LaTeX-Quelldatei und Abbildungen. Die Bilder wurden mit `xfig` erstellt; die `.fig`-Dateien sind hier neben den `.eps`-Dateien ebenfalls zu finden.

➪ Verzeichnis `VHDL-Implementation`

`bbs_pkg.vhd`	Konfigurationsdatei des BBS-Generators
`bbs.vhd`	Implementation des BBS-Generators
`tb_bbs.vhd`	Testumgebung zur Simulation des BBS-Generators
`ec_pkg.vhd`	Konfigurationsdatei des EC-Generators
`ec.vhd`	Implementation des EC-Generators
`tb_ec.vhd`	Testumgebung zur Simulation des EC-Generators
`lfsr_pkg.vhd`	Konfigurationsdatei des LFSR-Generators
`lfsr.vhd`	Implementation des LFSR-Generators
`tb_lfsr.vhd`	Testumgebung zur Simulation des LFSR-Generators
`rc4_pkg.vhd`	Konfigurationsdatei des RC4-Generators
`rc4.vhd`	Implementation des RC4-Generators
`tb_rc4.vhd`	Testumgebung zur Simulation des RC4-Generators

➪ Verzeichnis `VHDL-Simulation`

`bbs_waves.pdf`	Blum-Blum-Shub-Generator mit den Parametern $n = 1081$ (Modulus) und $x = 42$ (Startwert) im PDF-Format.
`bbs_waves (quer).pdf`	Dasselbe im Querformat.
`bbs_waves.ps`	Dasselbe als Postscriptdatei.
`ec_waves.pdf`	Zufallsgenerator mit elliptischen Kurven. Die Parameter sind $p = 33013$ (Modulus), $a = 2155$, $b = 16008$ (Kurvenparameter) und der Startpunkt $x = 3$, $y = 150$ (PDF-Format).

`ec_waves (quer).pdf`	Dasselbe im Querformat.
`ec_waves.ps`	Dasselbe als Postscriptdatei.
`lfsr_waves.pdf`	Linear rückgekoppeltes Schieberegister mit dem Polynom $P(x) = x^{32} + x^7 + x^6 + x^2 + x^0$ (=0x80000062) und dem Startwert 0xAAAAAAAA (PDF-Format).
`lfsr_waves (quer).pdf`	Dasselbe im Querformat.
`lfsr_waves.ps`	Dasselbe als Postscriptdatei.
`rc4_waves.pdf`	RC4-Zufallsgenerator mit dem Standardschlüssel (PDF-Format).
`rc4_waves (quer).pdf`	Dasselbe im Querformat.
`rc4_waves.ps`	Dasselbe als Postscriptdatei.

➪ Verzeichnis `Internetseiten`

`citeseer.nj.nec.com`	Die Quelle, die auf den Artikel von Guang Gong, Thomas A. Berson, Douglas R. Stinson ("Elliptic Curve Pseudorandom Sequence Generators", 1998) verweist. Im Literaturverzeichnis der Studienarbeit ist der Artikel als [Gong 98] bezeichnet.
`stat.fsu.edu`	Programm zur Untersuchung der Unabhängigkeit von Zufallszahlen.
`valley.interact.nl`	Frei käuflicher hardwaremäßiger Zufallszahlengenerator.
`www.protego.se`	Ein weiterer frei käuflicher hardwaremäßiger Zufallszahlengenerator.
`www.swox.com`	Die GNU-MP-Library `gmp` für Berechnungen mit beliebig großen Zahlen.
`www.contestcen.com`	Text zur Untersuchung des BBS-Generators.

B Beispielsitzungen

B.1 Schnelles Ausprobieren

Zum bequemen Nachvollziehen der Arbeitsweise der Zufallsgeneratoren wird nun das Beispiel aus Kapitel 4.4 (Abschnitt „Durchführung der Tests mit Rechnerunterstützung“, S. 34) mithilfe der CD-ROM nachvollzogen. Dort wurde der 4-Bit-RC4-Zufallsgenerator unter Benutzung des Standardschlüssels untersucht.

Die einzelnen Schritte zur Erzeugung und Untersuchung der Zufallszahlen werden im Folgenden dargestellt. Sämtliche notwendigen Dateien befinden sich im Verzeichnis `Beispiel` der beiliegenden CD (vgl. Anhang A). Die vorkompilierten Programme laufen nur unter dem Betriebssystem „Windows“; für andere Betriebssysteme müssen die Quelltexte einfach übersetzt werden.

1. Kopieren Sie die Dateien aus dem Verzeichnis `Beispiel` der beiliegenden CD auf die lokale Festplatte, und wechseln Sie in das lokale Verzeichnis, in das Sie die Dateien kopiert haben.

2. Starten Sie das Programm `RC4.EXE`, um die Zufallszahlen mit dem RC4-Generator zu erzeugen. Die Zahlen aus Tabelle 2 (Seite 34) werden ausgegeben und in die Datei `rc4.zz` geschrieben. Die durchlaufenen Zustände werden in der Datei `rc4-zustand.log` gespeichert.

3. Starten Sie das Programm `STATISTIK.EXE`. Es erscheint das Ergebnis der Statistiktests, bei denen die bit- und blockweisen Wahrscheinlichkeiten geprüft werden. Die Ausgabe entspricht derjenigen aus Kapitel 4.4 (Abschnitt „Durchführung der Tests mit Rechnerunterstützung“, S. 34ff.).

4. Starten Sie das Programm `STATISTIK-KIPPBIT.EXE`. Es untersucht, wie viele Bits bei der Erzeugung einer neuen Zufallszahl kippen, wie in Kapitel 4.3 beschrieben.

B.2 Beispiel: Entwurf eines Schieberegisters

Um mit den Implementationen tiefer gehend vertraut zu werden, wird nun exemplarisch ein lineares Schieberegister entworfen, mit C simuliert und anschließend mit VHDL umgesetzt. Dazu sind ein C-Compiler und ein VHDL-Compiler notwendig (hier werden `gcc` bzw. `vhdlan` benutzt).

Das Schieberegister soll 20 Bit breit sein und 1 Bit breite Zufallszahlen erzeugen; der Startwert des Registers soll 0101 0101 0101 0101 0101 sein. Dazu wird das primitive Polynom $P(x) = x^{20} + x^3 + x^0$ gewählt.

Umsetzung in C

Zuallererst muss der C-Quelltext von der CD-ROM auf Festplatte kopiert werden, weil sonst die Dateien nicht zurückgeschrieben werden können. Notwendig sind die Dateien `Bitblock.h`, `LFSR.c`, `Makefile`, `Statistik.c` und `Statistik-Kippbit.c` aus dem Verzeichnis `C-Implementation`.

Der C-Quellcode `LFSR.c` muss nach den gestellten Anforderungen angepasst werden. Dazu müssen folgende Einstellungen vorgenommen werden (die Einstellungen sind im Detail in Kapitel 6 und im C-Quelltext beschrieben):

```
#define BITBREITE 20
#define ANFANGSZUSTAND 0xaaaaa          // hexadezimaler Anfangszustand
#define POLYNOM 0x80004                 // Polynom
#define AUSGABEHOCH 1                   // Ausgabebits
#define AUSGABENIEDRIG 1
#define EINMAL_SCHIEBEN false
#define ANZAHL ((double) 1048576)
#define TESTE_PERIODE true
```

```
#define DATEIAUSGABE true                // Zahlen in Datei schreiben
#define DATEINAME "lfsr.zz"
#define KONSOLE false
#define ZUSTANDLOG true                  // kippende Bits untersuchen
#define LOGDATEINAME "lfsr-zustand.log"
```

Nach dem Speichern wird das Programm mit dem Befehl `make LFSR` kompiliert und kann mit dem Aufruf `LFSR` gestartet werden. Das Programm gibt folgendes aus:

```
Periode gefunden bei 1048575.
Warnung: Letztes Byte mit 4 Nullen aufgefüllt.
```

Da die Ausgabe auf der Konsole abgeschaltet wurde, werden die Zufallszahlen nicht ausgegeben; sie werden aber in der Datei `lfsr.zz` gespeichert. Der Zufallsgenerator gibt die erwartete Periode von $2^n - 1 = 1048575$ aus (Achtung: Wenn die Ausgabebitbreite erhöht wird und `EINMAL_SCHIEBEN` auf `true` gesetzt ist, sinkt die Periode, weil hier nicht die Zahl der Bits, sondern die Anzahl der erzeugten Zufallszahlen (also Blöcke) angegeben wird). Zusätzlich weist das LFSR-Programm darauf hin, dass die Ausgabedatei mit Nullen aufgefüllt werden musste, weil die Zufallszahlen nicht exakt in die Datei gepasst haben, da die Anzahl der erzeugten Bits nicht durch 8 teilbar war.

Statistische Untersuchung

Um die zuvor erzeugten Zufallszahlen zu untersuchen, muss der C-Quelltext `Statistik.c` konfiguriert werden, und zwar so:

```
#define DATEINAME "lfsr.zz"
#define BITBREITE 1
#define KONSOLE false
```

Durch die Eingabe von `make Statistik` wird der Quellcode kompiliert, und mit dem Befehl `Statistik` wird das Programm aufgerufen. Die Ausgabe sieht folgendermaßen aus:

```
Warnung: Die letzten 8 Einträge könnten aufgefüllte Nullen sein.
1048576 Zufallszahlen aus der Datei lfsr.zz geladen (Bitbreite=1 Bit).


*********** TEST: Anzahl der Einsen, aus denen die Zufallszahl bestehen ***********

Anz. Einsen|theoret. Häuf.|  Häufigkeit |th.Häuf.-Häuf. |approx. Normalvert. |Abw. Häuf.-Normalvert.
-----------|--------------|-------------|---------------|--------------------|----------------------
      0    |    524288.000|       524287|          1.000 |          507448.421 |     16838.579
      1    |    524288.000|       524289|         -1.000 |          507450.357 |     16838.643
-----------|--------------|-------------|---------------|--------------------|----------------------
   SUMME   |   1048576.000|      1048576|          0.000 |         1014898.778 |     33677.222

Chi-Quadrat-Test:
Chi^2 =   0.000 mit n=2 Freiheitsgraden

Theoretische Verteilung:
Erwartungswert     = 0.5000
Standardabweichung = 0.5000

Approximierte Normalverteilung (d.h. tatsächliche Verteilung):
Stichprobenmittel             = 0.5000 +/- 0.0016 Einsen
Stichproben-Standardabweichung = 0.5000 Einsen


*********** TEST: Häufigkeit der Zufallszahlen ***********

 Bitkombination | theoret. Häuf. |   Häufigkeit   |    Differenz
----------------|----------------|----------------|---------------
              0 |     524288.000 |         524287 |          1.000
```

```
             1 |     524288.000 |        524289 |         -1.000
----------------|----------------|---------------|---------------
      SUMME     |     1048576    |       1048576 |           0

Chi-Quadrat-Test:
Chi^2 =   0.000 mit n=2 Freiheitsgraden
```

Um die kippenden Bits des LFSR-Generators zu untersuchen, muss das Programm `Statistik-Kippbit.c` angepasst werden:

```
#define BITBREITE 20
#define ANZAHL_VAR 1
#define LOGDATEINAME "lfsr-zustand.log"
```

Das Programm wird mit `make Statistik-Kippbit` kompiliert und mit `Statistik-Kippbit` aufgerufen. Es liefert diese Ausgabe:

```
Warnung: Ignoriere 4 Bits am Dateiende.
1048577 Zustände zu je 1 Variablen aus der Datei lfsr-zustand.log geladen (Bitbreite=20 Bit).


*********** TEST: Anzahl der kippenden Bits bei Zustandswechsel ***********

 Anz. Wechsel |    Häufigkeit   | approx. Normalvert.
--------------|---------------|--------------------
       0      |             0 |          8.493
       1      |            20 |         56.784
       2      |           190 |        310.835
       3      |          1140 |       1393.068
       4      |          4845 |       5111.592
       5      |         15504 |      15356.114
       6      |         38760 |      37770.057
       7      |         77520 |      76059.793
       8      |        125970 |     125401.817
       9      |        167960 |     169275.345
      10      |        184756 |     187078.884
      11      |        167960 |     169276.636
      12      |        125970 |     125403.731
      13      |         77520 |      76061.534
      14      |         38760 |      37771.209
      15      |         15504 |      15356.699
      16      |          4845 |       5111.826
      17      |          1140 |       1393.142
      18      |           190 |        310.854
      19      |            20 |         56.788
      20      |             2 |          8.494
--------------|---------------|---------------------
     SUMME    |   1048576.000 |      1048573.693

Approximierte Normalverteilung (d.h. tatsächliche Verteilung):
Stichprobenmittel              = 10.0000 +/- 0.0072 Einsen
Stichproben-Standardabweichung = 2.2361 Einsen
```

Umsetzung in VHDL

Um das Schieberegister in VHDL umzusetzen, müssen die Quelltexte `lfsr_pkg.vhd`, `lfsr.vhd` und `tb_lfsr.vhd` von der CD-ROM auf Festplatte kopiert werden, weil die Dateien geändert werden.
Zur Einstellung des Schieberegisters muss die Datei `lfsr_pkg.vhd` wie folgt parametrisiert werden:

```
constant BITBREITE:         integer := 20;
constant STARTWERT:         std_logic_vector((BITBREITE-1) downto 0) := "10101010101010101010";
constant POLYNOM:           std_logic_vector((BITBREITE-1) downto 0) := "10000000000000000100";
constant AUSGABEHOCH:       integer := 1;
constant AUSGABENIEDRIG:    integer := 1;
constant AUSGABEBREITELOG: integer := 1;
```

Vor dem Kompilieren muss unter Unix das Verzeichnis `WORK` angelegt werden. Dann werden die VHDL-Programme mit `vhdlan lfsr_pkg.vhd` und `vhdlan lfsr.vhd` übersetzt. Die Testumgebung für die Simulation wird mit `vhdlan tb_lfsr.vhd` kompiliert. Der SYNOPSYS-Simulator wird mit `vhdldbx` gestartet.

Literaturverzeichnis

[Agnew 87] G. B. Agnew: *Random Sources for Cryptographic Systems*, Advances in Cryptology - EUROCRYPT 87 Proceedings, Springer 1988

[Bauer 97] Friedrich L. Bauer: *Entzifferte Geheimnisse*, Springer-Verlag, 2. Auflage 1997

[Bartsch 97] Hans-Jochen Bartsch: *Taschenbuch mathematischer Formeln*, Fachbuchverlag Leipzig, 17. Auflage 1997

[Barnick] Katrin Barnick: *Chip statt Magnetstreifen*, Entertainment Markt, Nr. 12, 15. Juni 2001

[Biggs 89] Norman L. Biggs: *Discrete Mathematics*, Oxford University Press, Revised Edition 1989

[Blum 86] L. Blum, M. Blum, M. Shub, *A Simple Unpredictable Pseudo-Random Number Generator*, Siam Journal on Computing, V. 15, N. 2, 1986, S. 364-383

[Fairfield 84] R. C. Fairfield, A. Matusevich, J. Plany: *An LSI Digital Encryption Processor (DEP)*, Advances in Cryptology: Proceedings of CRYPTO 84, Springer 1985

[Gong 98] Guang Gong, Thomas A. Berson, Douglas R. Stinson: *Elliptic Curve Pseudorandom Sequence Generators*, 1998, `http://citeseer.nj.nec.com/65929.html`

[Härtel 94] Frank Härtel: *Zufallszahlen für Simulationsmodelle*, Difo-Druck GmbH, Bamberg 1994

[Hogben 63] Lancelot Hogben: *Die Entdeckung der Mathematik - Zahlen formen ein Weltbild*, Chr. Belser Verlag Stuttgart 1963

[Hübner 96] Gerhard Hübner: *Stochastik*, Viewig & Sohn Verlagsgesellschaft, Braunschweig/Wiesbaden 1996

[IEEE P1363 / D13] IEEE P1363 / D13: *Standard Specifications for Public Key Cryptography*, Anhang A, Number-Theoretic Background, 1999

[Kahn 96] David Kahn, THE Codebreakers: *The Story Of Secret Writing*, SCRIBNER 1996

[Knuth 98] Donald Erwin Knuth: *The Art Of Computer Programming - Third Edition, Volume 2 – Seminumerical Algorithms*, Addison Wesley Longman, 3. Auflage 1998

[Lehmer 51] D. H. Lehmer: *Proc. 2nd Symposium on Large-Scale Digital Computing Machinery*, Cambridge: Harvard University Press 1951

[Massey 69] James L. Massey: *Shift-Register Synthesis and BCH Decoding*, IEEE Transactions on Information Theory, V. it-15, N. 1, 1969, S. 122-127

[Massey 69] James L. Massey: *Cryptography and System Theory*, Proceedings of the 24th Allerton Conference on Communicaton, Control and Computers, 1986, S. 1-8

[Maurer 90] Ueli M. Maurer: *A Universal Statistical Test for Random Bit Generators*, A.J. Menezes, S.A.Vanstone (Hrsg.): Advances in Cryptology - CRYPTO '90 Proceedings, Springer-Verlag Heidelberg 1991, S. 409 - 420

[Neuhaus 95] Klaus Behnen, Georg Neuhaus: *Grundkurs Stochastik - Eine integrierte Einführung in Wahrscheinlichkeitstheorie und mathematische Statistik*, Teubner, 3. Auflage 1995

[Odlyzko 84] A. Odlyzko: *Discrete Logarithms in Finite Fields and Their Cryptographic Significance*, Advances in Cryptology: Proceedings of EUROCRYPT 84, Springer 1985

[Rabin 68] M. O. Rabin: *Probabilistic Algorithms in Finite Fields*, SIAM Journal on Computing, V. 9, N. 2, 1980, S. 273-280

[Rabin 76] M. O. Rabin: *Probabilisic algorithms*, New directions and recent results in algorithms and complexity, 1976

[RAND 55] RAND Corporation: *A Million Random Digits with 100.000 Normal Deviates*, Glencoe, IL: Free Press Publishers, 1955

[Rankl 99] Wolfgang Rankl, Wolfgang Effing: *Handbuch der Chipkarten*, Hanser Verlag, 3. Auflage 1999

[Rechenberg 97] Gustav Pomberger, Peter Rechenberg: *Informatik-Handbuch*, Hanser 1997

[Richter 92] Manfred Richter: *Ein Rauschgenerator zur Gewinnung von quasi-idealen Zufallszahlen für die stochastische Simulation*, TU Aachen 1992, Dissertation

[RSA 78] Ron Rivest, Adi Shamir, Leonard Adleman: *A Method for Obtaining Digital Signatures and Public-Key Cryptosystems*, Communications of the ACM, V. 21, N. 2, 1978

[Rueppel 86] Rainer A. Rueppel: *Analysis and Design of Stream Ciphers*, Springer Verlag 1986

[Schiffmann 96] Wolfram Schiffmann, Robert Schmitz: *Technische Informatik 1 - Grundlagen der digitalen Elektronik*, Springer, 3. Auflage 1996

[Schneier 96] Bruce Schneier: *Angewandte Kryptographie*, Addison-Wesley 1996

[Selmer 66] E. S. Selmer: *Linear Recurrence over Finite Field*, University of Bergen, Norwegen, 1966

[Shannon 49] Claude E. Shannon, *The Communication Theory of Secrecy Systems*, Bell System Technical Journal, 1949

[Tanenbaum 96] Andrew S. Tanenbaum: *Computernetzwerke*, Prentice Hall, 3. Auflage 1996

[Vazirani 84] U.V. Vazirani, V.V. Vazirani, *Efficient and Secure Pseudo-Random Number Generation*, Proceedings of the 25th Symposium on the Foundations of Computer Science, 1984, S. 458-463

Erklärung

Hiermit erklären wir, dass wir die vorliegende Studienarbeit selbstständig erstellt und keine anderen Hilfsmittel als die angegebenen verwendet haben.

Peter Hartmann

Stefan Witt