Systematische Entwurfsmethoden für praktikable Kryptosysteme

Dissertation zur Erlangung des Doktorgrades
der Mathematisch-Naturwissenschaftlichen Fakultäten
der Georg-August-Universität zu Göttingen

vorgelegt von
Stefan Lucks
aus Lüdenscheid

Göttingen, 1997

Die Deutsche Bibliothek - CIP-Einheitsaufnahme

Lucks, Stefan:
Systematische Entwurfsmethoden für praktikable Kryptosysteme /
vorgelegt von Stefan Lucks. 1. Aufl. - Göttingen : Cuvillier, 1997
Zugl.: Göttingen, Univ., Diss., 1997
ISBN 3-89588-968-7

D 7

Referent:	Prof. Dr. St. Waack
Korreferent:	Prof. Dr. R. M. Switzer
Tag der mündlichen Prüfung:	24.04.1997

Nonnenstieg 8, 37075 Göttingen
Telefon: 0551-54724-0
Telefax: 0551-54724-21

1. Auflage, 1997
Gedruckt auf säurefreiem Papier

ISBN 3-89588-968-7

Inhaltsverzeichnis

1 Einleitung

Ich weiß nicht, was soll es bedeuten [...] – H. Heine

In der Vergangenheit galt die Kryptographie als die Lehre der Methoden, den Inhalt vertraulicher Botschaften durch Verschlüsseln zu verbergen.[1] Begrifflich unterschied man davon die Kryptoanalyse, d.h. den Versuch, die Verschlüsselung zu brechen, und die Kryptologie als Sammelbegriff für Kryptoanalyse und Kryptographie.

Schon in der Antike entwickelten die Menschen Verfahren, um ihre Botschaften vor unbefugter Einsichtnahme, einem sogenannten „Angriff", zu schützen. Bis in die frühe Neuzeit hinein dominierten steganographische Methoden, mit denen man versuchte, sogar die Existenz einer geheimen Botschaft zu verbergen. Beispiele dafür sind die Verwendung von Geheimtinte und die Absprache unverdächtiger Formulierungen als Stichworte für bestimmte Geheimnisse. Daneben verwendete man aber auch Permutations- und Substitutionschiffren und entwickelte Methoden, diese Chiffren zu brechen.

Erst im 19. Jahrhundert wurde die Kryptologie systematisiert und formalisiert. Im 20. Jahrhundert beschleunigte die maschinelle Datenverarbeitung und -übermittlung die Entwicklung der Kryptologie. So gelang es den Briten während des zweiten Weltkrieges, die Chiffre der deutschen Enigma, einer Familie elektromechanischer Verschlüsselungsmaschinen, zu brechen. Der britische Geheimdienst verwendete Spezialrechner, die eigens für diesen einen Zweck konstruiert worden waren.

1.1 Die moderne Kryptographie

Cryptography is about communication in the presence of adversaries. – R. Rivest

Früher war die Kryptographie vor allem für Militärs, Geheimdienste und Diplomaten wichtig. Das hat sich in jüngster Zeit drastisch geändert! Ausschlaggebend dafür sind die zunehmende Bedeutung von Computern und deren Vernetzung, besonders aber der Aufbau eines weltweiten Computernetzwerkes, des Internet.

Viele Internet-Nutzer „ahnen nicht, welche Spur sie mit jedem Mausklick hinterlassen und wie einfach es ist, Computer anzuzapfen" [30]. Nationale Datenschutzmaßnahmen sind in einem weltweiten Computernetzwerk wirkungslos. Mit juristischen oder organisatorischen Mitteln allein kann man keine Abhilfe schaffen, zumal das Internet dezentral

[1] Auf griechisch bedeutet „krýptein" soviel wie „verbergen".

organisiert ist, niemandem gehört und in keinem Staat ansässig ist. Nutzer und Anbieter sind trotzdem nicht völlig hilflos den „Räubern im Netz“ [30] ausgeliefert. Oft ist es möglich, sich in Eigeninitiative zu schützen, z.B. indem sich die Kommunikationsteilnehmer auf den Einsatz kryptographischer Verfahren verständigen. Die Kryptographie bietet hier zwar kein Allheilmittel, ist aber eine Schlüsseltechnologie für die Realisierung sicherer Kommunikation – auch in anderen Medien als dem Internet.

In der Vergangenheit dienten, wie bereits erwähnt, kryptographische Methoden der Geheimhaltung des Inhaltes vertraulicher Botschaften. Dagegen ist heute die Kryptographie als weit umfassendere Wissenschaft anzusehen. Der Einsatz kryptographischer Verfahren dient heute meist einem oder mehreren der folgenden Ziele:

- Der Geheimhaltung von Daten.
- Der Sicherstellung der Authentizität von Daten.
- Der Authentifizierung von Kommunikationsteilnehmern.
- Der Ermöglichung anonymer Kommunikation.

Alle Algorithmen und Protokolle, die der Erreichung dieser Ziele dienen, bezeichnen wir als *Kryptosysteme*. Zunehmend bürgert es sich heute ein, die Kryptoanalyse als Teilgebiet der Kryptographie zu sehen, also nicht mehr zwischen Kryptographie und Kryptologie zu unterscheiden. Entsprechend verfahren wir auch in der vorliegenden Arbeit.

1.2 Der Entwurf von Kryptosystemen: Drei methodische Ansätze

Cryptography [...] is finally, slowly, and painfully becoming a science. – J. Leichter

In der modernen Kryptographie kennt man drei Ansätze, die *Sicherheit* eines Kryptosystems zu beschreiben: Den informationstheoretischen, den komplexitätstheoretischen und den systembasierten Ansatz.

Der **informationstheoretische Ansatz** bietet unbedingte Sicherheit. Das heißt, nicht einmal Angreifer mit unbeschränkter Rechenkapazität können ein informationstheoretisch sicheres Kryptosystem brechen. Die Vernam-Chiffre, siehe Abschnitt 2.2, ist informationstheoretisch sicher, jedoch fast immer unpraktikabel. Überhaupt führt der informationstheoretische Ansatz nur selten zu praktikablen Kryptosystemen.[2]

Probleme, die ein Angreifer mit unbeschränkter Rechenkapazität theoretisch lösen kann, können für reale Angreifer praktisch unlösbar sein. Auf dieser Überlegung basiert der

[2] Auf einzelnen Teilgebieten der Kryptographie hat der informationstheoretische Ansatz sehr wohl zu unbedingt sicheren *und* praktikablen Kryptosystemen geführt. Dies gilt insbesondere für *Secret Sharing* Systeme und *Authentication Codes* (siehe Kapitel 10 und 11 in [67]).

komplexitätstheoretische Ansatz. Die Komplexitätstheorie betrachtet Probleme und versucht, die inhärente Schwierigkeit dieser Probleme zu bestimmen. Bezüglich eines Rechenmodells bezeichnet man die mindestens erforderlichen Ressourcen zum Lösen eines Problems als die *Komplexität* (also die *‚Schwierigkeit'*) des Problems. Übliche Berechnungsmodelle sind z.B. Turingmaschinen und Boolesche Schaltkreise. Das Ziel des komplexitätstheoretischen Ansatzes in der Kryptographie besteht darin, Angriffe auf Kryptosysteme zu formalisieren und Kryptosysteme zu finden, die anzugreifen beweisbar *praktisch unmöglich* ist. Üblicherweise betrachtet man in der Komplexitätstheorie Kryptosysteme in Abhängigkeit von einem Sicherheitsparameter. Ein Kryptosystem gilt als *sicher*, wenn es nur mit überpolynomiellem Ressourceneinsatz gebrochen werden kann. Eine Wahrscheinlichkeit gilt als *signifikant*, wenn es ein Polynom gibt, dessen Kehrwert schneller gegen Null geht als die Wahrscheinlichkeit. Wir sprechen in diesem Zusammenhang von der *asymptotischen Sichtweise.*

Der **systembasierte Ansatz** ist eher pragmatischer Natur. Beruhend auf „Versuch und Irrtum" (Preneel [50], Abschnitt 1.4.3) haben sich bestimmte kryptanalytische Prinzipien herausgebildet, die zu erfolgreichen Angriffen auf ältere Kryptosysteme geführt haben. Beim Entwurf neuer Kryptosysteme versucht man, geeignete Lehren aus dem Scheitern der älteren Systeme zu ziehen. Kryptanalytische Prinzipien, die in den letzten Jahren zu erfolgreichen Angriffen geführt haben, sind z.B. die differentielle und die lineare Kryptanalyse, siehe dazu den Übersichtsartikel von Schneier [59].

Wenn der **informationstheoretische Ansatz** zu praktikablen Kryptosystemen führt, ist er den konkurrierenden Ansätzen vorzuziehen. Dieser Fall ist jedoch eine seltene Ausnahme. Sehr oft sind die praktischen Anforderungen an ein Kryptosystem so, daß man sogar beweisen kann, daß es kein informationstheoretisch sicheres Kryptosystem gibt, welches die gestellten Anforderungen erfüllt. Ein einfaches Beispiel, das schon auf Shannon [64] zurückgeht, ist die Forderung, *‚viel Information'* verschlüsselt mit Hilfe eines *‚kleinen Schlüssels'* über einen unsicheren Kanal zu übermitteln. Im folgenden behandeln wir den informationstheoretischen Ansatz nicht weiter. Allerdings werden bei den Beweisen in Abschnitt 4 teilweise Methoden aus der Informationstheorie angewendet.

Wegen seiner pragmatischen Vorgehensweise, die auf Versuch und Irrtum basiert, kann man von dem **systembasierten Ansatz** kaum sagen, er führe zu systematischen Entwurfsmethoden für Kryptosysteme. Allerdings wird auch bei Anwendung des systembasierten Ansatzes oft versucht, die Frage nach der Sicherheit des entworfenen Kryptosystems mit formalen Methoden zu behandeln. So beweist Lai [27], daß die von ihm entworfene IDEA-Blockchiffre nicht durch differentielle Kryptanalyse gebrochen werden kann. Der prinzipielle Nachteil des systembasierten Ansatzes besteht darin, daß man beim Entwurf neuer Kryptosysteme stets nur auf bekanntgewordene kryptanalytische Prinzipien *reagiert*, während der informationstheoretische und der komplexitätstheoretische Ansatz geeignet sind, auch neue kryptanalytische Methden *vorwegzunehmen.* Trotzdem ist der systembasierte Ansatz in der Praxis sehr erfolgreich. Manche Autoren bezweifeln sogar generell den Nutzen des komplexitätstheoretischen Ansatzes für den

Entwurf praktikabler Kryptosysteme.[3]

Traditionell sucht die **Komplexitätstheorie** meist nach asymptotischen Aussagen. Ihr Ansatz in der Kryptographie beschränkt sich meistens sogar auf die folgende Sichtweise, die wir in dieser Arbeit etwas vereinfachend als *asymptotische Sichtweise* bezeichnen: Ein Kryptosystem ist *effizient*, wenn es in Polynomialzeit berechnet werden kann, und es ist *sicher*, wenn alle Angriffe superpolynomielle Zeit benötigen. Als *Theorie* versucht die Komplexitätstheorie primär einen abstrakten Erkenntnisgewinn zu erreichen. Asymptotische Aussagen sind in der Kryptographie wertvoll, weil sie von vielen Details abstrahieren. Der wichtigste Beitrag, den die Komplexitätstheorie bisher in der Kryptographie geleistet hat, besteht in der genauen Definition und Überprüfung von Konzepten, die ohne die Komplexitätstheorie vage geblieben wären. Aber für den Entwurf praktikabler Kryptosysteme sind asymptotische Aussagen wenig hilfreich. Eine asymptotische Aussage über den Aufwand, ein Kryptosystem zu brechen, liefert keinen Hinweis darauf, wie groß der Sicherheitsparameter bei einer praktischen Realisierung des Kryptosystems gewählt werden muß, um gegebenen Sicherheitsanforderungen zu entsprechen. Glücklicherweise sind die Beweistechniken aus der Komplexitätstheorie oft stark genug, um Aussagen treffen zu können, die genau genug für den Entwurf praktikabler Kryptosysteme sind.

1.3 Quantifizierte Reduktionen

One key technique, however, is [...] the 'reduction'. – D.S. Johnson

Eine Standardmethode bei der Programmierung ist die Verwendung von Unterprogrammen. Das komplexitätstheoretische Gegenstück zum Unterprogramm ist das *Orakel*. Ein Algorithmus A mit einem P-Orakel verhält sich wie jeder andere Algorithmus, nur kann er ein Orakel aufrufen wie ein Unterprogramm, das bei Eingabe einer Instanz des Problems P eine entsprechende Lösung liefert. Bei der Laufzeit von A zählt jeder Orakel-Aufruf als ein Elementarschritt. Wenn A effizient das Problem P_A löst, ist A eine *Reduktion* von P_A auf P, und das Problem P_A ist auf das Problem P *reduzierbar*. Die Reduktion A erlaubt es uns, die Komplexitäten der Probleme P und P_A zu vergleichen: Wenn das Problem P effizient lösbar ist, dann auch das Problem P_A. Umgekehrt gilt genauso: Wenn P_A nicht effizient lösbar ist, dann erst recht nicht P.

Der folgende Satz ist die Grundlage für den Sicherheitsbeweis der public-key Chiffre von Rabin [51]. Für uns liefert dieser Beweis ein erstes Beispiel für eine Reduktion.

Satz 1.1
Sei n das Produkt zweier Primzahlen. Wenn es einen effizienten probabilistischen Algorithmus gibt, der mit signifikanter Wahrscheinlichkeit zu einem zufälligen quadratischen

[3] Besonders deutlich wird Daemen [11]: "*Complexity theory is a powerful tool in an important part of cryptography. However, in the case of practical conventional cryptography, we believe that its relevance is marginal. By practical conventional cryptography we mean symmetric block ciphers, stream ciphers and cryptographic hash functions that actually have to be implemented and used. In our opinion the complexity theoretical way of thinking encourages poor design.*" (Hervorhebung im Original!)

Rest $y(\mathrm{mod} n)$ eine Wurzel x mit $x^2 \equiv y(\mathrm{mod} n)$ findet, dann gibt es einen effizienten Algorithmus, n zu faktorisieren.

Beweis: Wir betrachten den folgenden Algorithmus:

1. Wähle gemäß Gleichverteilung eine zufällige Zahl $r \in \mathbb{Z}_n$.
2. Berechne $y := r^2 \bmod n$.
3. Berechne $t := \mathrm{ggT}(y, n)$.
4. Ist $t > 1$, dann ist t ein nichttrivialer Teiler von n; beende den Algorithmus.
5. Berechne mit Hilfe eines Orakels einen Wert x mit $x^2 \equiv y \pmod n$.
6. Wenn $x \equiv \pm r \pmod n$, dann gib 1 aus. Sonst gib $\mathrm{ggT}(x + r, n)$ aus.

Ist $\mathrm{ggT}(y, n) = 1$, dann hat der quadratische Rest y vier Quadratwurzeln modulo n, unter anderem r und $-r$. Mit der Wahrscheinlichkeit 0,5 ist $x \notin \{r, -r\}$, und in diesem Fall ist $\mathrm{ggT}(x + r, n)$ ein nichttrivialer Teiler von n. □

Im Sinne der asymptotischen Betrachtungsweise ist Satz 1.1 eine durchaus befriedigende Aussage. Wenn wir aber die Rabin public-key Chiffre konkret realisieren wollen und davon ausgehen, daß es praktisch unmöglich ist, eine geeignet gewählte Zahl n zu faktorisieren, müssen wir uns mit der folgenden Frage beschäftigen: *Mit wieviel* Verlust *ist die Reduktion im Beweis von Satz 1.1 behaftet?*

Der Verlust bei einer Reduktion bezieht sich auf die Rechenzeit, die Anzahl der erforderlichen Orakel-Aufrufe und die Erfolgswahrscheinlichkeit. Bei Betrachtung einer Reduktion A von dem Problem P_A auf das Problem P wollen wir das folgende wissen:

(a) Wie komplex ist der Algorithmus A?

(b) Wieviele Aufrufe des P-Orakels sind im Durchschnitt notwendig?

(c) Wenn das P-Orakel bei Eingabe einer Instanz des Problems P mit der Wahrscheinlichkeit p eine entsprechende Lösung liefert, wie groß ist die Wahrscheinlichkeit p_A, daß der Algorithmus A eine Lösung einer Instanz des Problems P_A liefert?

Der Verlust der vorliegenden Reduktion ist wie folgt zu *quantifizieren*:

(a) Algorithmus A erfordert die zufällige Wahl eines Elements aus $\mathbb{Z}_n$, eine Quadratbildung modulo n und die Berechnung eines ggT.

(b) Das Orakel wird genau einmal aufgerufen.

(c) Wenn das Orakel mit der Wahrscheinlichkeit p eine Quadratwurzel $\bmod n$ liefert, dann liefert Algorithmus A mit der Wahrscheinlichkeit $p/2$ einen nichttrivialen Teiler von n.

Wenn wir eine Vermutung haben, wie schwierig es ist, n zu faktorisieren, können wir errechnen, wie effizient die Berechnung von Quadratwurzeln modulo n höchstens sein kann. Stellen wir uns vor, daß mit der gesamten derzeit auf der Welt existierende Rechenkapazität nur mit der Wahrscheinlichkeit 1 % innerhalb eines Jahres ein Teiler von n (außer 1 und n) berechnet werden könnte.[4] Angesichts des Gesamtaufwandes ist der unter (a) fallende Aufwand vernachlässigbar, also kann die Berechnung einer Quadratwurzel modulo n eines zufälligen quadratischen Restes, unter Verwendung derselben Rechenkapazität für die Dauer eines Jahres, höchstens mit der Wahrscheinlichkeit 2 % erfolgreich sein.

Die Reduktion aus dem Beweis von Satz 1.1 ist sehr verlustarm. Nicht immer kann man so verlustarme Reduktionen finden. Im Sinne der asymptotischen Sichtweise ist dies auch nicht nötig. So würde eine Reduktion, bei der $\lceil \log_2(n) \rceil^3$ Orakel-Aufrufe erforderlich wären, für den Beweis von Satz 1.1 ausreichen. Um, gestützt auf eine derartige Reduktion, für n feststellen zu können, daß es praktisch unmöglich ist, Quadratwurzeln modulo n zu berechnen, müßte die Zahl n nicht bloß so gewählt werden, daß ihre Faktorisierung praktisch unmöglich ist. Sondern es müßte der Aufwand für ihre Faktorisierung *weit* jenseits der Schwelle liegen, ab der ein Problem in der Praxis als praktisch unlösbar gilt.

In der vorliegenden Arbeit beruhen Sicherheitsbeweise stets auf quantifizierten Reduktionen. Letztlich muß der Leser bzw. die Leserin selbst entscheiden, welche Probleme er bzw. sie als praktisch unlösbar ansieht. Simultan zu der praktischen Sicht wird in dieser Arbeit auch die asymptotische Sichtweise beachtet.

Wie die schon zitierte Arbeit von Rabin [51] zeigt, ist die Betrachtung quantifizierter Reduktionen nicht neu in der Literatur. Sie wurde aber lange Zeit vernachlässigt und allenfalls informal durchgeführt. Erst in jüngster Zeit wird ihr größere Beachtung zuteil.

Bellare und Rogaway untersuchten die "exakte Sicherheit" von public-key Chiffren [6] und digitalen Unterschriften [7]. Die kryptographischen Hypothesen ihrer Arbeit liefern bestimmte Probleme der Zahlentheorie, die als praktisch unlösbar gelten, u.a. das der Faktorisierung geeignet gewählter ganzer Zahlen. Sie bezeichnen ein Kryptosystem als (t,p)-*sicher*, wenn es keinen Algorithmus gibt, der das Kryptosystem in Zeit t mit einer Wahrscheinlichkeit größer als p bricht. An einigen Stellen dieser Arbeit werden wir uns auf diesen Begriff beziehen, uns sonst aber, wie in diesem Abschnitt, mit einer summarischen Betrachtung des Verlustes begnügen.

Die Reduktion aus dem Beweis von Satz 1.1 erlaubt Aussagen über *beliebige ganzzahlige* n („wenn es praktisch unmöglich ist, nichttriviale Teiler der Zahl n zu finden, ...“). Analog dazu nutzen Bellare, Canetti und Krawczyk [5] quantifizierte Reduktionen für die „konkrete Sicherheitsanalyse“ pseudozufälliger Funktionen *ohne Sicherheitsparameter.*

[4] Die Annahme, daß die Faktorisierung geeignet gewählter Zahlen praktisch unmöglich ist, ist zwar allgemein verbreitet, aber unbewiesen. In der Kryptographie ist es üblich, mit derartigen Vermutungen als *kryptographischen Hypothesen* zu arbeiten. Man kennt bisher kein geeignetes Problem, dessen Lösung beweisbar praktisch unmöglich ist.

1.4 Zielsetzung und Inhalt der vorliegenden Arbeit

Simple analysis is a good thing, if it doesn't weaken the cipher. – C. Plumb

Mit dem systembasierten Ansatz erhält man sehr effiziente Kryptosysteme, kann aber kaum Aussagen über ihre Sicherheit machen. Bestenfalls kann man beweisen, daß *einzelne, bereits bekannte Methoden* der Kryptoanalyse unmöglich sind. Andererseits erhält man mit dem systembasierten Ansatz Kryptosysteme, die oft um mehrere Zehnerpotenzen schneller sind als die besten auf der Basis komplexitätstheoretischer Methoden entworfenen Kryptosysteme [70]. Der vorliegenden Arbeit liegt der Versuch zugrunde, möglichst effiziente Kryptosysteme mit Methoden der Komplexitätstheorie zu entwerfen.

Man beachte, daß man mit komplexitätstheoretischen Methoden nicht nur Anhaltspunkte für die Vermutung findet, daß ein gegebenes Kryptosystem tatsächlich sicher ist. Man erhält auch formale Kriterien, unter welchen Umständen bestimmte Kryptosysteme *nicht* benutzt werden sollten – nämlich dann, wenn die Voraussetzungen für ihre Sicherheitsbeweise nicht erfüllt sind. Der systembasierte Ansatz in der Kryptographie liefert im allgemeinen keine derartigen Kriterien.

In Kapitel 2 werden grundlegende Begriffe der Kryptographie motiviert und definiert. Auch einige schon in der Einleitung informell beschriebene und benutzte Begriffe werden in diesem Kapitel noch einmal und genauer eingeführt. Dazu gehört der Begriff der *praktischen Unlösbarkeit*, im asymptotischen und im praktischen Sinn.

Der Rest der Arbeit gliedert sich in zwei Teile, in denen zwei grundsätzlich verschiedene Ansätze verfolgt werden. In Kapitel 3 gehen wir von einem bestimmten Problem aus, dem *exakten Handlungsreisendenproblem* (XTSP) und entwerfen Kryptosysteme auf der Basis des XTSP. Sicherheitsbeweise für diese Kryptosysteme setzen die *praktische Unlösbarkeit des XTSP* als kryptographische Hypothese voraus. In Kapitel 4 dagegen setzen wir lediglich die *Existenz pseudozufälliger Funktionen* voraus.[5] Es werden Blockchiffren entworfen, die pseudozufällige Funktionen als Bausteine verwenden und deren Sicherheit nur von der Pseudozufälligkeit dieser Bausteine abhängt.

Im Einzelnen gehen wir in Kapitel 3 wie folgt vor: In den Abschnitten 3.1 und 3.2 werden das XTSP untersucht und notwendige Vorarbeiten geleistet. Es folgen Beschreibung und Sicherheitsnachweis eines auf dem XTSP basierenden zero-knowledge Identifikationsprotokolls. Die Abschnitte 3.5 und 3.6 sind der Suche nach effizienten Algorithmen gewidmet, exakte Handlungsreisendenprobleme zu lösen. Ausgehend von den besten dort gefundenen Algorithmen wird eine mutmaßlich sichere Wahl der Sicherheitsparameter vorgeschlagen. In Abschnitt 3.7 wird der Aufwand für einen Identifikationsvorgang untersucht. Es zeigt sich, daß bei einem praktischen Einsatz des Protokolls besonders auf den Kommunikationsaufwand geachtet werden muß. Insbesondere werden spezielle Protokollvarianten behandelt, um den Kommunikationsaufwand eines Identifikationsvorganges zu beschränken. Im Rest des Kapitels, der nur aus zwei Abschnitten besteht, werden eine

[5] Pseudozufällige Funktionen existieren genau dann, wenn es Einwegfunktionen gibt, siehe Abschnitt 2.1. Dies ist ein bekanntes Resultat, das in Kapitel 4 nicht weiter vertieft wird.

kryptographische Hashfunktion und ein pseudozufälliger Bitgenerator definiert, ihre Sicherheit nachgewiesen und ihre Effizienz diskutiert. Die Bedeutung dieses Teils der vorliegenden Arbeit liegt darin, daß hier ein Teilgebiet der Kryptographie bearbeitet wird, auf dem sich der komplexitätstheoretische Ansatz gegenüber dem systembasierten bisher als besonders erfolglos erwiesen hat, bzw. wo der Effizienz-Unterschied zwischen den bisher bekannten besten komplexitätstheoretisch begründeten Kryptosystemen und den besten systembasierten Kryptosystemen besonders groß ist. Außerdem gibt es kaum Ansätze, die Sicherheit scheinbar einfacher kryptographischer Grundoperationen wie Einweg-Hashing oder pseudozufälliger Bit-Erzeugung auf die praktische Unlösbarkeit NP-harter Probleme zurückführen, vgl. den Anfang von Kapitel 3. Zwar erreichen auch die hier dargestellten Kryptosysteme nicht die Effizienz der besten systembasierten Kryptosysteme, doch stellen sie einen Ansatz dar, den Effizienz-Unterschied zu verringern. Wegen der Vorarbeiten aus den Abschnitten 3.1 bis 3.7 fällt dieser Teil des Kapitels sehr kurz aus.

Kapitel 4 ist wie folgt gegliedert: Abschnitt 4.1 behandelt eine Konstruktion von beweisbar sicheren Blockchiffren, sogenannten „Luby-Rackoff Chiffren", aus Pseudozufallsfunktionen. Das zentrale Resultat dieses Kapitels besagt, daß unbalancierte Luby-Rackoff Chiffren sicher sind. Es verallgemeinert einen Satz von Luby und Rackoff [29]. Alle weiteren Ergebnisse sind Verschärfungen des zentralen Resultats, Anwendungen vorangegangener Resultate oder Anwendungen der verwendeten Beweismethode. In den Abschnitten 4.2 und 4.3 werden Verschärfungen des Hauptsatzes gefunden. Diese Verschärfungen erlauben vermehrte Variationsmöglichkeiten bei der Konstruktion von Luby-Rackoff Chiffren – beweisbar ohne Verlust der Sicherheit. Es folgt Abschnitt 4.4, in dem die Auswahl geeigneter Pseudozufallsfunktionen und ihre Auswirkung auf die Effizienz der entsprechenden Blockchiffre behandelt werden. Abschnitt 4.5 nutzt bisherige Resultate für die Kryptoanalyse eines im praktischen Einsatz befindlichen Kryptosystems. Die Sicherheitsbeweise in Kapitel 4 sind durchweg Reduktionen, Abschnitt 4.6 behandelt ihre Quantifizierung. Zur tatsächlichen Realisierung von Luby-Rackoff Chiffren braucht man geeignete Pseudozufallsfunktionen als Bausteine. In den Abschnitten 4.7–4.9 wird die Verwendung systembasiert entworfener Pseudozufallsfunktionen als Bausteine untersucht. Dieser Ansatz, also die Kombination des komplexitätstheoretischen Ansatzes mit dem systembasierten[6], erweist sich als überraschend erfolgreich. Die derart konstruierten Chiffren sind vergleichbar effizient wie die schnellsten rein systembasiert entworfenen Blockchiffren. Es folgen weitere Anwendungen der selben Beweismethodik in den Abschnitten 4.10 und 4.11. Behandelt werden das paradox erscheinende Problem, mit langsamen Smartcards schnell zu verschlüsseln, und Luby-Rackoff Chiffren, die besonders anspruchsvollen Sicherheitskriterien genügen. In Abschnitt 4.12 schließlich wird auf jüngste Forschungsergebnisse auf dem Gebiet der Luby-Rackoff Chiffren hingewiesen.

[6] Anwendungen der Kryptographie sind meist komplexeSysteme und von der sicheren Realisierung mehrerer Komponenten abhängig. In unserem Fall haben wir ein Konstruktionsprinzip, das zu beweisbar sicheren Blockchiffren führt, wenn die verwendeten Bausteine sicher sind. Die Sicherheit der tatsächlich verwendeten Bausteine ist aber eine unbewiesene Vermutung. Werden diese Bausteine ohnehin in der Anwendung verwendet und hängt die Sicherheit der Anwendung von der Sicherheit dieser Bausteine ab, kann man durch die Verwendung einer so konstruierten Blockchiffre die Abhängigkeit von einer *zusätzlichen* unbewiesenen Vermutung vermeiden.

1.5 Veröffentlichungen

[...] nothing substitutes for extensive peer review and years of analysis. – B. Schneier

Teile des in Kapitel 3 enthaltenen Materials sind in den folgenden Artikeln erschienen:

- S. Lucks (1994), *How to Exploit the Intractability of Exact TSP for Cryptography*, Fast Software Encryption, Springer LCS 1008, 298–304.
- S. Lucks (1995), *How Traveling Salespersons Prove their Identity*, Fifth IMA Conference on Cryptography and Coding, Springer LNCS 1025, 142–149.

Teile des in Kapitel 4 enthaltenen Materials sind in den folgenden Artikeln erschienen:

- S. Lucks (1996), *Faster Luby-Rackoff Ciphers*, Fast Software Encryption, Springer LNCS 1039, 189–203.
- S. Lucks (1996), *BEAST: A Fast Block Cipher for Arbitrary Blocksizes*, IFIP Conference on Computers and Communications Security, Chapman & Hall, 144–153.

1.6 Anmerkungen zu Sprache und Notation

Minds and parachutes are alike—they only work when they are open. – K. McCurley

In der vorliegenden Arbeit finden sich zahlreiche englische Fachbegriffe: *Bit, Chosen Ciphertext, Hash, Prover, zero-knowledge, ...* Ihre Verwendung ist auch in deutschen Fachpublikationen üblich. Sprachpuristen werden um Verständnis gebeten.

Protokolle regeln die Kommunikation zwischen zwei oder mehr Kommunikationsteilnehmern. In den Abschnitten 3.3 und 3.4 werden die Kommunikationsteilnehmer durch die virtuellen Charaktere Alice, Bob, Carla und Dan repräsentiert. Alice und Bob halten sich an das Protokoll, Carla und Dan sind Angreifer, die mit Alice und Bob kommunizieren können, ohne an das Protokoll gebunden zu sein. Die Verwendung solcher virtueller Charaktere dient der anschaulichen Darstellung eines Kommunikationsvorganges und ist in der Literatur weit verbreitet.

In der Informatik-Literatur allgemein übliche Schreibweisen und Begriffe werden in dieser Arbeit nicht erklärt. Zwei Schreibweisen, die möglicherweise erklärungsbedürftig sind, sind „$\in_{\mathrm{R}}$“ und „prob[...]“: Es ist $x \in_{\mathrm{R}} M$ ein *zufällig gewähltes Element*[7] der Menge M, und prob$[E|B]$ bezeichnet die *bedingte Wahrscheinlichkeit*, daß das Ereignis E eintritt, wenn die Bedingung B erfüllt ist.

[7] Mit *zufällig gewählt*, ohne weitere Bezeichnung der Wahrscheinlichkeitsverteilung, ist stets *zufällig gemäß Gleichverteilung gewählt* gemeint.

2 Elemente der Kryptographie

Das Hauptziel dieser Arbeit ist es, *effiziente* und *sichere* Kryptosysteme zu entwerfen. Ein Kryptosystem ist „effizient", wenn seine Realisierung mit effizienten Algorithmen möglich ist. Es ist „sicher" gegen einen bestimmten Angriff, wenn das Problem, diesen Angriff durchzuführen, praktisch unlösbar ist. Wir werden uns weiter unten mit verschiedenen Angriffsmöglichkeiten auf Kryptosysteme beschäftigen. Bezüglich Effizienz und praktischer Unlösbarkeit betrachten wir simultan zwei Sichtweisen:

1. Bei der **asymptotischen Sichtweise** betrachtet man Algorithmen und Probleme in Abhängigkeit von einem Parameter n. Dieser wird in der Kryptographie als *Sicherheitsparameter* bezeichnet. Der Ressourcenverbrauch wird asymptotisch (in n) betrachtet. Für uns besonders wichtig ist die Rechenzeit.

   $T(n)$-Zeit-Algorithmen heißen *effizient*, wenn es ein Polynom p gibt mit $T(n) < p(n)$. Ein Problem heißt *praktisch unlösbar*[1], wenn kein effizienter Algorithmus existiert, dieses Problem zu lösen.[2] Wenn nicht anders erwähnt, ist die Bezeichnung für den Sicherheitsparameter stets n, und mit *polynomiell* (ohne Zusatz) ist stets *polynomiell in* n gemeint.

   Offenbar können Probleme nur dann praktisch lösbar sein, wenn die Ausgabegröße und, bis auf Trivialfälle, die Eingabegröße polynomiell sind.

   Grundsätzlich können wir statt Polynomialzeitalgorithmen auch andere vernünftige Berechnungsmodelle verwenden, z. B. Schaltkreisfamilien polynomieller Größe. Eine wichtige Ausnahme davon sind kollisionsresistente Einweg-Hashfunktionen (vgl. Abschnitt 2.3).

2. Spätestens dann, wenn man einen Wert für den Sicherheitsparameter festlegt, muß man zur **praktischen Sichtweise** übergehen. Erst recht gilt dies für kryptographische Grundoperationen ohne Sicherheitsparameter.

[1] Es ist sprachlich unschön, einen theoretischen und formal definierten Begriff mit dem Adjektiv „praktisch" zu versehen – besonders, da es auch eine „praktische Sichtweise" der „praktischen Unlösbarkeit" gibt. Doch ist „praktisch lösbar" die übliche Übersetzung des englischen Begriffs "feasible". Eine Abweichung von der Standard-Bezeichnung scheint hier nicht gerechtfertigt.

[2] Diese Sichtweise ist nur sinnvoll, wenn NP≠P gilt. Doch selbst im Fall NP=P wäre eine verfeinerte asymptotische Sichtweise denkbar, die nicht danach fragt, *ob* es zu einem Algorithmus ein Polynom p gibt, das eine obere Schranke für die Laufzeit liefert, sondern nach dem Grad von p. Um so wichtiger wäre in diesem Fall die Betrachtung quantifizierter Reduktionen.

Ein Algorithmus gilt als um so *effizienter*, je weniger Ressourcen er in Anspruch nimmt, insbesondere je schneller er ist. Ein Problem ist *praktisch unlösbar*, wenn jeder Algorithmus, es zu lösen, mehr Ressourcen in Anspruch nimmt, als ein Angreifer erübrigen kann. Ohne konkretes Wissen um die Angreifer sollte man stets pessimistisch sein und ihre Ressourcen eher überschätzen. Eine übliche Einschätzung ist z.B.: *‚Die Angreifer verfügen für ein Jahr über die gesamte derzeit auf der Welt existierende Rechenkapazität.'*

Als Berechnungsmodell dient uns das der probabilistischen Turingmaschine. Analysen der Laufzeit von Algorithmen liegt das Modell der Registermaschine ("random access machine", kurz RAM) zugrunde (vgl. Abschnitt 1.5.1 in [39]). Eine RAM verfügt über eine unendliche Anzahl von Registern, die ganze Zahlen beliebiger Größe speichern können. Elementaroperationen einer RAM sind lesende und schreibende Registerzugriffe, arithmetische Grundoperationen (also „+", „−", „∗" und „/"), der Vergleich zweier Zahlen und bedingte und unbedingte Sprünge im RAM-Programm. Während die Turingmaschine als abstraktes mathematisches Modell gilt, entsprechen Registermaschinen realen Rechnern – mit einer Ausnahme: Reale Rechner haben nur endlich viele endlich große Register (bzw. Speicherplätze).

Zur Berechnung der Laufzeit einer RAM gibt es zwei Modelle: Das Einheitskostenmaß und das logarithmische Kostenmaß. *Einheitskosten* sind definiert durch die Anzahl der Elementaroperationen bei der Ausführung eines Algorithmus. Für die Berechnung der *logarithmischen Kosten* ist zu den Einheitskosten die Anzahl der bei allen Elementaroperationen mit Operanden (also jeder arithmetischen Grundoperation und jedem Vergleich) zur Darstellung der Operanden erforderlichen Bits zu addieren. *In der vorliegenden Arbeit gilt das logarithmische Kostenmaß.*[3]

In der Praxis ist es situationsabhängig, ob ein Algorithmus bzw. ein Kryptosystem als effizient eingestuft wird oder nicht. Erst recht können ursprünglich praktisch unlösbare Probleme im Laufe der Zeit dank des technischen Fortschritts lösbar werden. Z. B. war bei seiner Einführung in den siebziger Jahren die Schlüssellänge des amerikanischen DES (*"data encryption standard"*) von nur 56 Bits zwar umstritten, allgemein galt es jedoch

[3]Die Einheitskosten können die tatsächlich anfallende Rechenzeit grob unterschreiten – es gibt sogar Algorithmen, die auf einer RAM im Einheitskostenmaß polynomielle Zeit, auf einer Turingmaschine dagegen exponentielle Zeit benötigen. Derart große Unterschiede zum Modell der Turingmaschinen läßt das logarithmische Kostenmaß nicht zu.
Mit Bezug auf reale Rechner ist das logarithmische Kostenmaß allerdings auch problematisch. Reale Rechner haben Register (bzw. Speicherzellen) fester Größe, typischerweise 8–64 Bit. Eine Grundoperation der RAM muß also tatsächlich durch mehrere derart beschränkte Grundoperationen simuliert werden. Logarithmische Kosten können nur dann einen realistischen Eindruck der Rechenzeit vermitteln, wenn die Anzahl der beschränkten Elementaroperationen linear in der Anzahl der Eingabebits der simulierten RAM-Elementaroperation ist. Dies ist für Addition, Subtraktion und Vergleich kein Problem, wohl aber für Multiplikation und Division (vgl. [26, 69]).
Bei einigen der in der vorliegenden Arbeit betrachteten Algorithmen werden große ganze Zahlen addiert bzw. verglichen. Deswegen ist das Einheitskostenmaß für uns nicht sinnvoll. Dagegen treten Punktrechnungen mit großen ganzen Zahlen nicht auf. Dies rechtfertigt die Verwendung des logarithmischen Kostenmaßes.

als nicht realisierbar, den DES mit "brute force"-Methoden zu brechen. Heute ist der Bau geeigneter Hardware realisierbar, mit der im Durchschnitt ein DES-Schlüssel in $3\frac{1}{2}$ Stunden gebrochen werden kann [71].

Weil in der Praxis die Begriffe „Effizienz" und „praktische Unlösbarkeit" nicht formal definierbar, sondern situationsabhängig sind, ist eine theoretisch zufriedenstellende Aussage wie ‚wenn es einen effizienten Algorithmus gibt, das Kryptosystem A zu brechen, dann gibt es einen effizienten Algorithmus, das Problem B zu lösen' zu schwach für den Entwurf praktikabler Kryptosysteme. Mit Hilfe von quantifizierten Reduktionen suchen wir Aussagen der Form: *‚Wenn das Problem B (p', t')-schwierig ist, dann ist das Kryptosystem A (t, p)-sicher'* (in Anlehnung an die in der Einleitung erwähnte Schreibweise von Bellare und Rogaway [6, 7]). Dabei wird angestrebt, daß die Sicherheit des Kryptosystems A und die Härte des Problems B eng miteinander verknüpft sind, also daß weder $t \gg t'$ noch $p \ll p'$ gilt. Das Problem B liefert damit so etwas wie eine *‚Meßlatte'* für das Kryptosystem A.

In Anlehnung an das bisher Gesagte bezeichnen wir eine Wahrscheinlichkeit als *„vernachlässigbar"*, wenn sie für jedes Polynom p schneller als $1/p(n)$ gegen Null geht bzw. (aus praktischer Sicht) wenn sie *‚extrem klein'* ist. Die Wahrscheinlichkeit $P(n)$ heißt *„überwältigend"*, wenn ihr Komplement $1 - P(n)$ vernachlässigbar ist.

Abgesehen davon, daß die praktische Sichtweise auf die „praktische Unlösbarkeit" nicht formal definierbar ist, führt die praktische Sichtweise, anders als die asymptotische, nicht zu einer abgeschlossenen Begriffsbildung. So ist die Eigenschaft, im Sinne der asymptotischen Sichtweise effizient reduzierbar zu sein, transitiv. Mit anderen Worten: Wenn man das Problem A effizient auf das Problem B reduzieren kann und das Problem B effizient auf das Problem C, dann kann man auch das Problem A effizient auf das Problem C reduzieren. Um aber den Verlust der Reduktion von A auf C zu berechnen, muß man die Verluste der beiden einzelnen Reduktionen berechnen oder direkt eine Reduktion vom Problem A auf das Problem C untersuchen.

2.1 Bausteine für Kryptosysteme

In diesem Abschnitt definieren wir pseudozufällige Bit-, Funktions- und Permutationsgeneratoren. Diese Begriffe aus der Komplexitätstheorie werden im Rest der vorliegenden Arbeit für die Konstruktion von Kryptosystemen genutzt.

Definition: Ein *Bitgenerator* ist ein effizienter Algorithmus zur Berechung von Funktionen der Form $f : \{0,1\}^n \longrightarrow \{0,1\}^{l(n)}$ für $l(n) > n$.

In der Kryptographie braucht man insbesondere Bit-Generatoren, deren $l(n)$-Bit Ausgabe praktisch unmöglich von einem $l(n)$-Tupel von Zufallsbits zu unterscheiden ist – wenn die Eingabe $\in_R \{0,1\}^n$ ist. Dies motiviert die folgende Begriffsbildung: Sei A ein beliebiger probabilistischer Algorithmus, der bei Eingabe eines $l(n)$-Bit Strings S in Zeit t den Wert $A(S) \in \{0,1\}$ liefert. Wir bezeichnen mit P_{rand} die bedingte Wahrscheinlichkeit,

daß A bei zufälliger Eingabe eine 1 liefert. Analog definieren wir P_{pseu}, also

$$P_{\text{rand}}(A) = \text{prob}[A(S) = 1 | S \in_{\text{R}} \{0,1\}^{l(n)}] \qquad \text{und}$$
$$P_{\text{pseu}}(A) = \text{prob}[A(S) = 1 | S = f(x), x \in_{\text{R}} \{0,1\}^{n}].$$

Definition: Die *Unterscheidungswahrscheinlichkeit des Bitgenerators f bezüglich A* ist die Differenz $p_{\Delta}^{A}(n) = |P_{\text{rand}} - P_{\text{pseu}}|$. Der Bitgenerator $f : \{0,1\}^{n} \longrightarrow \{0,1\}^{l(n)}$ ist *pseudozufällig*, wenn für alle Polynomialzeitalgorithmen A mit $l(n)$ Eingabe-Bits und einem Ausgabe-Bit und alle Polynome p das Produkt $p_{\Delta}^{A}(n) * p(n)$ gegen Null konvergiert.

Pseudozufällige Bitgeneratoren existieren genau dann, wenn Einweg-Funktionen existieren, siehe [22] und [20]. Die Ausgabe eines pseudozufälligen Bitgenerators ist ein pseudozufälliger *Bit-String*. Die iterierte Anwendung eines pseudozufälligen Bitgenerators ermöglicht die Erzeugung pseudozufälliger Bit-Strings beliebiger (aber polynomieller) Länge aus einem n-Bit Startwert. Der Startwert ist der erste interne Zustand des Generators; bei jeder Iteration werden von den $l(n)$ erzeugten pseudozufälligen Bits $l(n) - n$ ausgegeben, die restlichen n Bits dienen als neuer interner Zustand.

Im Anwendungsfall müssen wir eine *‚sehr große'* Zeitschranke t festlegen und dafür Sorge tragen, daß für jeden t-Zeit-Algorithmus die Unterscheidungswahrscheinlichkeit p_{Δ}^{A} *‚klein genug'* ist. Für festes n und genügend Zeit T gibt es natürlich stets einen T-Zeit-Algorithmus B mit $p_{\Delta}^{B} \geq 1 - 2^{n-l(n)}$.

Definition: Ein *Funktionsgenerator* ist ein effizienter Algorithmus zur Berechnung einer Funktion $G : \{0,1\}^{k(n)} \times \{0,1\}^{n} \longrightarrow \{0,1\}^{m}$, mit polynomiell großem $k(n)$. Man kann G auch als von $K \in \{0,1\}^{k(n)}$ abhängige Familie von Funktionen $\{0,1\}^{n} \longrightarrow \{0,1\}^{m}$ betrachten. Eine *Zufallsfunktion* ist eine Funktion $g : \{0,1\}^{n} \longrightarrow \{0,1\}^{m}$. Jedem der 2^{n} Eingabe-Werte sind m Zufallsbits als Ausgabe zugeordnet. Der Funktionsgenerator $G : \{0,1\}^{k(n)} \times \{0,1\}^{n} \longrightarrow \{0,1\}^{m}$ heißt *pseudozufällig*, wenn für $K \in_{\text{R}} \{0,1\}^{k(n)}$ die Funktion g mit $g(x) = G(K,x)$ praktisch unmöglich von einer Zufallsfunktion zu unterscheiden ist.

Dabei kann man, um über die Zufälligkeit der Funktion $g : \{0,1\}^{n} \longrightarrow \{0,1\}^{m}$ zu entscheiden, keine Beschreibung von g verwenden. Es gibt $2^{2^{n}} 2^{m}$ derartige Funktionen. Zufällig eine Funktion zu wählen, die mit polynomiell vielen Bits beschrieben werden kann, ist überwältigend unwahrscheinlich.

Wir betrachten einen probabilistischen t-Zeit-Algorithmus A mit einem *g-Orakel*. A kann zu jedem Zeitpunkt seiner Abarbeitung dem Orakel eine Eingabe x_i geben und von ihm den Funktionswert $g(x_i)$ verlangen, siehe Abbildung 2.1. Das Orakel verbraucht dabei soviel Rechenzeit, wie A selbst für die Berechnung eines Wertes $G(K, x_i)$ brauchen würde[4]. Die Ausgabe von A ist entweder eine 0 oder eine 1.

Bezeichnung: Analog zu pseudozufälligen Bitgeneratoren kann man *pseudozufällige Funktionsgeneratoren* definieren. P_{rand} ist die bedingte Wahrscheinlichkeit, daß A eine

[4]Manche Autoren, z.B. Lai [27], unterscheiden zwischen der Anzahl der vom Orakel gelieferten Werte ("data-complexity") und dem Aufwand ohne Berücksichtigung der Orakel-Befragungen ("processing-complexity").

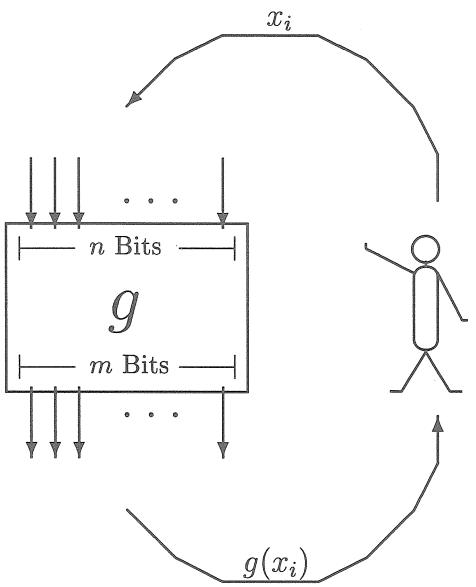

Abbildung 2.1: Befragung eines g-Orakels.

1 liefert, wenn g eine Zufallsfunktion ist. P_{pseu} steht für die Wahrscheinlichkeit, daß A eine 1 liefert bei $K \in_{\text{R}} \{0,1\}^k$ und $g(x) = G(K,x)$. Die *Unterscheidungswahrscheinlichkeit von G bezüglich A* ist $|P_{\text{rand}} - P_{\text{pseu}}|$. Ist $G : \{0,1\}^k \times \{0,1\}^n \longrightarrow \{0,1\}^m$ ein pseudozufälliger Funktionsgenerator und $K \in_{\text{R}} \{0,1\}^k$, dann ist $g(x) = G(K,x)$ eine *pseudozufällige Funktion.*

Bezeichnung: Analog zu dem bisherigen Vorgehen betrachten wir auch Zufallsfunktionen $g : \{0,1\}^* \longrightarrow \{0,1\}^m$ und (pseudozufällige) Funktionsgeneratoren $G : \{0,1\}^k \times \{0,1\}^* \longrightarrow \{0,1\}^m$ *mit beliebiger endlicher Eingabelänge.* „Beliebig" ist aus asymptotischer Sicht natürlich immer noch polynomiell.

Goldreich, Goldwasser und Micali [18] konstruierten pseudozufällige Funktionsgeneratoren $\{0,1\}^{n'} \times \{0,1\}^m \longrightarrow \{0,1\}^n$ aus pseudozufälligen Bitgeneratoren wie folgt: Seien $f' : \{0,1\}^{n'} \longrightarrow \{0,1\}^{2n'}$ und $f : \{0,1\}^{n'} \longrightarrow \{0,1\}^n$ pseudozufällige Bitgeneratoren. Wenn wir einen von beiden gegeben haben, können wir den anderen daraus durch Iterieren oder Abschneiden von Bits konstruieren. Sei $K \in_{\text{R}} \{0,1\}^{n'}$. Die Funktionen f'_0 und f'_1 bezeichnen die linke und die rechte Hälfte der Ausgaben von f', also $f'_0(x) = (y_1, \ldots, y_n)$ und $f'_1(x) = (y_{n+1}, \ldots, y_{2n})$ mit $(y_1, \ldots, y_{2n}) = f'(x)$. Dann ist

$$F_K(x) : \{0,1\}^m \longrightarrow \{0,1\}^n, \quad F_K(x_1, \ldots, x_m) = f(f'_{x_1}(f'_{x_2}(\ldots(f'_{x_m}(K))\ldots)))$$

eine pseudozufällige Funktion, und $F : \{0,1\}^{n'} \times \{0,1\}^m \longrightarrow \{0,1\}^n$ mit $F(K,x) = F_K(x)$ ist ein pseudozufälliger Funktionsgenerator. Wir können uns die Funktion F_K als Binärbaum vorstellen. Bei der Berechung des Wertes $F_K(x_1, \ldots, x_m)$ beginnt man an der Wurzel K, verzweigt – abhängig von $x_m \in \{0,1\}$ – entweder in den Teilbaum mit der Wurzel $f'_0(K)$ oder in den mit der Wurzel $f'_1(K)$, ..., bis schließlich das Blatt $f'_{x_1}(\ldots)$

erreicht ist. Dieses Blatt repräsentiert n' pseudozufällige Bits. Um n pseudozufällige Bits zu erhalten, wenden wir die Funktion f an. In der Praxis kann man die Laufzeit als proportional zu m ansehen – mit einer Proportionalitätskonstanten, die so groß ist, daß einem tatsächlichen Einsatz der Konstruktion enge Grenzen gesetzt sind.

Bezeichnung: Oft betrachtet man statt allgemeiner Zufallsfunktionen $\{0,1\}^n \longrightarrow \{0,1\}^m$ zufällige Permutationen über $\{0,1\}^n$. Die Definitionen für *(pseudozufällige) Permutationsgeneratoren* und *(pseudozufällige) Permutationen* sind analog zu allgemeinen Funktionen.

Man beachte, daß die Umkehrfunktion einer pseudozufälligen Permutation zwar existieren muß, aber nicht notwendigerweise ein effizienter Algorithmus, sie zu berechnen.

Luby und Rackoff [29] beschrieben die Konstruktion pseudozufälliger Permutationsgeneratoren unter Einsatz pseudozufälliger Funktionsgeneratoren. Auf ihren Ansatz wird in Kapitel 4 näher eingegangen.

2.2 Flußchiffren und Blockchiffren

Der Einsatz von Fluß- oder Blockchiffren schützt die *Vertraulichkeit* von Daten. In dieser Arbeit beschränken wir uns auf symmetrische Chiffren, d.h., alle legitimen Nutzer benutzen einen gemeinsamen Schlüssel, der geheim bleiben muß.

Das klassische Modell für symmetrische Chiffren stammt von Shannon [64], siehe Abb. 2.2. In diesem Modell vereinbaren Sender und Empfänger unter Benutzung eines sicheren Kanals einen geheimen Schlüssel K ("key"). Der Sender benutzt K, um den Klartext X mit Hilfe der Verschlüsselungsoperation E ("encrypt") in den Chiffretext Y zu übersetzen. Y wird über einen unsicheren Kanal an den Empfänger übertragen. Der Empfänger benutzt K, um den Chiffretext Y mit Hilfe der Entschlüsselungsoperation D ("decrypt") in den Klartext X zurückzuverwandeln. Potentielle Angreifer kennen Y; alles, was der legitime Empfänger den Angreifern voraus hat, ist die Kenntnis von K. Dies spiegelt Kerkhoffs Prinzip wieder, die Sicherheit von Kryptosystemen nicht von Geheimnissen abhängig zu machen – mit Ausnahme des geheimen Schlüssels.

Kerkhoffs Prinzip:
Die Sicherheit eines Kryptosystems darf nicht von der Geheimhaltung des Ver- und Entschlüsselungsprozesses abhängen, nur von der Geheimhaltung des geheimen Schlüssels.

Kann man, unter Beachtung von Kerkhoffs Prinzip, überhaupt sichere Chiffren finden? Die Antwort lautet „ja", es gibt sogar *beweisbar sichere* Chiffren.

Man kann X, Y und K im Shannon-Modell als Zufallsvariablen auffassen – und, wie Abb. 2.2 nahelegt, X und K als voneinander unabhängige Zufallsvariablen. Ideal wäre es sicherlich, wenn sogar X und Y voneinander unabhängig wären. Dann würden abgefangene Chiffretexte Y keine Rückschlüsse auf X erlauben.

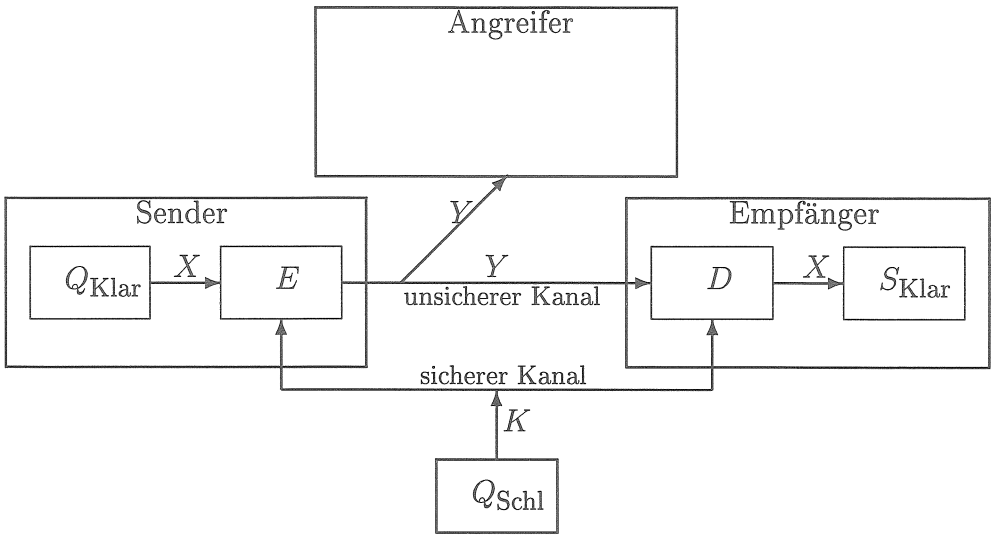

Abbildung 2.2: Shannons Modell für verschlüsselte Kommunikation.
Q_{Klar}: Klartext-Quelle, Q_{Schl}: Schlüssel-Quelle E: Verschlüsseler,
S_{Klar}: Klartext-Senke, D: Entschlüsseler.

Eine Chiffre heißt *perfekt*, wenn die Gesamtheit aller mit dem gleichen Schlüssel verschlüsselten Klartexte statistisch unabhängig von ihren korrespondierenden Chiffretexten ist. Informationstheoretisch betrachtet muß ein Schlüssel für eine perfekte Chiffre damit mindestens soviel Information umfassen, wie alle Klartexte, die mit ihm verschlüsselt werden, zusammen.

Vernam (um 1920) wird die folgende Chiffre zugeschrieben: M ist eine endliche Gruppe mit der Gruppenoperation $\otimes$. Die Schlüssel K_1, K_2, ... $\in M$ sind voneinander unabhängige Zufallswerte, und es gilt die Beziehung $Y_i = X_i \otimes K_i$ zwischen Klartexten $X_i \in M$ und Chiffretexten $Y_i \in M$. 1949 bewies Shannon [64] für die *Vernam-Chiffre*, oft auch als "one-time pad" bezeichnet, daß sie perfekt ist.

Perfektion kann man zwar anstreben, aber fast nie erreichen. Auf die angewandte Kryptographie bezogen bedeutet dies: Die Vernam-Chiffre ist zwar perfekt, aber fast immer unpraktikabel. Die Erzeugung von genug gleichverteilten und voneinander unabhängigen Schlüsselbits kann schwierig sein, erst recht gilt dies für die Übertragung der Schlüsselbits über einen sicheren Kanal sowie deren sichere Speicherung.

O.B.d.A. betrachten wir nur noch den binären Fall $M = \{0, 1\}$ mit der Exklusiv-Oder-Funktion (XOR) als Gruppenoperation. Eine praktikable Alternative zur Vernam-Chiffre ist die Verwendung eines effizienten pseudozufälligen Bitgenerators. Als Schlüssel dient der Startwert des Bitgenerators. Ist es praktisch unmöglich, die Ausgabe des Bitgenerators von Zufallsbits zu unterscheiden, dann ist es ebenso unmöglich, diese Chiffre zu brechen – obwohl sie nicht perfekt ist. Unter anderem deshalb ist die Suche nach effizienten pseudozufälligen Bitgeneratoren ein wichtiges Anliegen in der Kryptographie.

Derartige Chiffren, die für kontinuierliches Ver- bzw. Entschlüsseln einen kontinuierlichen, zufälligen oder pseudozufälligen Fluß von Schlüsselbits brauchen, bezeichnet man als *Flußchiffren*. Flußchiffren sind Transformationen, die vom Schlüssel *und* von einem sich im Lauf der Zeit ändernden internen Zustand des Verschlüsselers abhängen. *Blockchiffren* dagegen operieren mit festen Transformationen von Klartextblöcken in Chiffretextblöcke, abhängig nur vom Schlüssel.

Definition: Eine *Blockchiffre* ist ein Fünf-Tupel (M_K, M_X, M_Y, E, D) mit endlichen Mengen M_K, M_X und M_Y und effizient berechenbaren Funktionen $E : M_K \times M_X \longrightarrow M_Y$ und $D : M_K \times M_Y \longrightarrow M_X$, so daß für alle $K \in M_K$, $X \in M_X$ gilt: $D(K, E(K, X)) = X$.

M_K, M_X und M_Y bezeichnet man als Schlüsselmenge, Klartextmenge und Chiffretextmenge, E als Verschlüsselungsfunktion und D als Entschlüsselungsfunktion. Die Blockgröße der Chiffre ist $\log_2(|M_X|)$.

Wir betrachten in dieser Arbeit keine asymmetrischen ("public-key") Chiffren, bei denen der Schlüssel $K = (K_p, K_s)$ in zwei (Teil-)Schlüssel gespalten ist und die Kenntnis des „öffentlichen" Teils K_p reicht, $E(K, X)$ zu berechnen, während die Berechnung von $D(K, Y)$ ohne Kenntnis von K_s praktisch unmöglich ist.

Soweit nicht anders erwähnt, sind bei den in dieser Arbeit betrachteten Blockchiffren stets Chiffretextmenge und Klartextmenge gleich, also $M_X = M_Y$. Dann ist E eine Permutation und D die Umkehrpermutation von E.

Bei Angriffen auf Blockchiffren werden typischerweise nach und nach Paare der Form $(X_1, E(K, X_1))$, $(X_2, E(K, X_2))$, ..., $(X_q, E(K, X_q))$ bekannt. Das Ziel von Angreifern, die den geheimen Schlüssel K nicht kennen, besteht meist darin, zu weiteren Chiffretextblöcken $E(K, X_{q+1})$, $E(K, X_{q+2})$, ... die passenden Klartexte X_{q+1}, X_{q+2}, ... zu bestimmen. Besonders sicher ist eine Blockchiffre offenbar dann, wenn für $X_{q+1} \notin \{X_1, X_2, \ldots, X_q\}$ der Wert $E(K, X_{q+1})$ unabhängig ist von den Werten $E(K, X_1)$, $E(K, X_2)$, ..., $E(K, X_q)$ – abgesehen von der Beziehung $E(K, X_{q+1}) \neq E(K, X_i)$ für $i \in \{1, \ldots q\}$, die gilt, weil E eine Permutation ist.

Mit anderen Worten, die „sicherste" Blockchiffre – von Shannon [64] als "random cipher" bezeichnet – ist eine Zufallspermutation. Für alle praktisch brauchbaren Blockgrößen ist die "random cipher" ganz und gar unpraktikabel. Deshalb weicht man – analog zu Flußchiffren – auf pseudozufällige Permutationsgeneratoren aus. Diese kann man unter der Einschränkung verwenden, daß nicht nur die pseudozufälligen Permutationen, sondern auch deren Umkehrfunktionen effizient berechenbar sind.

Wir betrachten einen Angriff auf eine Blockchiffre als erfolgreich, wenn die Angreifer mit signifikanter Erfolgswahrscheinlichkeit zwischen der Blockchiffre und einer Zufallspermutation unterscheiden können. Offen bleibt, inwieweit es für Angreifer selbst nach einem erfolgreichen Angriff schwierig ist, für sie nützliche Information zu gewinnen, z.B. Klartexte. Dies hängt von der konkreten Absicht der Angreifer ab und ist einer allgemeinen Untersuchung nicht zugänglich. Wenn die Angreifer die Blockchiffre von einer Zufallspermutation unterscheiden können, haben sie zumindest schon *‚einen Fuß in der Tür'*, und die Chiffre sollte schon deshalb als unsicher angesehen werden.

Für Angriffe auf Blockchiffren gibt es drei klassische Modelle:

(a) *"Ciphertext-only attack":* Angreifer haben keine zusätzlichen Informationen außer den abgefangenen Chiffretexten.

(b) *"Known-plaintext attack":* Angreifer kennen zusätzlich einige Klartexte und die zugehörigen Chiffretexte bezüglich des geheimen Schlüssels K.

(c) *"Chosen-plaintext attack"(CPA):* Angreifer können selbst Klartexte in das System einspeisen und erhalten die zugehörigen Chiffretexte bezüglich des geheimen Schlüssels K.

Anders ausgedrückt: Angreifer können ein Verschlüsselungsorakel befragen, das, wie in Abb. 2.1, eine Verschlüsselungsfunktion g berechnet.

Eine Chiffre gilt üblicherweise als sicher, wenn sie einem CPA widersteht. Insbesondere ist sie dann erst recht sicher gegen die anderen beiden Angriffe.

Im allgemeinen sind Angreifer am Ziel ihrer Wünsche, wenn sie gegebene Chiffretexte entschlüsseln können. Daher ist es scheinbar sinnlos, die mögliche Befragung eines Entschlüsselungsorakels in ein Angriffsmodell einzubeziehen. Doch nicht immer ist die Geheimhaltung bestimmter Chiffretexte der Sinn einer kryptographischen Anwendung, und bei manchen kryptographischen Anwendungen können Angreifer Chiffretexte selbst wählen, in das System einspeisen und entschlüsseln lassen. Für solche Anwendungen muß man auch eines der beiden folgenden Modelle in Betracht ziehen:

(d) *"Chosen-ciphertext attack" (CCA):* Angreifer können ein Entschlüsselungsorakel befragen.

(e) *"Combined chosen-plaintext/chosen-ciphertext attack":* Angreifer können sowohl ein Ver- als auch ein Entschlüsselungsorakel befragen und beliebig zwischen beiden Orakeln wechseln.

Auch diese Angriffe gelten genau dann als erfolgreich, wenn die Angreifer nachweisen können, daß die Blockchiffre wahrscheinlich keine Zufallspermutation ist.

Bei Blockchiffren mit gleich großen Klartext- und Chiffretextmengen verschwindet der Unterschied zwischen CPAs und CCAs – nur muß man daran denken, die Rollen von Ver- und Entschlüsselungsfunktion gegebenenfalls miteinander zu vertauschen. Wir sprechen in diesem Fall von *einseitigen Angriffen.* Entsprechend heißen "combined chosen-plaintext/chosen-ciphertext" Angriffe auch *zweiseitige Angriffe.*

Einseitige Angriffe in unserem Sinne werden in der Literatur oft auch als adaptiv bezeichnet. Z. B. spricht man oft von *"adaptive chosen-plaintext attacks"*, weil die Angreifer die Orakel-Eingabe X_i abhängig von den bisherigen Orakel-Ausgaben $g(X_1)$, ..., $g(X_{i-1})$ wählen können. Ebenso spricht man bei zweiseitigen Angriffen oft von *adaptive chosen-plaintext/chosen-ciphertext attacks.* Da nicht-adaptive Angriffe nur selten betrachtet werden, verzichten wir auf den Zusatz „adaptiv".

2.3 Kryptographisches Hashing

Hashfunktionen sind Funktionen, deren Eingabemenge größer ist als die Ausgabemenge. In dieser Arbeit betrachten wir oft Hashfunktionen, die eine Eingabe beliebiger Länge zu einer Ausgabe fester Länge verarbeiten. Asymptotisch betrachtet und bezogen auf den Sicherheitsparameter n, darf die Eingabe natürlich nur polynomiell groß werden.

Kryptographische Hashfunktionen braucht man, um die Authentizität von Daten zu überprüfen. Derartige Hashfunktionen können von einem geheimen Schlüssel abhängen – in diesem Fall spricht man von einem *"message authentication code" (MAC)* – oder als *Einweg-Hashfunktionen* (1W-HF) ohne Schlüssel benutzbar sein. 1W-HF können außerdem noch *kollisionsresistent* oder *universell* sein – oder keins von beidem.

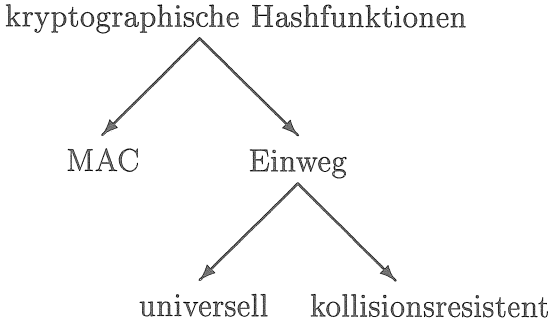

Abbildung 2.3: Einteilung der kryptographischen Hashfunktionen.

Eine vor allem in jüngerer Zeit oft zusätzlich an Hashfunktionen gestellte Forderung ist, *pseudozufällig* zu sein bzw. sich wie eine zufällige Funktion zu verhalten. Anderson [1] demonstriert, wie gefährlich es sein kann, diese Bedingung zu verletzen.

Einweg-Hashfunktionen (1W-HF) werden oft genutzt, um *elektronische Fingerabdrücke* von Daten und Programmen anzulegen. Vergleicht man die ursprünglich genommenen Fingerabdrücke mit den Ist-Werten, kann man Manipulationen an den Daten bzw. Programmen entdecken.

Definition: Die Funktion $h : \{0,1\}^* \longrightarrow \{0,1\}^{l(n)}$ ist eine *Einweg-Hashfunktion*, wenn bei gegebenen m und $X \in_R \{0,1\}^m$ der Wert $h(X)$ effizient berechenbar ist, es hingegen praktisch unmöglich ist, einen Bit-String $Y \in \{0,1\}^*$ mit $h(X) = h(Y)$ zu finden.

Manchmal braucht man eine stärkere Definition. Es könnte bei 1W-HF möglich sein, zwei Eingaben $X, Y \in \{0,1\}^*$ mit $h(X) = h(Y)$ zu fabrizieren. Wenn X z.B. ein Vertrag ist und der Hashwert $h(X)$ digital unterschrieben wurde, dann könnte die Partei, die X formuliert hat, tatsächlich ein Paar (x, y) von Verträgen fabriziert haben mit $h(X) = h(Y)$. Für Dritte wäre nicht zu entscheiden, ob X oder Y der gültige Vertrag ist.

Definition: Die 1W-HF $h : \{0,1\}^* \longrightarrow \{0,1\}^{l(n)}$ ist *kollisionsresistent* (bzw. eine *kollisionsresistente Hashfunktion*), wenn es praktisch unmöglich ist, ein Paar $(X, Y) \in (\{0,1\}^*)^2$ zu finden mit $h(X) = h(Y)$. Ein solches Paar (X, Y) heißt *Kollision*.

Man beachte, daß ein Sicherheitsbegriff, der auf nichtuniformen Berechnungsmodellen wie Schaltkreisfamilien polynomieller Größe beruht, für kollisionsresistente Hashfunktionen sinnlos ist. Egal, wie groß der Sicherheitsparameter n gewählt wird, es gibt unendlich viele Kollisionen. Kennt man eine, ist die Konstruktion eines Schaltkreises zu ihrer Ausgabe trivial. Der Aufwand zur Berechnung eines Schaltkreises, also zur Kollisionssuche, hat keinen Einfluß auf die Schaltkreisgröße.

Während die Definitionen für Einweg- und kollisionsresistente Hashfunktionen (und auch für MACs, s.u.) praktische Erfordernisse widerspiegeln, sind universelle 1W-HF mehr von theoretischem Interesse.

Definition: Sei $n > l(n)$. Die Familie F von Funktionen $f_i : \{0,1\}^n \longrightarrow \{0,1\}^{l(n)}$ ist eine *Familie universeller Einweg-Hashfunktionen*, wenn es für *jedes* $X \in \{0,1\}^n$ und zufällig gewähltes i mit überwältigender Wahrscheinlichkeit praktisch unmöglich ist, ein $Y \in \{0,1\}^n$ zu finden mit $h(X) = h(Y)$.

Hier braucht X nicht zufällig gewählt zu sein, aber die Wahl der Funktion $f_i \in F$ darf nicht von X abhängen. Jede Familie F hängt von dem Sicherheitsparameter n ab; strenggenommen haben wir also eine Familie von Familien von Funktionen.

Wenn man eine Familie universeller 1W-HF $f_i : \{0,1\}^n \longrightarrow \{0,1\}^{l(n)}$ gegeben hat, kann man mit Hilfe des Kompositionslemmas von Naor und Yung [44] eine Familie universeller Hashfunktionen mit beliebig (aber polynomiell) großer Eingabe konstruieren. Das Vorgehen ist analog zur Erzeugung von Bit-Strings beliebiger endlicher Länge. Durch zufällige Wahl eines Familienmitgliedes erhält man mit überwältigender Wahrscheinlichkeit eine 1W-HF. Manche Autoren, etwa Pieprzyk und Sadeghiyan [48], unterscheiden begrifflich nicht einmal zwischen universellen und allgemeinen 1W-HF.

Definition: Ein *Message Authentication Code (MAC)* ist ein Tripel (M_Y, M_K, h) mit endlichen Mengen M_Y und M_K und einer effizient berechenbaren Funktion $h : M_K \times \{0,1\}^* \longrightarrow M_Y$.

MACs werden dazu benutzt, zu einer Botschaft $X \in \{0,1\}^*$ mit Hilfe eines geheimen Schlüssels $K \in M_K$ einen Authentifikationswert $h(K,X) \in M_Y$ zu generieren. Ein Angriff besteht prinzipiell darin, ein neues Paar $(Z,B) \in (M_Y, \{0,1\}^*)$ mit $Z = h(K,B)$ zu erzeugen, ohne den geheimen Schlüssel K zu kennen. Man spricht von einer "existential forgery", wenn für den Angreifer dabei der Inhalt der gefälschten Botschaft B keine Rolle spielt. Kerkhoffs Prinzip folgend, muß man davon ausgehen, daß die Angreifer alles kennen, nur den geheimen Schlüssel K nicht.

In Analogie zu Blockchiffren geht man insbesondere davon aus, daß den Angreifern nach und nach Paare $(h(K,S_1),S_1)$, $(h(K,S_2),S_2)$, ... bekannt werden, und man betrachtet "chosen plaintext attacks" (CPAs), bei denen die Angreifer die Klartexte S_1, S_2, ... selbst wählen. Wir treiben die Analogie zu Blockchiffren noch weiter und betrachten schon die Fähigkeit als erfolgreichen Angriff, für ein Paar $(Y,S) \in M_Y \times \{0,1\}^*$ entscheiden zu können, ob es der Form $(h(K,S),S)$ entspricht und damit authentisch ist. Dabei darf S keiner der zuvor gewählten Klartexte S_1, S_2, ... sein.

Definition: Der MAC $(\{0,1\}^{l(n)}, \{0,1\}^{k(n)}, h)$ heißt *sicher* dann und nur dann, wenn die Funktion $h : \{0,1\}^{k(n)} \times \{0,1\}^* \longrightarrow \{0,1\}^{l(n)}$ pseudozufällig ist.

Aus einer Vielzahl von Gründen wünscht man sich Einweg-Hashfunktionen, die „pseudozufällig" sind. Die Autoren von RIPE-MD [53] schreiben: *"It is the general view that in order to avoid possible statistical attacks, a good cryptographic function should behave like a random function."* Leider erklären sie nicht, was es für eine fest gewählte Funktion bedeutet, „sich wie eine Zufallsfunktion zu verhalten." Fest gewählten Funktionen fehlt ein zufälliger geheimer Schlüssel, im Gegensatz etwa zu MACs. Wir verwenden die folgende Definition für „Zufälligkeit" fest gewählter Funktionen:

Definition: Die Funktion $h : \{0,1\}^{k(n)} \longrightarrow \{0,1\}^{l(n)}$ heißt *pseudozufällig*, wenn die Funktion $f : \{0,1\}^{k(n)} \times \{0,1\}^{k(n)} \longrightarrow \{0,1\}^{l(n)}$ mit $f(K,X) = h(K \oplus X)$ pseudozufällig ist. Dabei bezeichnet $\oplus$ die XOR-Operation. Die Funktion $h : \{0,1\}^* \longrightarrow \{0,1\}^{l(n)}$ heißt *pseudozufällig*, wenn für jedes Polynom k die Restriktion $h_k : \{0,1\}^{k(n)} \longrightarrow \{0,1\}^{l(n)}$ mit $h_k(X) = h(X)$ pseudozufällig ist.

Um die Zufälligkeit fest gewählter Funktionen zu definieren, scheint es natürlich zu sein, wie oben einen Schlüssel K künstlich einzuführen und mit der Eingabe zu verknüpfen, wie wir es getan haben. Neben dem XOR sind andere Gruppenoperationen denkbar; ebenso die Konkatenation von Bit-Strings, die keine Gruppenoperation ist. Unserer Definition haftet demnach etwas Willkürliches an. Immerhin stellt die Verwendung einer Gruppenoperation jedoch sicher, daß für jeden Schlüssel K die Wertemenge der Funktion $f(K,\cdot)$ nicht kleiner als die Wertemenge der Funktion $h(\cdot)$ ist.

Symmetrische Chiffren und MACs hängen von einem geheimen Schlüssel ab; deshalb kann man Angriffe auf sie mit Hilfe eines Orakels modellieren. Bei 1W-HF gilt dies nicht. Man braucht keinerlei geheime Information, um die Hashfunktion ausführen zu können. In Anlehnung an Kerkhoffs Prinzip müssen wir deshalb davon ausgehen, daß Angreifer Hashwerte alleine, ohne Hilfe eines Orakels, berechnen können.

2.4 Identifikationsprotokolle

Identifikationsprotokolle dienen der Authentifikation. Im Gegensatz zu kryptographischen Hashfunktionen werden nicht Daten authentifiziert, sondern Kommunikationsteilnehmer. Das übliche Modell für Identifikationsprotokolle sieht vor, daß ein "Verifier" die Zugangsberechtigung zu einem Kommunikationskanal kontrolliert. Um auf den Kanal zugreifen zu können, muß ein "Prover" ein interaktives Protokoll mit dem Verifier durchlaufen. Der Verifier soll berechtigten Provern den Zugang erlauben, unberechtigte aber von dem Kanal ausschließen.

Definition: Bei einem *interaktiven Protokoll* (P,V) zwischen P (Prover) und V (Verifier) werden zwischen P und V Nachrichten ausgetauscht. P und V sind stets effiziente

Algorithmen.[5] Nach Ablauf des Protokolls kann V entweder akzeptieren oder verwerfen.

Ein *Identifikationsschema* kann man in zwei Phasen untergliedern:

1. In der *Initialisierungsphase* erzeugt jeder Prover ein Schlüsselpaar (PK,SK), bestehend aus einem öffentlichen Schlüssel PK und einem geheimen Schlüssel SK.[6]

2. Die *Operationsphase* läuft als interaktives Protokoll (P, V) zwischen P und V ab. V kennt den öffentlichen Schlüssel PK von P. P weist das Wissen um den geheimen Schlüssel nach. Ein Ablauf der Operationsphase, also ein *Identifikationsvorgang*, ist genau dann *erfolgreich*, wenn V akzeptiert. (Man beachte, daß P die Identität des Verifiers V nicht kennen muß.)

Mit View(P, V, PK) bezeichnen wir die Abfolge der bei einem Protokolldurchlauf ausgetauschten Nachrichten zwischen einem Prover P und einem Verifier V, wobei PK der öffentliche Schlüssel von P ist. Da P und V probabilistische Algorithmen sind, ist View(P, V, PK) eine Zufallsvariable. Ebenfalls eine Zufallsvariable ist $M(\mathrm{PK})$, die Ausgabe des effizienten probabilistischen Algorithmus M bei Eingabe von PK.

Im Rest dieses Abschnittes bezeichnet P stets einen *‚ehrlichen'* Prover, d.h., P kennt den geheimen Schlüssel SK und hält sich an das Protokoll. Ebenso hält sich V stets an das Protokoll. Es bezeichne P^* einen Prover, der Ps geheimen Schlüssel SK nicht kennt, aber versucht, sich als P auszugeben. Es bezeichne V^* einen beliebigen Verifier, der sich nicht notwendigerweise an das Protokoll halten muß.

Definition: Ein Identifikationsschema ist *sicher*, wenn es die folgenden beiden Eigenschaften erfüllt:

1. Identifikationsvorgänge zwischen P und V sind mit überwältigender Wahrscheinlichkeit erfolgreich.

2. Es gibt keine Koalition $\{P^*, V^*\}$, so daß P^* nach Einsicht in beliebig (d.h. polynomiell) viele Mitschriften View(P, V, PK) und View(P, V^*, PK) von Identifikationsvorgängen zwischen P und V bzw. zwischen P und V^* bei einem Identifikationsvorgang (P^*, V) von V mit überwältigender Wahrscheinlichkeit akzeptiert wird.

Die Wahrscheinlichkeit wird dabei über der Verteilung des Schlüsselpaares (PK, SK) von P und der Zufallsentscheidungen bei Ausführung der interaktiven Protokolle der probabilistischen Algorithmen P, V, V^* und V^* berechnet.

In der Praxis wünscht man sich natürlich, daß P^* den Identifikationsvorgang nicht einmal mit signifikanter Wahrscheinlichkeit erfolgreich bestehen kann. Dies kann man z.B. erreichen, indem man den Identifikationsvorgang polynomiell oft wiederholt und den Prover nur dann akzeptiert, wenn alle Teil-Identifikationsvorgänge zuvor erfolgreich waren.

[5] Aus der Bedingung „P und V sind effizient“ können wir eine obere Schranke für den Nachrichtenumfang ableiten. Insbesondere gilt bei der asymptotischen Betrachtungsweise, daß sowohl die Anzahl der Nachrichten als auch die Länge jeder einzelnen Nachricht polynomiell sind.

[6] Teile des öffentlichen Schlüssels können allen Provern gemeinsam sein. Dies ist für die Bestimmung des Speicherplatzbedarfs für den öffentlichen Schlüssel *eines* Provers wichtig.

Ein direkter Sicherheitsnachweis ist für viele Identifikationsprotokolle schwierig. Hier erweist sich der Zero-knowledge-Begriff oft als hilfreich. Neben solchen eher theoretischen Nützlichkeitserwägungen ist der Begriff aber auch von originärer praktischer Bedeutung. Der Verifier V kann nach einem erfolgreich bestandenen Identifikationsvorgang nur eine Mitschrift $\mathrm{View}(P, V, \mathrm{PK})$ der ausgetauschten Nachrichten speichern. Solche Mitschriften kann V aber auch alleine erzeugen. Damit ist kein schlüssiger Nachweis möglich, daß ein bestimmter Identifikationsvorgang überhaupt stattgefunden hat. Die Unmöglichkeit eines solchen Nachweises ist unter Datenschutzgesichtspunkten oft ein Vorteil.

Definition: Das interaktive Protokoll (P, V) ist *zero-knowledge*, wenn es zu jedem Verifier V^* einen effizienten Algorithmus M_{V^*} gibt, den „Simulator", so daß es praktisch unmöglich ist, die durch die Zufallsvariablen $\mathrm{View}(P, V^*, \mathrm{PK})$ und $M_{V^*}(\mathrm{PK})$ induzierten Wahrscheinlichkeitsverteilungen voneinander zu unterscheiden.[7]

Bei dem Entwurf von public-key zero-knowledge Identifikationsprotokollen benutzen wir "Commitments". Wir müssen den sehr technischen Begriff des *Commitments* nicht allgemein definieren. Es genügt vielmehr der folgende Spezialfall: Sei h eine pseudozufällige kollisionsresistente Hashfunktion. P wählt $S \in \{0,1\}^*$ und $R \in_{\mathrm{R}} \{0,1\}^{k(n)}$. Wir bezeichnen S als den *Inhalt des Commitments* und R als *Commitment-Versteck*[8]. P berechnet das Commitment $h(S||R)$, dabei bezeichnet „||" die Konkatenation von Bit-Strings. P *öffnet das Commitment* $h(S||R)$ durch Bekanntgabe von S und R. Aus den Eigenschaften von h folgt:

- Weil h kollisionsresistent ist, kann P keine Bit-Strings S' und R' kennen, mit $S' \neq S$ oder $R' \neq R$. Beim Öffnen des Commitments kann P insbesondere keinen von S abweichenden Inhalt angeben.
- Weil h pseudozufällig ist, kann der Verifier V, bevor das Commitment $h(S||R)$ geöffnet wird, keinerlei Information über dessen Inhalt S gewinnen. Selbst wenn V den Inhalt S vermutet und lediglich R nicht kennt, ist es für V praktisch unmöglich zu verifizieren, ob S tatsächlich der Inhalt von $h(S||R)$ ist. Mit anderen Worten: Der Aufwand für die Verifikation ist exponentiell in $k(n)$.

[7] Unsere Definition des Begriffs "zero-knowledge" wird auch als "computational zero-knowledge" bezeichnet. Eine Abgrenzung zu "perfect zero-knowledge" bzw. "statistical zero-knowledge" ist für den Entwurf praktikabler Kryptosysteme nicht von Bedeutung.

[8] Auf englisch wird R als *"salt"* bezeichnet.

3 Wie man die mutmaßliche Härte des exakten Handelsreisendenproblems nutzt

In diesem Kapitel beschäftigen wir uns mit dem exakten Handelsreisendenproblem und entwerfen ein public-key zero-knowledge Identifikationsprotokoll, eine Einweg-Hashfunktion und einen pseudozufälligen Bitgenerator auf seiner Basis.

Der komplexitätstheoretische Ansatz war besonders in der public-key Kryptographie erfolgreich. Fast immer diente die mutmaßliche praktische Unlösbarkeit zahlentheoretischer Probleme, z.B. des Faktorisierungsproblems, als kryptographische Hypothese. Auf anderen Gebieten der Kryptographie, etwa im Bereich der Einweg-Hashfunktionen und der secret-key Chiffren, gibt es bis heute kein Kryptosystem, das unter einer vernünftigen Annahme beweisbar sicher ist *und* effizient genug für den praktischen Einsatz.

Alle bekannten Kryptosysteme, deren Sicherheit auf der Härte zahlentheoretischer Probleme zurückführbar ist, beruhen auf dem Multiplizieren großer ganzer Zahlen. Die sich daraus ergebende nur mäßige Effizienz ist ein Anreiz, auch andere als zahlentheoretische Probleme zu betrachten. Ein weiterer Anreiz besteht darin, daß es gefährlich ist, alles *‚auf eine Karte'* – wie die Zahlentheorie – zu setzen, solange die praktische Unlösbarkeit der zugrundeliegenden Probleme nur eine Vermutung ist.

Besonders gut untersucht sind die Klassen der NP-harten und NP-vollständigen Probleme. Ihrer Behandlung widmeten sich auch Garey und Johnson, deren schon 1979 geschriebenes Lehrbuch [16] immer noch als Standardwerk gilt. Bis heute ist die Erforschung dieser Klassen und der in ihnen enthaltenen Probleme ein Schwerpunkt in der theoretischen Informatik. Zwar ist ein NP-hartes Problem nicht notwendigerweise auch praktisch unlösbar, selbst wenn NP$\neq$P gilt (s.u.), doch kann die NP-Härte zumindest dazu beitragen, Vertrauen in die praktische Unlösbarkeit eines Problems zu gewinnen. Besonders vertrauensfördernd ist es natürlich, wenn das Problem lange bekannt ist, intensiv erforscht wurde und man keinen praktikablen Ansatz zu seiner Lösung kennt.

Merkle and Hellman [37] entwarfen 1978 eine public-key Chiffre, die auf einem NP-harten Problem basiert. Jedoch konnten sie die Sicherheit ihrer public-key Chiffre nicht auf die praktische Unlösbarkeit des zugrundeliegenden Problems reduzieren, und später wurde die Chiffre gebrochen. In der Tat gilt es als sehr unwahrscheinlich, daß man Angriffe auf eine public-key Chiffre auf ein NP-hartes Problem zurückführen kann.

Erfolgreicher als bei public-key Chiffren war man bei public-key zero-knowledge Identifikationsprotokollen. 1989 veröffentlichte Shamir [63] ein solches Protokoll, das auf dem "Permuted Kernel Problem" (PKP) beruht, einem NP-harten algebraischen Problem.

Mehrere Arbeiten beschäftigten sich mit der Frage nach der praktischen Lösbarkeit des PKP, siehe [47] und die Referenzen dort. Stern schlug 1993 [65] und 1994 [66] Protokolle vor, die auf einem Kodierungsproblem ("Syndrome Decoding") bzw. einem Problem aus der Kombinatorik ("Constrained Linear Equations"') beruhen. 1995 veröffentlichten Pointcheval [49] und Lucks [32] Protokolle, basierend auf dem "Permuted Perceptrons Problem" aus der Lerntheorie und dem graphentheoretischen exakten Handelssreisendenproblem (XTSP). Alle diese Probleme sind NP-hart.

Anders als bei public-key zero-knowledge Identifikationsprotokollen gibt es in der Literatur kaum Ansätze, die Sicherheit von scheinbar einfacheren kryptographischen Grundbausteinen, wie Einweg-Hashfunktionen oder pseudozufälligen Bitgeneratoren, auf die praktische Unlösbarkeit NP-harter Probleme zu reduzieren. Immerhin unternahmen Impagliazzo und Naor [23] sowie Damgård [12] derartige Versuche. 1989 schlugen Impagliazzo und Naor universelle Einweg-Hashfunktionen und pseudozufällige Bitgeneratoren vor, die auf dem NP-harten Subset Sum Problem basieren. Im selben Jahr beschrieb Damgård ganz ähnliche Hashfunktionen, die ebenfalls auf dem Subset Sum Problem basieren und sogar kollisionsresistent sein sollten. Impagliazzo und Naor vermieden einen Vorschlag für die Wahl der Sicherheitsparameter, insbesondere für die Größe der Summandenmenge. Damgårds Vorschlag stellte sich schnell als zu optimistisch heraus. Camion and Patarin [10] zeigten 1991, wie man Damgårds Hashfunktionen brechen kann.[1]

Sogar wenn NP≠P gilt, sind NP-harte Probleme nicht notwendig praktisch unlösbar in unserem Sinne. Die Theorie der NP-Härte bzw. -Vollständigkeit basiert auf worst-case Betrachtungen. Wir suchen aber Probleme, für die man effizient und zufällig Instanzen erzeugen kann, die mit überwältigender Wahrscheinlichkeit praktisch unlösbar sind. Bis heute kennt man keine geeignete Komplexitätsklasse mit vollständigen Problemen. Statt einer allgemeinen Vermutung wie „NP≠P" muß man von einer problemspezifischen Vermutung der Form „eine mit dem effizienten Verfahren Ψ zufällig gewählte Instanz des Problems Π ist mit überwältigender Wahrscheinlichkeit praktisch unlösbar" ausgehen.

Zufallsinstanzen des Subset Sum Problems scheinen tatsächlich leichter zu sein als ursprünglich erwartet. Darauf weisen z.B. empirische Ergebnisse von Schnorr und Euchner [61] sowie Schnorr und Hörner [62] hin. Das Vertrauen der "cryptographic community" in die Härte des Subset Sum Problems ist inzwischen verlorengegangen. Die auf dem XTSP basierenden Einweg-Hashfunktionen und pseudozufälligen Bitgeneratoren, die Lucks [31] 1994 publizierte, könnten die entstandene Lücke füllen (vgl. Abschnitte 3.8 und 3.9). Für pseudozufällige Bitgeneratoren gibt es noch eine 1996 publizierte Konstruktion von Fischer und Stern [15] auf der Basis des "Syndrome Decoding" Problems.

Das Handelsreisendenproblem ("traveling salesperson problem", TSP) ist eines der ältesten und berühmtesten Probleme aus der Theorie der effizienten Algorithmen und der Komplexität. Es galt schon als wissenschaftliche Herausforderung, bevor die Theorie der NP-Härte überhaupt entstand, siehe Hoffman u. Wolfe [21].

[1] Die wesentliche Errungenschaft von Damgårds Artikel [12] ist unbestritten sein Konstruktionsprinzip für kollisionsresistente Einweg-Hashfunktionen. Lediglich Damgårds Beispiele zur Anwendung dieses Konstruktionsprinzips sind unglücklich gewählt.

3.1 Das exakte Handelsreisendenproblem (XTSP)

Gegeben sind die Entfernungen $a_{i,j}$ zwischen n Städten und die Zahl B. Gesucht ist eine Rundreise („*Tour*") der Länge B, d.h. eine Permutation π mit $(\sum_{i=1}^{n-1} a_{\pi(i),\pi(i+1)}) + a_{\pi(n),\pi(1)} = B$. Dieses Problem bezeichnen wir als das *exakte Handelsreisendenproblem* oder XTSP ("exact traveling salesperson problem"). Soweit nicht anders vermerkt, beschränken wir uns auf *asymmetrische* Probleme, oft auch als *gerichtet* bezeichnet, d.h., in der Regel gilt $a_{i,j} \neq a_{j,i}$. Jedoch ist es leicht, die Beweismethoden und Resultate dieser Arbeit auf symmetrische Probleme zu übertragen.

Graphentheoretisch gesprochen, ist das XTSP das Problem, in einem vollständigen gerichteten Graphen G_n mit n Knoten bezüglich der Distanzmatrix $A = (a_{i,j})$ eine Tour X der Länge B zu finden. Für die Länge von X schreiben wir $\text{Length}_A(X)$. Wir setzen stets $a_{i,j} \in \mathbb{Z}_{2^{l(n)}}$ voraus, d.h., alle Distanzen sind mit $l(n)$ Bits darstellbar. Wir betrachten die folgenden Probleme:

1. Das TSP: Gegeben seien eine $n \times n$-Matrix A und eine Zahl B. Finde eine Tour X mit $\text{Length}_A(X) \leq B$.

2. Die Minimierungsvariante des TSP: Gegeben sei eine $n \times n$-Matrix A. Finde eine Tour minimaler Länge.

3. Die ganzzahlige Variante des XTSP: Gegeben seien eine $n \times n$-Matrix A und eine Zahl B. Finde eine Tour X mit $\text{Length}_A(X) = B$.

4. Die modulare Variante des XTSP: Gegeben seien eine $n \times n$-Matrix A und eine Zahl B. Finde eine Tour X mit $\text{Length}_A(X) \equiv B \pmod{2^{l(n)}}$.

Man kann die ganzzahlige Variante des XTSP ihrerseits als Variante des TSP betrachten, bei der man eine Tour einer gegebenen Länge sucht – statt einer besonders kleinen. Der Übergang von der ganzzahligen zur modularen Variante liegt nahe. Wenn die Summanden in $\mathbb{Z}_{2^{l(n)}}$ sind, sollte die Länge auch in $\mathbb{Z}_{2^{l(n)}}$ sein.

Die obigen Definitionen stehen für die *Suchvarianten* der Probleme. Zu dem ersten, dritten und vierten Problem können wir auch eine *Entscheidungsvariante* definieren: Gegeben seien eine $n \times n$-Matrix A und eine Zahl B. Gibt es dann eine Tour mit $\text{Length}_A(X) \leq B$ bzw. $\text{Length}_A(X) = B$ bzw. $\text{Length}_A(X) \equiv B \pmod{2^{l(n)}}$?

Strenggenommen erhalten wir Familien von Problemen, die von $l(n)$ abhängen. Für ein Familienmitglied mit der Zahlenlänge $l(n)$ schreiben wir XTSP-$l(n)$.

Der Standardbeweis [16] für die NP-Härte des TSP ist eine Reduktion auf das Hamiltonkreisproblem in unvollständigen Graphen. Er ist auch für die Minimierungsvariante und für beide Varianten des XTSP gültig. Die NP-Härte des modularen XTSP-$l(n)$ setzt $l(n) > \log_2(n)$ voraus, um Längen in $\{0, \ldots, n\}$ voneinander unterscheiden zu können. Ansonsten reicht $l(n) = 1$.

Wir wollen für zufällig gewählte Distanzmatrizen A die Funktion Length_A als „Einweg-Funktion“ nutzen. Damit kommen beide XTSP-Varianten als Ausgangsprobleme in Frage. Sie lassen sich effizient aufeinander reduzieren. Es genügt, die meist-signifikanten $\lceil \log_2(n) \rceil$ Bits der Längen zu ignorieren bzw. zu ergänzen. Vorteile der modularen Variante sind:

- Die Menge $\mathbb{Z}_{l(n)}$ bildet mit der Addition eine Gruppe.
- Für jede fest gewählte Tour X ist bei zufälliger (d.h. gleichverteilter) Wahl der Distanzmatrix A die Länge $\text{Length}_A(X)$ eine gleichverteilte Zufallsvariable in $\mathbb{Z}_{l(n)}$.

Deshalb beschränken wir uns im folgenden auf die modulare Variante, verzichten auf den Zusatz „modular“ und schreiben $\text{Length}_A(X)$ für die Länge modulo $2^{l(n)}$.

3.2 Eigenschaften des (modularen) XTSP

In diesem Abschnitt definieren und untersuchen wir *zufällige* und *quasi-zufällige* Instanzen des modularen XTSP sowie *balancierte*, *über-* und *unterspezifizierte* Probleme. Dies sind notwendige Vorarbeiten für die Analyse der in dem Rest dieses Kapitels beschriebenen Kryptosysteme.

Hilfssatz 3.2.1 *Seien $X \neq Y$ zwei beliebige Touren. Sei $A \in_R \mathbb{Z}_{2^{l(n)}}^{n \times n}$. Dann ist die Gleichung $Length_A(X) = Length_A(Y)$ mit der Wahrscheinlichkeit $2^{-l(n)}$ erfüllt.*

Beweis: Wegen $X \neq Y$ gibt es mindestens ein Paar (i, j) von Knoten, für die die Kante von i nach j Bestandteil von X, nicht jedoch von Y ist. Da $a_{i,j} \in_R \mathbb{Z}_{2^{l(n)}}$ ist und die anderen Werte der Matrix A nicht von $a_{i,j}$ abhängen, ist die Differenz $\text{Length}_A(X) - \text{Length}_A(Y)$ ihrerseits eine gleichverteilte Zufallsvariable aus $\mathbb{Z}_{2^{l(n)}}$. Gleich Null ist die Differenz mit genau der Wahrscheinlichkeit $2^{-l(n)}$. □

Dem Beweis des Hilfssatzes können wir sogar entnehmen, daß die Längen zweier verschiedener Touren paarweise voneinander unabhängig sind. Jedoch können mehr als 2 Touren sehr wohl voneinander abhängig sein. Wie wahrscheinlich ist eine zufällige Tour eindeutig, und wie wahrscheinlich existiert zu einer zufälligen Länge genau eine Tour, wenn mindestens eine existiert?

Hilfssatz 3.2.2 *Sei X eine Tour. Die Wahrscheinlichkeit, daß es für $A \in_R \mathbb{Z}_{l(n)}^{n \times n}$ eine Tour $Y \neq X$ der Länge $Length_A(Y) = Length_A(X)$ gibt, ist kleiner als*

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

Beweis: Die Wahrscheinlichkeit, daß mindestens eines von mehreren möglichen Ereignissen eintritt, kann nicht größer sein als die Summe der Einzelwahrscheinlichkeiten dieser

Ereignisse. Jede von $(n-1)!-1$ Touren $Y_i \neq X$ hat mit der Wahrscheinlichkeit $2^{-l(n)}$ die Länge $\text{Length}_A(X)$, siehe Hilfssatz 3.2.1. Aufsummieren dieser Einzelwahrscheinlichkeiten ergibt den Wert $2^{-l(n)}((n-1)!-1) < 2^{-l(n)}(n-1)!$. □

Lemma 3.1
Seien $B \in_R \mathbb{Z}_{2^{l(n)}}$ und $A \in_R \mathbb{Z}_{2^{l(n)}}^{n \times n}$. Wenn es eine Tour der Länge B bezüglich A gibt, ist sie mit einer Wahrscheinlichkeit über $1 - \frac{(n-1)!}{2^{l(n)}}$ eindeutig.

Das Lemma folgt sofort aus Hilfssatz 3.2.2. Wie wahrscheinlich gibt es zu einer zufälligen Zahl $B \in \mathbb{Z}_{l(n)}$ überhaupt eine Tour der Länge B?

Lemma 3.2
Sei $2^{l(n)} < 2 * (n-1)!$, $B \in \mathbb{Z}_{2^{l(n)}}$ und $A \in_R \mathbb{Z}_{2^{l(n)}}^{n \times n}$. Mit einer Wahrscheinlichkeit über 25% kann man B bezüglich A als Länge einer Tour darstellen.

Beweis: In diesem Beweis setzen wir $m = (n-1)!$ und bezeichnen eine Menge $\{X, Y\}$ von zwei Touren $X \neq Y$ als *Treffer*, wenn beide die gleiche Länge in $\mathbb{Z}_{2^{l(n)}}$ haben.

Die Wahrscheinlichkeit des Ereignisses „bezüglich A gibt es eine Tour der Länge B" hängt nicht von B ab, wenn B nicht von A abhängt.

Es genügt, die Behauptung für das größtmögliche $l(n)$ nachzuweisen, d.h. für $m \leq 2^{l(n)} < 2m$. Es gibt $m(m-1)/2$ zwei-elementige Mengen von Touren. Hilfssatz 3.2.1 zufolge ist für jede derartige Menge die Treffer-Wahrscheinlichkeit gleich $2^{-l(n)}$. Damit ist der Erwartungswert für die Anzahl der Treffer gleich

$$\frac{m(m-1)}{2^{l(n)}2}.$$

Maximal können m verschiedene Tourenlängen auftreten. Die Anzahl der verschiedenen Längen sinkt pro Treffer höchstens um 1. Der Erwartungswert E für die Anzahl der auftretenden Tourenlängen ist damit $E \geq m - m(m-1)/(2^{l(n)}2)$. Es gilt $m-1 < m \leq 2^{l(n)}$ und damit $m(m-1)/2^{l(n)+1} < m/2$, also

$$E \geq m - \frac{m(m-1)}{2^{l(n)}2} > m - \frac{m}{2} > \frac{m}{2}.$$

Es ist $2^{l(n)} < 2m$, und es gibt nur $2^{l(n)}$ Zahlen in $\mathbb{Z}_{2^{l(n)}}$. Also sind im Durchschnitt mehr als 25% davon bezüglich $A \in_R \mathbb{Z}_{2^{l(n)}}^{n \times n}$ als Länge einer Tour darstellbar. □

Wie sollte man $l(n)$ wählen, damit zufällige Instanzen eines XTSP-$l(n)$ besonders schwierig sind? Als *zufällige Instanz eines XTSP-$l(n)$* bezeichnen wir ein Paar $(B, A) \in_R \mathbb{Z}_{2^{l(n)}} \times \mathbb{Z}_{2^{l(n)}}^{n \times n}$. Gesucht ist eine Tour X mit $\text{Length}_A(X) = B$. Als *lösbar* bezeichnen wir (B, A), wenn es ein solches X gibt. Schon die Entscheidung, ob eine Instanz lösbar ist, ist NP-vollständig (vgl. [16]).

Es gibt $(n-1)!$ Touren und $2^{l(n)}$ mögliche Längen. XTSP-$l(n)$ mit $l(n) < \log_2((n-1)!)$ bezeichnen wir als *unterspezifiziert*, XTSP-$l(n)$ mit $l(n) > \log_2((n-1)!)$ als *überspezifiziert* und XTSP-$l(n)$ mit $l(n) \approx \log_2((n-1)!)$ als *(etwa) balanciert.*

Die empirische Erfahrung mit Heuristiken für ein ähnlich gelagertes Problem, dem Subset Sum Problem, weist auf das folgende hin (siehe etwa Schnorr und Euchner [61] und Schnorr und Hörner [62]): Gleichgewichtige Probleme sind besonders schwer, stark über- oder unterspezifizierte Probleme können praktisch einfach sein. In der Tat zeigt der folgende Satz, daß balancierte Varianten des XTSP nicht zu nennenswert leichteren Problemen führen als über- oder unterspezifizierte.

Satz 3.3
Sei $l'(n) \leq l(n) < \log_2((n-1)!) + 1$ *oder* $l'(n) \geq l(n) > \log_2((n-1)!) + 1$. *Wenn es einen effizienten probabilistischen Algorithmus gibt, der mit der Wahrscheinlichkeit* $p > 0$ *Lösungen für lösbare Zufallsinstanzen des XTSP-*$l(n)$ *findet, dann gibt es einen ebensolchen Algorithmus für das XTSP-*$l'(n)$*, dessen Erfolgswahrscheinlichkeit* $p/4$ *übersteigt.*

Beweis: Sei $(B', A') \in_R \mathbb{Z}_{2^{l'(n)}} \times \mathbb{Z}_{2^{l'(n)}}^{n \times n}$, eine zufällige Instanz des XTSP-$l'(n)$, gegeben.

Für $l'(n) \leq l(n) < \log_2((n-1)!) + 1$ ergänzen wir die Länge B' und die Koeffizienten $a'_{i,j}$ aus A' um jeweils $l(n) - l'(n)$ Zufallsbits und erhalten eine Zufallsinstanz (B, A) des XTSP-$l(n)$ mit $B' = B \pmod{2^{l'(n)}}$, $A = (a_{i,j})$ und $a'_{i,j} = a_{i,j} \bmod 2^{l'(n)}$. Aus Lemma 3.2 folgt, daß (B, A) mit mehr als 25% Wahrscheinlichkeit lösbar ist. Ist (B, A) lösbar, kann man nach Voraussetzung mit der Wahrscheinlichkeit p eine Lösung finden. Jede Lösung für (B, A) ist zugleich eine Lösung für (B', A').

Für $l'(n) \geq l(n) > \log_2((n-1)!) + 1$ berechnen wir (B, A) mit $B = B' \pmod{2^{l(n)}}$, $A = (a_{i,j})$ und $a_{i,j} = a'_{i,j} \bmod 2^{l(n)}$. Da (B', A') eine Zufallsinstanz des XTSP-$l'(n)$ ist, ist (B, A) eine solche des XTSP-$l(n)$. Lemma 3.1 zufolge ist (B, A) mit mehr als der Wahrscheinlichkeit $1 - \frac{2^{l(n)}}{2*(n-1)!}$, also mehr als 50%, eindeutig lösbar – wenn überhaupt. Ist X die einzige Lösung für (B, A), ist X erst recht die einzige Lösung für (B', A') – wenn (B', A') lösbar ist. □

Im folgenden wenden wir die vermutete Härte des XTSP an, um Kryptosysteme zu definieren und ihre Sicherheit auf die Schwierigkeit, gewisse XTSP-Instanzen zu brechen, zurückzuführen. Wir setzen die Existenz einer Distanzmatrix A voraus. Sie sollte *'zufällig'* gewählt sein, z.B. anhand der Binärdarstellung der Zahl π, und darf öffentlich bekannt sein – letzteres in Analogie beispielsweise zu den S-Boxen des amerikanischen DES. Es treten Instanzen (B', A) des XTSP auf, die praktisch unmöglich zu lösen sein sollen. Dabei kann es vorkommen, daß B' nicht zufällig gewählt wurde, sondern sich als Länge einer Zufallstour errechnet. Dann ist die Instanz (B', A) keine Zufallsinstanz im Sinne dieses Abschnitts – wir sprechen von einer *quasi-zufälligen Instanz.* Der folgende Satz zeigt, daß quasi-zufällige Instanzen nicht viel leichter sein können als zufällige.

Satz 3.4
Wenn ein effizienter probabilistischer Algorithmus existiert, der, gegeben eine quasi-zufällige Instanz des XTSP-$l(n)$, mit der Wahrscheinlichkeit p eine Lösung findet, dann gibt es einen ebensolchen Algorithmus für lösbare Zufallsinstanzen (B, A), dessen Erfolgswahrscheinlichkeit $p(n-1)/n^2$ übersteigt.

Beweis: Sei (B, A) gegeben. Wir betrachten den folgenden Algorithmus:

1. Wähle eine Zufallstour X und berechne $B' = \text{Length}_A(X)$. Ist $B = B'$, terminiere.
2. Wähle eine Zufallskante $v \to w$ auf X und berechne $A' = (a'_{i,j})$ mit $a'_{i,j} = a_{i,j}$ für $(i \neq v$ oder $j \neq w)$ und $a'_{v,w} = a_{v,w} - B' + B$. Dann gilt $\text{Length}_{A'}(X) = B$.
3. Versuche, eine Lösung Y für die quasi-zufällige Instanz (B, A') zu finden.
4. Wenn die Kante $v \to w$ auf Y liegt, gilt $\text{Length}_A(Y) = B'$, sonst gilt $\text{Length}_A(Y) = B$, und Y ist eine Lösung für (B, A).

Da die Kante $v \to w$ zufällig gewählt wurde, ist eine Lösung für die Instanz (B, A) mit der Wahrscheinlichkeit $(n-1)/n$ auch eine Lösung für die Instanz (B, A'). Insbesondere gibt es mit mindestens dieser Wahrscheinlichkeit mehr als eine Lösung für (B, A'). Die Lösung X ist zufällig gewählt. Wenn es daher mehr als eine Lösung für (B, A') gibt und im dritten Schritt eine Lösung Y gefunden wird, gilt mit mindestens der Wahrscheinlichkeit $1/2$ die Ungleichheit $X \neq Y$. Gilt sie, gibt es mindestens zwei Kanten auf X, die nicht auf Y liegen, insbesondere liegt die zufällige Kante $v \to w$ mit mehr als der Wahrscheinlichkeit $2/n$ nicht auf Y. Wenn im ersten Schritt noch keine Lösung für (B, A) gefunden wird und wenn im dritten Schritt eine Lösung Y für (B', A) gefunden wird, ist Y mit mehr als der Wahrscheinlichkeit $(n-1)/n^2$ eine Lösung für (B, A).

Nach Voraussetzung kann man mit der Wahrscheinlichkeit p im dritten Schritt eine Lösung finden, wenn die Eingaben quasi-zufällig sind. Ist die bedingte Wahrscheinlichkeit, eine Lösung zu finden, wenn es mehr als eine Lösung gibt, nicht kleiner als p, sind wir fertig. Andernfalls muß die bedingte Wahrscheinlichkeit, eine Lösung zu finden, wenn diese eindeutig ist, größer als p sein. Die Wahrscheinlichkeit, eine eindeutig lösbare Instanz zu haben, ist für lösbare Zufallsinstanzen größer als für quasi-zufällige Instanzen. Deshalb wendet man in diesem Fall den Algorithmus aus Schritt 3 auf (B, A) an. □

Bei dem Beweis von Satz 3.3 lösen wir XTSP-$l'(n)$-Instanzen unter Einsatz von XTSP-$l(n)$-Orakeln. Für jede XTSP-$l'(n)$-Instanz erfolgt genau ein Orakel-Aufruf. Der Rechenaufwand zusätzlich zu den Orakel-Aufrufen ist gering. Von der Erfolgswahrscheinlichkeit p bleibt mehr als $p/4$ übrig. Der Verfasser dieser Arbeit sieht darin eine sehr verlustarme Reduktion. Auch die Reduktion aus dem Beweis von Satz 3.4 sieht er als verlustarm an. Zwar kann die Erfolgswahrscheinlichkeit nicht um einen nur konstanten, sondern um einen zu n proportionalen Faktor sinken, doch im Kontext dieser Arbeit wird n nur moderat groß (siehe Abschnitt 3.6 bzw. 3.9). Man beachte, daß die Größe einer Instanz des modularen XTSP $l(n)(n^2 - n + 1)$ ist.

Bei symmetrischen XTSPs lassen sich mit den gleichen Argumenten Aussagen analog zu den Sätzen 3.3 und 3.4 gewinnen. Nur die Anzahl der Touren halbiert sich.

3.3 Wie Handelsreisende sich identifizieren

Im folgenden stellen wir ein public-key zero-knowledge Identifikationsprotokoll auf der Basis des XTSP dar. Wir können den Parameter $l(n)$ beliebig wählen – im Licht von Satz 3.3 setzen wir $l(n) = \lceil \log_2((n-1)!) \rceil$ voraus. Sei n so gewählt, daß zufällige XTSP praktisch unlösbar sind. Es dient $k(n)$ als zusätzlicher Parameter für die Sicherheit des Commitment-Schemas (vgl. Abschnitt 2.4). Der Aufwand, einen richtig erratenen Commitment-Inhalt zu verifizieren, ist exponentiell in $k(n)$. Ist z.B. $k(n) = 80$, kann man gegenwärtig einen solchen Verifikations-Angriff als praktisch unmöglich erachten.

Weiter nehmen wir die Existenz einer pseudozufälligen kollisionsresistenten Hashfunktion $H : \{0,1\}^* \longrightarrow \{0,1\}^l$ an. Wäre H eine pseudozufällige Einweg-Hashfunktion, aber nicht kollisionsresistent, so wäre unser Identifikationsprotokoll unsicher gegen die von Girault und Stern [17] beschriebenen Angriffe.

Permutationen π über $\{1,\ldots,n\}$ wenden wir auch auf $(n \times n)$-Matrizen an, im Fall $A = (a_{i,j})$ schreiben wir $\pi(A) = (a_{\pi(i),\pi(j)})$. Ebenso wenden wir π auch auf Touren an, insbesondere gilt

$$\mathrm{Length}_{\pi(A)}(\pi(X)) = \mathrm{Length}_A(X)$$

für jede Tour X und jede Distanzmatrix A.

Die Summe $A + A' = (a''_{i,j})$ zweier Distanzmatrizen $A = (a_{i,j})$ und $A' = (a'_{i,j})$ berechnen wir komponentenweise, d.h. $a''_{i,j} = a_{i,j} + a'_{i,j} \bmod 2^{l(n)}$.

Der öffentliche Schlüssel besteht aus der Matrix A und der Länge B, wobei die Matrix A von vielen Provern gemeinsam verwendet werden kann. Der geheime Schlüssel ist eine Tour X mit $\mathrm{Length}_A(X) = B$.

Wir benutzen das Komma „," in Hashfunktionen für die Konkatenation von Bit-Strings.

Das Protokoll läuft zwischen "Prover" Alice und "Verifier" Bob ab wie folgt:

1. Alice wählt eine Zufallsmatrix A^* und eine zufällige Permutation π über $\{1,\ldots,n\}$. Sie wählt die Commitment-Verstecke $r_0, r_1, r_2 \in_{\mathrm{R}} \{0,1\}^{k(n)}$, berechnet die Werte
   $B^* = \mathrm{Length}_{A^*}(X')$, $h_0 = H(A^*, r_0)$,
   $X' = \pi(X)$, $h_1 = H(X', B^*, r_1)$,
   $A' = \pi(A)$ und $h_2 = H(A^* + A', r_2)$
   und übermittelt die Commitments h_0, h_1 und h_2 an Bob.

2. Bob wählt eine zufällige Herausforderung $c \in \{0,1,2\}$ und sendet diese an Alice.

3. Im Fall $c = 0$ antwortet Alice mit r_0, r_1, A^* und X'.
   Im Fall $c = 1$ antwortet Alice mit r_1, r_2, $A^* + A'$ und X'.
   Im Fall $c = 2$ antwortet Alice mit r_0, r_2, A^* und π.

4. Für Bob besteht Alices Antwort aus zwei Commitment-Verstecken r_i und r_j und einem Paar (M, N) bzw. (M, σ), wobei M eine Matrix ist, N eine Tour und σ eine Permutation über $\{1\ldots., n\}$.
   Im Fall $c = 0$ testet Bob $H(M, r_i) = h_0$ und $H(N, \text{Length}_M(N), r_j) = h_1$.
   Im Fall $c = 1$ testet Bob $H(M, r_j) = h_2$ und $H(N, \text{Length}_M(N) - B, r_i) = h_1$.
   Im Fall $c = 2$ testet Bob $H(M, r_i) = h_0$ und $H(M + \sigma(A), r_j) = h_2$.

Bob akzeptiert, wenn die getesteten Bedingungen erfüllt sind.

3.4 Eigenschaften des Protokolls

Für den Nachweis der Sicherheit unseres Identifikationsprotokolls verwenden wir den folgenden Hilfssatz.

Hilfssatz 3.4.1 *Das Identifikationsprotokoll aus Abschnitt 3.3 hat die folgenden Eigenschaften:*

1. *Es kann mit effizienten probabilistischen Algorithmen realisiert werden.*
2. *Wenn Alice und Bob sich an das Protokoll halten, verläuft das Protokoll erfolgreich.*
3. *Wenn Bob die Herausforderung c zufällig gemäß Gleichverteilung wählt, sind bei Provern, die X nicht kennen und c erst im zweiten Schritt erfahren, die Bedingungen der Tests im letzten Schritt höchstens mit der Wahrscheinlichkeit 2/3 erfüllt.*

Beweis: Die ersten beiden Eigenschaften sind offensichtlich.

Zum Nachweis der dritten Eigenschaft stellen wir uns vor, daß Carla, die X nicht kennt, sich Bob gegenüber als Alice auszugeben versucht.

Dank der Kollisionsresistenz von H muß Carla sich schon im ersten Schritt auf die Matrizen A^*, $A^* + A'$, die Tour X' und die Länge B^* festlegen. Sind die Commitments h_0, h_1 und h_2 erst einmal gemacht, liegt ihr Inhalt fest.

Um im Fall $c = 2$ die richtige Antwort geben zu können, muß Carla eine Permutation π mit $\pi(A) = A' = (A^* + A') - A^*$ kennen[2]. Wenn $\text{Length}_{A^*}(X') \neq B^*$ ist, kann Carla im Fall $c = 0$ keine Antwort geben, die die Bedingungen erfüllt. Die Bedingungen im Fall $c = 1$ erzwingen schließlich

$$\text{Length}_{A'+A^*}(X') = \text{Length}_{A'}(X') + \text{Length}_{A^*}(X') \bmod 2^{l(n)} = B + B^* \bmod 2^{l(n)}.$$

Zwar gilt nicht notwendigerweise $X' = \pi(X)$, wohl aber

$$\text{Length}_{A'}(X') = B = \text{Length}_A(\pi^{-1}(X')).$$

[2]Wir setzen voraus, daß alle $a_{i,j}$ mit $i \neq j$ paarweise verschieden sind. Für sinnvoll gewählte Parameter n und $l(n)$ und zufällige $a_{i,j}$ gilt dies ohnehin mit überwältigender Wahrscheinlichkeit. Dann ist π eindeutig durch $\pi(A) = A'$ bestimmt.

Wenn also Carla in der Lage ist, Bob in jedem der drei möglichen Fälle von ihrer Identität als Alice zu überzeugen, dann kann sie entweder eine Lösung des zugrundeliegenden XTSP berechnen oder Kollisionen für die Hashfunktion H finden. Kann sie Bob in einem der drei Fälle nicht überzeugen, bleibt ihr nur eine Wahrscheinlichkeit von $2/3$, Bob erfolgreich zu betrügen. □

Man beachte, daß aus Hilfssatz 3.4.1 nicht unmittelbar folgt, daß das Protokoll sicher ist. Der Hilfssatz würde z.B. auch gelten, wenn Alice einfach ihren geheimen Schlüssel X an den Verifier übermitteln würde. In diesem Fall könnte z.B. der Verifier Dan die Tour X erfahren, X an Carla weitergeben und Carla sich als Alice ausgeben. Was kann ein beliebiger Verifier bei unserem Protokoll über den geheimen Schlüssel erfahren?

Satz 3.5
Das Identifikationsprotokoll aus Abschnitt 3.3 ist zero-knowledge.

Beweis: Bei einem Protokolldurchlauf gibt Alice nur eine Zufallsmatrix M und entweder eine zufällige Tour N oder eine Zufallspermutation σ preis. Verifier kennen dann die Inhalte von zwei der drei Commitments h_0, h_1 und h_2. Der Inhalt des verbleibenden Commitments bleibt ihnen unbekannt.

Zufällige Matrizen, zufällige Touren und zufällige Permutationen kann ein Simulator SIM alleine erzeugen. SIM wählt zuerst c, erzeugt eine passende Antwort (M, N) bzw. (M, σ) und berechnet die beiden entsprechenden Commitments. Statt des dritten Commitments, das ungeöffnet bleibt, wählt der SIM einen Zufallswert. Da die für die Commitments verwendete Hashfunktion pseudozufällig ist, ist der Zufallswert praktisch unmöglich von einem *'echten'* Commitment zu unterscheiden. □

Dies zeigt, daß Carla von Dan nicht mehr über X erfahren kann als durch Einsatz des Simulators SIM aus eigener Kraft. Aus Hilfssatz 3.4.1 folgt damit

Satz 3.6
Das Identifikationsprotokoll aus Abschnitt 3.3 ist sicher.

3.5 Heuristische Algorithmen für das XTSP

Um das Identifikationsprotokoll zu implementieren, muß man den Parameter n festlegen. Mit Hilfe einer vollständigen Durchsuchung aller $(n-1)!$ Touren kann man jedes XTSP lösen. Gibt es auch effizientere Algorithmen?

Für die Minimierungsvariante des TSP kennt man gute Heuristiken, namentlich "branch–and–bound" und davon abgeleitete Verfahren wie "branch–and–cut". Padberg und Rinaldi [46] lösten mit Hilfe von branch–and–cut Euklidische bzw. geographische TSPs[3] mit mehreren tausend Städten.

[3] Ein Handelsreisendenproblem (TSP) heißt „Euklidisch", wenn die Städte so als Punkte in der Ebene darstellbar sind, daß die Distanz zweier Städte deren Euklidische Entfernung ist. Geht man von der Ebene zu einer Kugeloberfläche über, erhält man „geographische" TSPs. Euklidische und geographische TSPs sind symmetrisch, und ihre Entfernungen gehorchen der Dreiecksungleichung.

Grob kann man "branch–and–bound" wie folgt beschreiben: Man startet mit der Menge aller möglichen Lösungen als Lösungsraum. Enthält der Lösungsraum S nur eine Lösung, untersucht man sie. Sonst unterteilt man S in kleinere Teilräume $S_1, S_2, \ldots, S_k$, berechnet zu jedem Teilraum S_i eine obere und/oder eine untere Schranke für alle Lösungen in S_i. Anhand der Schranken wird getestet, ob S_i die gesuchte Lösung überhaupt enthalten kann. Ist dies der Fall, wendet man "branch–and–bound" rekursiv auf den Lösungsraum S_i an. Sonst braucht S_i nicht weiter betrachtet zu werden.

Dies entspricht dem Durchlaufen eines Lösungsbaumes. Wenn der Knoten S kein Blatt ist, zweigen von S die Teilbäume $S_1, S_2, \ldots, S_k$ ab. Mit Hilfe der oberen und unteren Schranken für die Lösungen in S_i versucht man, den Lösungsbaum zu beschneiden.

Die Erfolge von "branch–and–..." bei der Minimierungsvariante des TSP zeigen, daß es möglich ist, *'nahe der Wurzel' 'viele Teilbäume'* abzuspalten, für die man zeigen kann, daß jede in ihnen liegende Lösung mit Sicherheit *'zu groß'* ist. Natürlich ist bei der Minimierungsvariante des TSP zu erwarten, daß fast alle Touren viel größer als die gesuchte minimale Tour sind. In unserem Fall wird aber eine zufällige Tour gesucht, deren Länge fast nie besonders groß oder besonders klein ist. Dies erschwert es, *'nahe der Wurzel'* Teilbäume zu finden, in denen alle Touren größer oder alle Touren kleiner als die gesuchte sind – schlechte Voraussetzungen für "branch–and–..."!

Empirische Erfahrung bestätigt diese Überlegungen. Thienel [68] schrieb, basierend auf dem Code von Jünger, Reinelt und Thienel [25], einen TSP-Löser für TSPs unter Nebenbedingungen. Die Bedingung, daß die gesuchte Lösung eine Tour einer fest vorgegebenen Länge sein soll, kann man als solche Nebenbedingung betrachten. Thienel experimentierte mit symmetrischen ganzzahligen XTSPs. Er benutzte eine Sun Sparcstation 10 mit 60 Mhz Taktfrequenz und Speicherplatz für etwa 30 000 "branch–and–bound"-Knoten.

Die folgenden Distanzmatrizen wurden betrachtet:

- `gr17`, eine symmetrische $17 * 17$ Matrix aus der TSPLIB [54], einer Sammlung von Benchmark-Problemen für das TSP, und
- `pi`x`-1E`y, symmetrische $x \times x$ Matrizen mit Distanzen aus $\{0, \ldots, 10^y - 1\}$, die einem Vorschlag von Lucks folgend, „zufällig" aus den Ziffern von π erzeugt wurden – mit $(x, y) \in \{(13, 8), (14, 8), (18, 3), (18, 4), (18, 5), (18, 8)\}$.

Die Eingabe des Programms bestand aus der jeweiligen Distanzmatrix und, in Form einer Nebenbedingung, der Ziel-Länge. Zu jeder Distanzmatrix wurden fünf zufällige Touren erzeugt und ihre Längen berechnet. Von den insgesamt 35 Instanzen konnten nur die fünf `gr17`-basierten, vier der `pi-18E3`-basierten und zwei der `pi-18E4`-basierten Instanzen gelöst werden. Die dabei beanspruchten Ressourcen sind in Tabelle 3.1 zu finden. Genau gesagt, es werden die Anzahl der "branch–and–bound"-Knoten als Maß für den benötigten Speicherplatz und die Zeit in Minuten und Sekunden angegeben. Die Untersuchung der übrigen 24 Instanzen wurde nach jeweils etwa 15 Minuten Rechenzeit abgebrochen, weil der "branch–and–bound"-Baum eine Größe von etwa 30 000 Knoten erreichte und dem Programm nicht mehr genug Speicher zur Verfügung stand.

Die sieben Distanzmatrizen definieren sieben Instanzen des TSP-Minimierungsproblems. Diese Instanzen wurden mit einem einzigen "branch–and–bound"-Knoten in weniger als einer Sekunde gelöst.

Tabelle 3.1: 35 Experimente, quasi-zufällige XTSP-Instanzen zu lösen

Matrix	Knoten	Zeit
gr17	323	0:13
	333	0:14
	1193	0:47
	1575	1:06
	1658	0:47
pi18-1E3	1575	1:43
	2933	3:00
	5515	5:49
	6881	8:05
	1 abgebrochen	
pi18-1E4	157	0:07
	4361	4:29
	3 abgebrochen	
pi18-1E5	5 abgebrochen	
pi13-1E8	5 abgebrochen	
pi14-1E8	5 abgebrochen	
pi18-1E8	5 abgebrochen	

Da Thienel symmetrische ganzzahlige XTSPs betrachtete, kommen hier auf $(x-1)!/2$ Touren $x10^y$ mögliche Längen, also führen `pi18-1E3`, `pi18-1E4`, `pi18-1E5` und `pi18-1E8` zu mehr oder weniger stark unterspezifizierten Problemen, `pi13-1E8` und `pi14-1E8` dagegen zu etwa balancierten. Es konnten nur besonders stark unterspezifizierte Probleme überhaupt gelöst werden. In Abschnitt 3.2 äußerten wir die Vermutung, daß balancierte Probleme besonders schwierig sind. Thienels Ergebnisse sind ein Hinweis auf die Richtigkeit dieser Vermutung. Leider erlaubt die Zahlendarstellung von Thienels Programm keine aussagekräftigen Experimente mit überspezifizierten XTSPs oder mit balancierten XTSPs mit mehr als 14 Knoten.

Die oberen und unteren Schranken bei "branch–and–..." werden üblicherweise mit Hilfe von „Relaxationen" gewonnen. Dies sind einfachere Probleme, die stets richtige, in der Regel aber unscharfe Schranken für das Ausgangsproblem zur Lösung haben. Oft werden, wie im Falle von Thienels TSP-Löser, Lineare Programme zur Berechnung der Schranken herangezogen. Thienel weist auf die Möglichkeit hin, statt dessen die scharfen Schranken zu berechnen, d.h. TSP-Minimierungs- bzw. -Maximierungsprobleme zu lösen. Dies kann seinerseits z.B. mit "branch–and–cut" erfolgen und ist vergleichsweise einfach.

Thienel hat diesen verfeinerten Ansatz zur Lösung des XTSP nicht implementiert. Ohnehin ist fraglich, ob man so den Aufwand zum Lösen von XTSPs nennenswert drücken

kann. Für effizientes "branch–and–bound" ist die Zerlegung des Lösungsraumes ebenso wichtig wie das Finden guter Schranken. Es sollen große Teilräume entstehen, die man nicht weiter zu zerlegen braucht, weil alle in ihnen enthaltenen Lösungen entweder alle zu groß oder alle zu klein sind. Scharfe Schranken helfen, einen solchen Teilraum zu erkennen, wenn man nach einer Zerlegung auf ihn stößt – nur muß man auch auf ihn stoßen. Hier scheint der hauptsächliche Grund für die Härte des XTSPs zu liegen. Instanzen, bei denen die Ziel-Länge eine der kleinsten oder größten Längen ist, sind wahrscheinlich ähnlich einfach wie die korrespondierenden Minimierungs- bzw. Maximierungsprobleme. Bei zufällig gewählten Touren kann man aber erwarten, daß es *,viele'* kürzere und *,viele'* längere Touren gibt und eine geeignete Zerlegung deshalb schwierig ist – wenn nicht sogar praktisch unmöglich.

Die Suche nach einer Tour einer bekannten Länge legt auch Zerlegungen nahe, die für TSP-Minimierungsprobleme wenig sinnvoll wären. So kann man versuchen, Teile des Lösungsraums abzutrennen, in denen alle Längen $\not\equiv k \bmod p$ sind, wenn die Ziel-Länge $\equiv k \bmod p$ ist. Um die vorhandenen Methoden für das XTSP *,deutlich'* zu verbessern, muß man über *,großen'* Lösungs-Teilräumen Inkongruenzen modulo p nachweisen, eine Aufgabe, die sich ebenfalls als praktisch unmöglich erweisen könnte.

3.6 Hamiltonsche Pfade und ihre Bruchstücke

Gibt es Algorithmen, die – im Gegensatz zu den oben dargestellten Heuristiken – exakt analysierbar und deutlich effizienter als die vollständige Suche sind? Letztere kostet offenbar $(n-1)!$ Suchschritte, bei einer Zahlenlänge $l(n)$ ist die erforderliche Rechenzeit[4] proportional zu $O(\,l(n)\,(n-1)!\,)$. Mit Hilfe der dynamischen Programmierung kann man TSP-Minimierungsprobleme in $O(n^2 2^n)$ Schritten lösen (siehe Algorithmus B in [24]), doch ist nicht zu sehen, wie dieser Ansatz auf das XTSP übertragen werden kann.

Wir betrachten das XTSP$^{\rightsquigarrow}$, die Variante des XTSP, bei der statt eines Hamiltonkreises ein Hamiltonpfad einer gegebenen Länge gesucht wird. Das XTSP und das XTSP$^{\rightsquigarrow}$ sind effizient aufeinander reduzierbar:

Satz 3.7
Für die Probleme: XTSP-$l(n)$ und XTSP$^{\rightsquigarrow}$-$l(n)$ gilt: Wenn es einen effizienten probabilistischen Algorithmus für das eine Problem gibt, der mit der Wahrscheinlichkeit p eine Lösung für Zufallsinstanzen findet, dann gibt es einen ebensolchen Algorithmus für Zufallsinstanzen des anderen Problems, dessen Erfolgswahrscheinlichkeit p/n^2 übersteigt.

Beweis: Seien eine XTSP-Instanz (B, A) und ein effizienter Algorithmus für das XTSP$^{\rightsquigarrow}$ gegeben. Wir raten eine zufällige Kante $v \to w$ und berechnen $B' = B - a_{v,w}$. Wenn (B, A) eine Zufallsinstanz des XTSP ist, dann ist (B', A) eine Zufallsinstanz des XTSP$^{\rightsquigarrow}$. Da v und w zufällige Knoten $v \neq w$ sind, ist eine Lösung der XTSP$^{\rightsquigarrow}$–Instanz (B', A) mit der Wahrscheinlichkeit $1/(n(n-1))$ eine Lösung der XTSP-Instanz (B, A).

[4] Wir legen die logarithmischen Kosten einer RAM zugrunde (vgl. Kap. 2, S. 13).

Analog läuft der Beweis für die Umkehrung. □

Die Reduktion aus dem Beweis von Satz 3.7 ist nicht so verlustarm wie die anderen Reduktionen aus diesem Kapitel. Die Erfolgswahrscheinlichkeit kann um einen in n quadratischen Faktor sinken. Doch ist der Rechenaufwand klein: Es ist nur ein Orakel-Aufruf notwendig. Und wenn $l(n)$ monoton steigt, ist n^2 höchstens linear in der Instanzengröße.

Vollständige Suche ist nicht die beste Methode, das XTSP$^{\rightsquigarrow}$-Problem (oder das XTSP-Problem) zu lösen. Die im folgenden beschriebene Idee besteht darin, den Lösungsraum in viele disjunkte Teilräume zu zerlegen.

Hilfssatz 3.6.1 *Sei n ungerade. Es gibt einen Algorithmus, der bei Eingabe einer Instanz (B, A) des XTSP$^{\rightsquigarrow}$-$l(n)$, eines Knotens P und zweier $(n-1)/2$-elementiger Knotenmengen M_1 und M_2 in der Laufzeit $O\left(\frac{n-1}{2}! * \log\left(\frac{n-1}{2}!\right) * l(n)\right)$ einen Hamiltonpfad der Länge B findet, der durch alle Knoten aus M_1, dann durch P und zuletzt durch alle Knoten aus M_2 führt, falls ein solcher Hamiltonpfad existiert.*

Beweis: Wir betrachten den folgenden Algorithmus:

1. Berechne die Längen aller Hamiltonpfade über der Knotenmenge $M_1 \cup \{P\}$, die in P enden, und sortiere die Hamiltonpfade nach ihrer Länge zu der Liste L_1.
2. Berechne die Längen aller Hamiltonpfade über der Knotenmenge $\{P\} \cup M_2$, die in P starten, und sortiere die Hamiltonpfade nach ihrer Länge zu der Liste L_2.
3. Solange weder L_1 noch L_2 leer sind, durchlaufe die folgende Schleife:
   (a) Berechne die Summe S der Länge des längsten Hamiltonpfades $H_{\max}$ aus L_1 und der Länge des kürzesten Hamiltonpfades $H_{\min}$ aus L_2.
   (b) Im Fall $B = S$: Gib den aus den Teilpfaden $H_{\max}$ und $H_{\min}$ bestehenden Pfad aus und terminiere.
   (c) Im Fall $B < S$: Streiche $H_{\max}$ aus L_1.
   (d) Im Fall $B > S$: Streiche $H_{\min}$ aus L_2.

Die ersten beiden Schritte brauchen jeweils Zeit $O\left(\frac{n-1}{2}! * \log\left(\frac{n-1}{2}!\right) * l(n)\right)$. Die Schleife im dritten Schritt wird weniger als $\left(2 * \left(\frac{n-1}{2}!\right)\right)$-mal durchlaufen. Jeder der Teilschritte (a) bis (d) der Schleife kann in Zeit $O(l(n))$ gelöst werden. Damit ist die Schranke für die Laufzeit klar.

Nach dem zweiten Schritt gilt: Jeder Pfad, der sich aus einem Teilpfad in L_1 und einem Teilpfad in L_2 zusammensetzt, ist in der für uns zu betrachtenden Lösungsmenge und kein anderer Pfad.

Bei jedem Schleifendurchlauf tritt einer der drei Fälle $B = S$, $B < S$ oder $B > S$ ein. Im Fall $B = S$ hat man einen Pfad der gesuchten Länge gefunden. Im Fall $B < S$ wird ein

Pfad betrachtet, festgestellt, daß dessen Länge S zu groß ist, und dann wird ein Teilpfad von der weiteren Betrachtung ausgeschlossen. Die Anzahl der noch zu betrachtenden Lösungen verringert sich um bis zu $\left(\left(\frac{n-1}{2}!\right)\right)$; alle elemenierten Pfade sind zu lang. Analog argumentieren wir im Fall $B > S$, wenn zu kurze Pfade elemeniert werden. □

Die in der O-Notation *,versteckten'* Proportionalitätskonstanten sind klein; deshalb ist der im Beweis angegebene Algorithmus praktikabel. Er ermöglicht es, XTSP$^{\rightsquigarrow}$-Probleme um den Faktor $O\left(\left(\frac{n-1}{2}!\right) / \log\left(\frac{n-1}{2}!\right)\right)$ schneller zu finden als mit vollständiger Suche:

Satz 3.8
Sei n ungerade. Es gibt einen Algorithmus, der bei Eingabe einer Instanz (B, A) des XTSP$^{\rightsquigarrow}$-$l(n)$ in der Laufzeit

$$O\left(\frac{n! * \log\left(\frac{n-1}{2}!\right) * l(n)}{\left(\frac{n-1}{2}!\right)}\right)$$

einen Hamiltonpfad der Länge B findet, falls ein solcher Hamiltonpfad existiert.

Beweis: Wir zerlegen die n Knoten in einen Pivot-Knoten P und zwei Knotenmengen M_1 und M_2 mit jeweils $(n-1)/2$ Knoten. Es gibt insgesamt $n!/(\frac{n-1}{2}!)^2$ derartige Zerlegungen. Die Behauptung folgt nun aus Hilfsatz 3.6.1. □

In Verbindung mit Satz 3.7 können wir damit XTSP-Instanzen deutlich effizienter lösen als mit der Methode der vollständigen Suche. Man kann auch den Ansatz von Satz 3.8 direkt auf das XTSP übertragen, indem man zwei Pivot-Knoten P_1 und P_2 und zwei $(n-2)/2$-elementige Mengen betrachtet. Anschließend sucht man nach Touren, die in P_1 starten und enden und zwischendurch durch alle Knoten von M_1, durch P_2 und durch alle Knoten von M_2 führen (in dieser Reihenfolge). Damit gilt:

Satz 3.9
Sei n gerade. Es gibt einen Algorithmus, der bei Eingabe einer lösbaren XTSP-Instanz (B, A) in der Laufzeit

$$O\left(\frac{(n-1)! * \log\left(\frac{n-2}{2}!\right) * l(n)}{\left(\frac{n-2}{2}!\right)}\right)$$

eine Tour der Länge B findet.

Berücksichtigt man die Tatsache, daß es bei n Knoten $n!$ Hamiltonsche Pfade, aber nur $(n-1)!$ Touren (Hamiltonkreise) gibt, führen die Sätze 3.8 und 3.9 zu exakt der gleichen Beschleunigung gegenüber der vollständigen Suche.

Es ist nicht schwer, die hier dargestellten Algorithmen, ohne sie sehr zu verlangsamen, auf gerade *und* ungerade Probleme zu verallgemeinern. Um sicherzustellen, daß zufällige

bzw. quasi-zufällige XTSP-Instanzen „praktisch unlösbar“ sind, sollte man $n = 41$ und $l(n) = 160$ festlegen. Vollständige Suche über alle Touren würde $\approx 2^{159}$ Suchschritte benötigen. Mit den Methoden dieses Abschnitts braucht man mehr als 2^{100} Schritte. Selbst wenn es möglich sein sollte, balancierte XTSP-Instanzen in $\sqrt{(n-1)!}$ Schritten zu lösen, wären die erforderlichen $\approx 2^{79,5}$ Schritte gegenwärtig „praktisch undurchführbar“.

Auch wenn die Annahmen bezüglich der Härte des XTSP plausibel erscheinen – wenigstens ihrem Autor –, ist nicht auszuschließen, daß sie ganz und gar falsch sind. In der Kryptographie geht man allgemein davon aus, daß Kryptosysteme, deren Sicherheit sich auf unbewiesene Annahmen stützt, um so vertrauenswürdiger sind, je länger Experten vergeblich versucht haben, sie zu brechen. Alle Experten für Krypanalyse bzw. Effiziente Algorithmen sind aufgerufen, ihr Können an dem XTSP zu versuchen!

3.7 Die Kosten des Protokolls

Wir betrachten, welche Kosten der Einsatz unseres Identifikationsprotokolls bei Festlegung der Sicherheitsparameter $n = 41$ und $l(n) = 160$ verursacht. Als Kostenfaktoren gelten die Größen des öffentlichen und des geheimen Schlüssels, Rechen- und Speicheraufwand zur Realisierung des Protokolls und der Aufwand für die Kommunikation zwischen Prover Alice und Verifier Bob. Ein Hashwert bestehe aus 160 Bits.

Beide Schlüssel, der öffentliche (ohne Berücksichtigung der Matrix A, da diese von vielen Provern gemeinsam genutzt werden kann) und der geheime, sind erfreulich klein. Sie beanspruchen jeweils 160 Bits. Auch die Rechenzeit ist klein, da die bei der Durchführung des Protokolls anfallenden Aufgaben offensichtlich effizient lösbar sind – auch im praktischen Sinn.

Mit Blick auf die Matrizen A und A^* scheint der erforderliche Speicherplatz zur Realisierung des Protokolls beträchtlich zu sein. Wenn wir jedoch voraussetzen, daß die Matrizen A und A^* aus einem *‚kleinen’* $k(n)$-Bit-Startwert pseudozufällig erzeugt werden, z.B. unter Verwendung der Hashfunktion H, können wir die Koeffizienten der Matrizen A, $\pi(A)$, A^* und $A^* + A'$ bei Bedarf erzeugen. Dieser “time-space-tradeoff” erlaubt eine drastische Reduzierung des erforderlichen Speicherplatzes auf Kosten eines etwas größeren Rechenbedarfs.

Die gleiche Idee hilft auch, den Kommunikationsaufwand zu begrenzen. Im Durchschnitt in zwei von drei Runden sendet Alice die Matrix A^* an Bob. Statt der Matrix A^* kann Alice einfach ihren $k(n)$-Bit-Startwert senden.

Wie sollte der Sicherheisparameter $k(n)$ gewählt werden? Es gibt $2^{k(n)}$ mögliche Startwerte (bzw. $2^{k(n)}$ mögliche Commitment-Verstecke, siehe Abschnitt 3.3). Gegenwärtig gelten Suchräume der Größe 2^{80} als „praktisch undurchsuchbar“, wenn es keine bessere Suchstrategie gibt als die vollständige Suche. Wenn wir voraussetzen, daß die bei der Realisierung des Protokolls verwendeten Pseudozufallsfunktionen entsprechend sicher sind, genügt daher $k(n) = 80$.

Unser *,Trick'*, Kommunikation zu sparen, hilft nicht im Fall $c = 1$, wenn Alice das Commitment h_2 zu senden hat. Denn Inhalt des Commitments h_2 ist die Summe $A^* + A'$, während die Summanden im Fall $c = 1$ geheim bleiben müssen.

Hilfreich ist hier die Beobachtung, daß Bob die Matrix lediglich braucht, um den Wert $\text{Length}_{A^*+A'}(X')$ zu berechnen. Dafür genügen genau n Koeffizienten der Matrix. Um im Fall $c = 2$ hingegen die einschlägigen Tests durchführen zu können, muß Bob bei gegebenen Matrizen A^* und A' das Commitment h_2 über ihre Summe $A^* + A'$ überprüfen können, ohne X' zu kennen. Also müssen wir ein Commitment über eine Matrix machen, das wir entweder vollständig öffnen können durch Bekanntgabe aller Koeffizienten oder nur teilsweise durch Bekanntgabe einzelner Koeffizienten.

Die folgende Konstruktion ist vergleichbar mit der parallelisierten Version von Damgårds Hashfunktion [12] und Merkles "tree authentication" [38]. Sie ist nach Wissen des Autors jedoch neu für zero-knowledge Identifikationsprotokolle.

Die grundlegende Idee besteht darin, die gegebene Hashfunktion H nur für *,kleine'* Eingaben zu benutzen. Sei $(x_1, \ldots, x_m)$ mit $x_i \in \{0,1\}^{l(n)}$ eine *,große'* Eingabe. Für $i < j$ berechnen wir statt $H(x_1, \ldots, x_m)$ den Wert $H^*(x_1, \ldots, x_m)$ mit

$$H^*(x_i, \ldots, x_j) = H\left(H^*(x_i, \ldots, x_{i-1+\lceil (j-i)/2 \rceil}), H^*(x_{i+\lceil (j-i)/2 \rceil}, \ldots, x_j)\right)$$

und $H^*(x_i) = H(x_i)$. Man stelle sich einen binären Baum mit der Wurzel $H^*(x_1, \ldots, x_m)$ und den Blättern $x_1, \ldots, x_m$ vor.

Sei $h = H^*(x_1, \ldots, x_m)$ gegeben. Um $x = x_i$ für einen Wert x nachzuweisen, genügt es, den Index i und maximal $\lceil \log_2 m \rceil$ Zwischenresultate $H^*(\ldots)$ bekanntzugeben. Zur Verifikation ist der Baum vom Blatt x_i bis zur Wurzel $H^*(x_1, \ldots, x_m)$ zu durchlaufen.

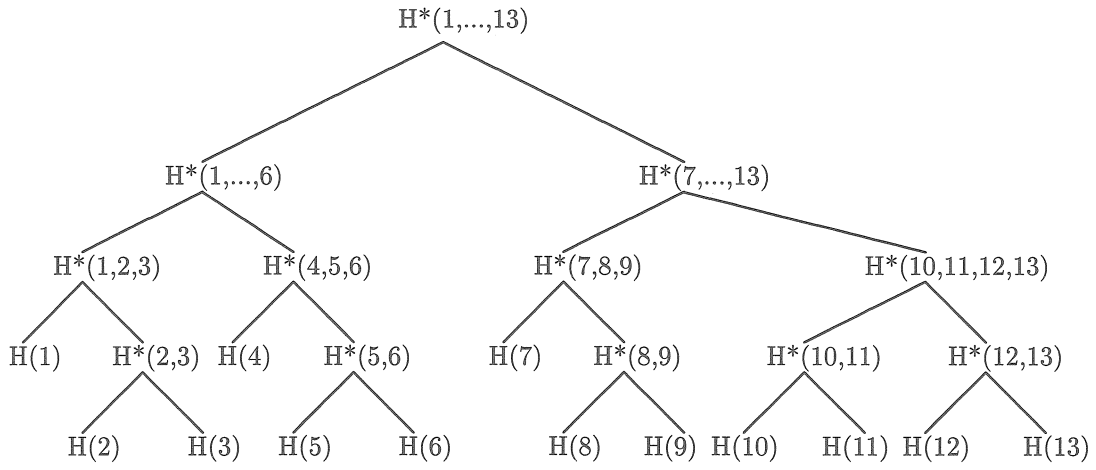

Abbildung 3.1: H*(1,...,13)

In Abbildung 3.1 ist der Hash-Baum des Vektors $(X_1, ..., X_{13}) = (1, ..., 13)$ als Beispiel dargestellt. Um $X_6 = 6$ nachzuweisen, braucht man die Zwischenwerte H(5), H(4), H*(1,2,3) und H*(7,...,13), für $X_7 = 7$ genügen H*(8,9), H*(10,11,12,13) und H*(1,...,6).

Unser Fall ist etwas komplizierter. Zur Berechung des Commitments h_2 über $A^+ = A^* + A' = (a^+_{i,j})$ verwenden wir die Fomel[5]

$$h'_2 = H\left(H^*(a^+_{1,1}, \ldots, a^+_{1,n}),\ \ldots,\ H^*(a^+_{n,1}, \ldots, a^+_{n,n}) \right).$$

Statt $h_2 = H(A^* + A', r_2)$, wobei $r_2 \in_R \{0,1\}^{k(n)}$ das Commitment-Versteck für h_2 ist, schreiben wir dann

$$h_2 = H(h'_2, r_2).$$

Im Fall der Herausforderung $c = 1$ hat Alice n Werte $a^+_{i,j}$ preiszugeben, nämlich einen Wert a^+_{1,j_1}, einen Wert a^+_{2,j_2}, ... und einen Wert a^+_{n,j_n}. Um den Wert a^+_{i,j_i} als Koeffizient der Matrix nachzuweisen, genügt es, $\lceil \log_2 n \rceil$ oder $\lfloor \log_2 n \rfloor$ Zwischenwerte bekanntzugeben, anstelle der $n-1$ Blätter $a^+_{i,j}$ mit $j \neq j_i$.

Ohne auf Details einzugehen, sei darauf hingewiesen, daß sich für Alice mehrere Möglichkeiten eines "time-space-tradeoffs" bei der Implementation bieten:

- Schon bei der Berechnung von h_2 kennt Alice die Werte a^+_{i,j_i}, die sie im Fall $c = 1$ an Bob senden muß. Auch alle Zwischenwerte werden in dieser Phase berechnet. Um Rechenzeit einzusparen, kann sie die zugehörigen Zwischenwerte speichern und sich ihre erneute Berechung ersparen.
- Der Rechenaufwand für die Neuberechnung eines Zwischenwertes verdoppelt sich mit jedem Schritt auf dem Weg vom Blatt zur Wurzel. Wenn es nicht möglich ist, alle erforderlichen Zwischenwerte zu speichern, sollte Alice sich auf die *,nahe an der Wurzel liegenden'* beschränken.
- Besonders aufwendig für Alice ist die Berechnung der Hashwerte h_0 und h_2. Die entsprechenden Werte kann Alice *,auf Vorrat'* berechnen, also beliebig lange vor der tatsächlichen Kontaktaufnahme mit Bob, und speichern.

Wer eine praktische Verwendung unseres Protokolls erwägt, sollte besonders auf die Kommunikationskosten achten.

In der Literatur versucht man bei public-key Identifikationsprotokollen üblicherweise, die Anzahl der Runden so zu wählen, daß die Wahrscheinlichkeit eines unentdeckten Betruges unter 1 / 1 Million liegt. Wir brauchen dafür mindestens 35 Runden. Trotz des Tricks, mit "tree authentication" Kommunikation zu sparen, dominiert der Aufwand im Fall $c = 1$ den gesamten Kommunikationsaufwand. Im Durchschnitt tritt in jeder dritten Runde der Fall $c = 1$ auf. Dann sind $41 * \log_2(41) * 160$ Bits $\approx 35\,150$ Bits zu übertragen – bei 35 Runden $\frac{35}{3} * 35\,150$ Bits $\approx 410\,000$ Bits. Eine langsame ("low-bandwidth") Smartcard, wie Blaze [8] sie beschreibt, kann 9600 Bits pro Sekunde übertragen. Pro Identifikation verbraucht sie also durchschnittlich fast 45 Sekunden Kommunikationszeit. Dies ist zwar nicht völlig unpraktikabel, könnte von ihren Nutzern aber als *,an der*

[5] Die Werte $a^+_{i,i}$ sind unwichtig und könnten ganz ausgelassen werden. Eine Berechnung des Commitments h_2 ohne die Blätter $a^+_{i,i}$ wäre etwas effizienter, seine Darstellung aber unübersichtlicher.

Grenze des Erträglichen' empfunden werden. Andere, vermutlich ebenso sichere Identifikationsprotokolle, etwa das von Shamir [63], erfordern weniger Kommunikation.

Ein praktischer Einsatz des hier dargestellten Protokolls ist nur angebracht, wenn Prover und Verifier schnell bzw. billig kommunizieren können – zumindest in der Richtung vom Prover zum Verifier.

3.8 Einweg-Hashing und das Erzeugen von Pseudozufallsbits

In den vorangegangenen Abschnitten haben wir die vermutete Härte des XTSP für ein Identifikationsprotokoll ausgenutzt. Da balancierte Zufallsprobleme nicht wesentlich leichter als andere Zufallsprobleme sein können, beschränkten wir uns auf ungefähr balancierte Probleme. Grundsätzlich ist es für unser Protokoll gleichgültig, ob ein über- oder ein unterspezifiziertes oder ein balanciertes Problem vorliegt. In diesem Kapitel beschäftigen wir uns mit Kryptosystemen, die natürlicherweise mit über- bzw. unterspezifizierten Problemen verbunden sind.

Die folgenden Überlegungen basieren auf der Arbeit von Lucks [31]. Schon bisher haben wir die Length_A-Funktion bezüglich einer zufälligen Distanz-Matrix A wie eine Einwegfunktion benutzt. Wir wollen – ohne den Begriff der Einweg-Funktion formal zu definieren – an diesem Ansatz festhalten. Wenn die Parameter n und $l(n)$ so gewählt sind, daß die Anzahl der Eingabe-Bits die der Ausgabe-Bits übersteigt, d.h., wenn $m := \lfloor \log_2((n-1)!) \rfloor > l(n)$ ist und wenn das korrespondierende XTSP-$l(n)$ (trotzdem) praktisch unlösbar ist, dann ist Length_A eine Einweg-Hashfunktion.

Satz 3.10
Seien $l(n) < \lfloor \log_2((n-1)!) \rfloor$ und $A \in_R \mathbb{Z}_{2^{l(n)}}^{n \times n}$. Wenn es einen probabilistischen Algorithmus gibt, der effizient mit der Erfolgswahrscheinlichkeit p die mutmaßliche Einweg-Hashfunktion $f_m(X) = Length_A(X)$ bricht, dann gibt es einen effizienten probabilistischen Algorithmus, der mit einer Erfolgswahrscheinlichkeit über $2p/n$ Zufallsinstanzen des XTSP löst.

Beweis: Es ist das folgende zu zeigen: Wenn man die mutmaßliche Einweg-Hashfunktion $f_m(X) = \text{Length}_A(X)$ effizient brechen kann, dann kann man auch Zufallsinstanzen des XTSP-$l(n)$ effizient lösen. Fast wie beim Beweis von Satz 3.4 betrachten wir den folgenden Algorithmus:

1. Wähle eine Zufallstour X und berechne $B' = \text{Length}_A(X)$. Ist $B = B'$, terminiere.
2. Wähle eine Zufallskante $v \to w$ auf X und berechne $A' = (a'_{i,j})$ mit $a'_{i,j} = a_{i,j}$ für $(i \neq v$ oder $j \neq w)$ und $a'_{v,w} = a_{v,w} - B' + B$. Dann gilt $\text{Length}_{A'}(X) = B$.
3. Versuche, eine Tour $Y \neq X$ mit $\text{Length}_A(X) = \text{Length}_A(Y)$ zu finden.
4. Wenn die Kante $v \to w$ auf Y liegt, gilt $\text{Length}_A(Y) = B'$, sonst gilt $\text{Length}_A(Y) = B$, und Y ist eine Lösung für (B, A).

Nach Voraussetzung wird im dritten Schritt des Algorithmus mit der Wahrscheinlichkeit p eine solche Tour Y gefunden. Wegen $X \neq Y$, und weil die Kante $v \to w$ zufällig ist, liegt die Kante $v \to w$ mit mehr als der Wahrscheinlichkeit $2p/n$ nicht auf Y. □

Unsere Methode, bei $l(n) < \lfloor \log_2((n-1)!) \rfloor$ eine Length_A-Funktion als Hashfunktion einzusetzen, führt uns sogar zu Familien universeller Einweg-Hashfunktionen.

Satz 3.11
Sei $l(n) < \lfloor \log_2((n-1)!) \rfloor$. Wenn für eine Zufallsmatrix A die Funktion $Length_A$ mit überwältigender Wahrscheinlichkeit eine Einweg-Hashfunktion ist, dann konstituiert

$$F = \{Length_M \,|\, M \in \mathbb{Z}_{2^{l(n)}}^{n*n}\}$$

eine Familie universeller Einweg-Hashfunktionen.

Beweis: Seien A eine zufällige Distanzmatrix und X eine Zufallstour. Sei die Tour X^* *beliebig* gewählt, jedoch ohne Wissen um A oder X. Dann ist es leicht, eine Permutation π zu bestimmen, für die $\pi(X) = X^*$ gilt. Offenbar ist $\pi(A)$ eine Zufallsmatrix[6] – wie A. Wäre F keine Familie universeller Einweg-Hashfunktionen, könnte man effizient eine Tour Y^* finden mit $\text{Length}_{\pi(A)}(X^*) = \text{Length}_{\pi(A)}(Y^*)$. In diesem Fall könnte man auch $Y = \pi^{-1}(Y^*)$ berechnen, es wäre $\text{Length}_A(X) = \text{Length}_A(Y)$ erfüllt. Also wäre für eine Zufallsmatrix A die Einweg-Hashfunktion Length_A gebrochen. □

Die Sicherheit unserer Hashfunktionen haben wir auf die Härte unterspezifizierter Probleme zurückgeführt. Im Falle überspezifizierter XTSPs liegt eine Length_A-Funktion vor, deren Ausgabe länger als die Eingabe ist. Mit Hilfe der Length_A-Funktion können wir also *‚Bits erzeugen'*. Wenn es praktisch unmöglich ist, die Ausgabe der Length_A-Funktion von *‚tatsächlich zufällig erzeugten'* Bits zu unterscheiden, dann können wir mit der Length_A-Funktion pseudozufällige Bit-Strings erzeugen.

Die Aufgabe, bei einer gegebenen Zufallsmatrix zwischen Zufallswerten und Längen bezüglich A einerseits und Zufallstouren andererseits zu entscheiden, ist die *Entscheidungsvariante* des XTSP. Könnte ein Orakel für die Entscheidungsvariante auch bei der Lösung der Suchvariante von Nutzen sein?

Es bezeichne Δ einen probabilistischen Algorithmus mit XTSP-Instanzen als Eingabe und einer Ausgabe $\in \{0,1\}$. Mit $P_{\text{zufällig}}$ bezeichnen wir die Wahrscheinlichkeit, daß Δ eine 1 liefert, wenn die Eingabe eine Zufallsinstanz ist. Mit P_{quasi} bezeichnen wir die Wahrscheinlichkeit, daß Δ eine 1 liefert, wenn die Eingabe quasi-zufällig ist. Uns interessieren nicht die Wahrscheinlichkeiten $P_{\text{zufällig}}$ und P_{quasi} selbst. Vielmehr interessiert uns, wie gut Δ zwischen zufälliger und quasi-zufälliger Eingabe *unterscheiden* kann. Deshalb betrachten wie die *Wahrscheinlichkeitsdifferenz* von Δ

$$P_\Delta = \left| P_{\text{quasi}} - P_{\text{zufällig}} \right|.$$

[6] Es sei an die Notation aus Abschnitt 3.3 erinnert!

Ebenso bezeichne Γ einen Algorithmus, bei Eingabe einer XTSP-Instanz (B, A) mit $B = \text{Length}_A(X)$ und einer Kante $v \to w$ zu unterscheiden, ob $v \to w$ auf X liegt oder nicht. Analog definieren wir auch die *Wahrscheinlichkeitsdifferenz* P_Γ.

Satz 3.12
Seien $l(n) \geq 1 + \log_2((n-1)!)$ *und* $A \in_R \mathbb{Z}_{2^{l(n)}}^{n \times n}$. *Es gebe einen effizienten probabilistischen Algorithmus* Δ, *um zwischen zufälligen und quasi-zufälligen Instanzen des XTSP mit der Wahrscheinlichkeitsdifferenz* P_Δ *zu unterscheiden. Dann gibt es auch einen ebensolchen Algorithmus* Γ, *der mit der Wahrscheinlichkeitsdifferenz* $P_\Gamma = P_\Delta$ *für eine quasi-zufällige XTSP-Instanz* (B, A) *mit* $B = Length_A(X)$ *und eine Zufallskante* $v \to w$ *entscheidet, ob die Zufallskante* $v \to w$ *auf der Tour* X *liegt.*

Beweis: Der Beweis beruht auf dem folgenden Algorithmus:

1. Gegeben sei eine quasi-zufällige XTSP-Instanz (B, A) mit $B = \text{Length}_A(X)$.
2. Wir wählen eine Zufallskante $v \to w$ und berechnen, analog zum Beweis von Satz 3.10, eine Matrix A', die bis auf den Koeffizienten $a'_{v,w}$ gleich der Matrix A ist. Wir wählen einen Zufallswert $r \in_R \mathbb{Z}_{2^{l(n)}}$ und setzen $a'_{v,w} := a_{v,w} + r \pmod{2^{l(n)}}$ sowie $B' = B + r \pmod{2^{l(n)}}$.
3. Wir testen, ob die Instanz (B', A') zufällig oder quasi-zufällig ist.

Wenn $v \to w$ auf X liegt, gilt $\text{Length}_{A'}(X) = B'$, und (B', A') ist eine quasi-zufällige Instanz des XTSP. Andernfalls gilt $\text{Length}_{A'}(X) = B'$. Dann ist (B', A') eine Zufallsinstanz. □

Bei den Beweisen der Sätze 3.10, 3.11 und 3.12 treten Reduktionen auf, die, wenn man die gleichen Maßstäbe anlegt wie der Autor bei den Beweisen der Sätze 3.3 und 3.4 (siehe den vorletzten Absatz aus Abschnitt 3.2), als verlustarm einzustufen sind.

3.9 Anmerkungen zu den Kryptosystemen aus Abschnitt 3.8

Wie sollten die Sicherheitsparameter für über- oder unterspezifizierte Probleme gewählt werden? Unser einziger Anhaltspunkt ist die Annahme, daß etwa balancierte Zufallsinstanzen (und damit auch etwa balancierte quasi-zufällige Instanzen) des XTSP bei $n = 41$ Städten praktisch unlösbar sind. Dies führte zur Wahl $l(n) = 160$. Das Prinzip der Vorsicht gebietet, von diesen Werten auszugehen und auf die Härte von Problemen zu vertrauen, die nicht *‚sehr stark'* über- bzw. unterspezifiziert sind.

Für die *Hashfunktion* kann man etwa $n = 46$ und $l(n) = 170$ verwenden. Es ist $45! \approx 2^{186,3}$, die resultierende Hashfunktion bildet 186 Eingabe-Bits auf 170 Ausgabe-Bits ab. Mit jeder Berechnung der Länge einer TSP-Tour können wir damit 16 Bits verarbeiten.

Zur *Erzeugung pseudozufälliger Bits* kann man auf $n = 44$ und $l(n) = 192$ zurückgreifen. Es ist $43! \approx 2^{175,3}$, deshalb bildet der entsprechende Pseudozufalls-Funktionsgenerator 176 Eingabe-Bits auf 192 Ausgabe-Bits ab, erzeugt also 16 pseudozufällige Bits pro Aufruf.

Die Eingaben für die Hashfunktion und den Pseudozufalls-Bitgenerator sind TSP-Touren, auf geeignete Weise als Bit-Strings einer vorgegebenen Länge codiert. Es ist trivial, eine geeignete Codierung zu finden. Allerdings gilt für $n > 3$:

- Im Falle des Hashings sind bestimmte Touren als Eingabe nicht zulässig.
- Ebenso sind manche Touren als Eingabe für den Pseudozufalls-Funktionsgenerator doppelt so wahrscheinlich wie andere.

Beide Fakten hängen damit zusammen, daß die Anzahl der Bit-Strings eine Zweierpotenz ist, aber $(n-1)!$ für $n > 3$ nie eine Potenz von zwei sein kann. Beide Fakten führen nur zu einer marginalen Schwächung der entsprechenden Kryptosysteme.

Beide Kryptosysteme benötigen eine Matrix von Zufallsbits. Unser Vorschlag für die Wahl der Sicherheitsparameter erfordert bei den Einweg-Hashfunktionen die Speicherung von $46 * 45 * 170$ Bits, also etwa 43 Kilobytes (KB). Im Falle des Pseudozufalls-Bitgenerators sind es $44 * 43 * 102$ Bits $\approx 44,3$ KB. Dieser Speicheraufwand stellt beispielsweise im Hinblick auf handelsübliche PCs kein Problem dar. Anders kann es aber bei *'kleinen'* Rechnern aussehen, z.B. bei Smartcards.

Was kann man tun, um den Speicherbedarf zu vermindern? Im Falle des Identifikationsprotokolls besteht die Möglichkeit, die Distanzen $a_{i,j}$ bei Bedarf pseudozufällig zu erzeugen, anstatt sie zu speichern. Dies bringt uns hier wenig, insbesondere im Fall des Pseudozufalls-Bitgenerators.

Eine mögliche Alternative scheint darin zu bestehen, n zufällige Punkte in einer endlichen Ebene (oder einem Raum höherer Dimension) zu erzeugen und die Distanzen $a_{i,j}$ als Entfernung[7] zwischen Punkt i und j zu berechnen. Natürlich gilt $a_{i,j} = a_{j,i}$, d.h., das resultierende Problem ist ein symmetrisches XTSP. Problematisch erscheint aber nicht die Symmetrie, sondern die Tatsache, daß in dem resultierenden XTSP die Dreiecksungleichung gilt. Zwar ist nicht offensichtlich, wie man die Dreiecksungleichung ausnutzen kann, um effizient XTSP-Instanzen zu lösen. Es gibt jedoch effiziente deterministische Approximationsalgorithmen für Instanzen des TSP-Minimierungsproblems, bei denen die Dreiecksungleichung gilt. Im Hinblick auf die Sicherheit unserer Kryptosysteme erscheint es wenig ratsam, auf die praktische Unlösbarkeit von Problemen mit einer so starken inneren Struktur zu vertrauen. Wenn P≠NP ist, können keine derartigen Algorithmen für das allgemeine TSP-Minimierungsproblem existieren.

Wie praktikabel sind unsere Kryptosysteme? Wenn der erforderliche Speicherplatz zur Verfügung steht, ist die Geschwindigkeit der ausschlaggebende Faktor. Schon in der

[7] In Frage kommen z.B. eine ganzzahlige Näherung der Euklidischen Entfernung oder die Manhattan-Distanz.

Einleitung haben wir den systembasierten und den komplexitätstheoretischen Ansatz in der Kryptographie beschrieben. Beim Entwurf unserer Kryptosysteme sind wir dem komplexitätstheoretischen Ansatz gefolgt. Wir verzichten in dieser Arbeit darauf, eine detaillierte Analyse der in der Praxis zu erreichenden Effizienz durchzuführen. Bei der Realisierung unserer Kryptosysteme sind die folgenden beiden Schritte durchzuführen:

1. Die Umwandlung einer Binärzahl $X \in \mathbb{Z}_{2^{l(n)}}$ in eine Folge $(i_2, \ldots, i_n)$ von $n-1$ verschiedenen Zahlen aus $\{2, \ldots n\}$ und

2. die Berechnung von $B = a_{1,i_1} + a_{i_1,i_2} + a_{i_2,i_3} + \cdots + a_{i_{n-1},i_n} + a_{i_n,i_1}$ modulo $2^{l(n)}$.

Es ist $B = \text{Length}_A(X)$, und die vorgeschlagene Größe der Sicherheitsparameter n und $l(n)$ ist moderat ($n < 50$, $l(n) < 200$). Auch ohne einen detaillierten Vergleich der Laufzeiten ist leicht einzusehen, daß unsere Kryptosysteme drastisch effizienter sind als ihre auf zahlentheoretischen Problemen beruhenden Gegenstücke, die auf Multiplikationen modulo N, z.B. $N \approx 2^{1000}$, beruhen. Damit wird der Effizienznachteil des komplexitätstheoretischen Ansatzes gegenüber dem systembasierten deutlich verkleinert. Die in diesem Kapitel untersuchten Kryptosysteme erscheinen damit geeignet, jene Lücke zu schließen, die der Vertrauensverlust in die Härte des Subset Sum Problems gerissen hat. Wie das Subset Sum Problem, so sollten einschlägige Experten auch das XTSP noch genauer auf seine praktische Lösbarkeit untersuchen!

4 Luby-Rackoff Chiffren

Unter Verwendung von *Zufallsfunktionen* beschrieben Luby und Rackoff [29] beweisbar sichere *Blockchiffren.* Dies war ein gefeierter theoretischer Durchbruch. Luby und Rackoff wiesen darauf hin, daß ihr Resultat auch für den Entwurf praktikabler Kryptosysteme nutzbar ist. Statt *,echter'* Zufallsfunktionen verwendet man pseudozufällige Funktionen, die von geheimen Schlüsseln abhängig sind. Die Sicherheit einer derartigen Blockchiffre hängt natürlich von der Sicherheit der eingesetzten pseudozufälligen Funktionen ab. Ein in der Krypto-Folklore beliebter Ansatz besteht darin, dedizierte Hashfunktionen als Pseudozufallsfunktionen einzusetzen – „zufälliges Verhalten" ist ohnehin ein Kennzeichen „guter Hashfunktionen" [53].

Bemerkenswert ist der einfache Aufbau von Luby-Rackoff Chiffren aus Pseudozufallsfunktionen. 1994 riefen Needham und Wheeler dazu auf, Chiffren zu konstruieren, die nicht nur effizient und sicher sein sollten, sondern auch einfach und kurz zu beschreiben. Ihr "tiny encryption algorithm" (TEA) sei *"short enough to be programmed from memory or a copy"*, so Needham und Wheeler [45]. Wenn man das Vorhandensein geeigneter Pseudozufallsfunktionen voraussetzt, sind Luby-Rackoff Chiffren einfacher und kürzer zu beschreiben als TEA. Als Pseudozufallsfunktionen erscheinen dedizierte Hashfunktionen gut geeignet. Dies ist u.a. von rechtlicher und politischer Bedeutung. In einigen Ländern sind Import, Export oder Besitz von Verschlüsselungssoftware und -hardware verboten. Derartige Verbote kann man durch Einsatz von Luby-Rackoff Chiffren und mit Hilfe dedizierter Hashfunktionen umgehen. Letztere unterliegen meist keinen rechtlichen Einschränkungen, denn sie dienen eigentlich der Authentifikation, nicht der Verschlüsselung.

Feistel-Chiffren sind aus einfachen Grundoperationen, sogenannten Runden, zusammengesetzt, die wiederholt angewandt werden. Luby-Rackoff Chiffren sind spezielle Feistel-Chiffren mit drei oder vier Runden und besonders starken, nämlich pseudozufälligen Rundenfunktionen. Viele heutzutage praktisch verwendeten Blockchiffren sind Feistel-Chiffren. Die Rundenfunktionen sind meist nicht pseudozufällig, dafür ist die Anzahl der Runden größer. Beispiele sind DES [41], Blowfish [57] und TEA [45].

4.1 Sichere unbalancierte Luby-Rackoff Chiffren

Luby und Rackoff betrachteten Chiffren mit einem balancierten Aufbau. Wir untersuchen die unbalancierte Verallgemeinerung ihrer Chiffren. Schneier und Kelsey [60] untersuchten "unbalanced Feistel networks" aus einer systemtheoretischen Sichtweise, gaben aber keine Sicherheitsbeweise.

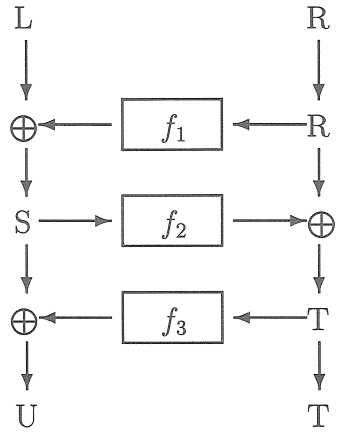

Abbildung 4.1: Permutation $\psi(f_1, f_2, f_3)(L, R) = (U, T)$

Seien f_1, f_2 und f_3 Zufallsfunktionen[1] $f_1, f_3 : \{0,1\}^r \longrightarrow \{0,1\}^l$ und $f_2 : \{0,1\}^l \longrightarrow \{0,1\}^r$. Es bezeichne „$\oplus$" das Bit-weise Exklusiv-Oder (XOR). Wir berechnen $S, U \in \{0,1\}^l$ und $T \in \{0,1\}^r$ durch $S = L \oplus f_1(R)$, $T = R \oplus f_2(S)$ und $U = S \oplus f_3(T)$. So erhalten wir die Permutation $p(L, R) = \psi(f_1, f_2, f_3)(L, R) = (U, T)$ über $\{0,1\}^{l+r}$, siehe Abbildung 4.1. Die Umkehrung von g ist $g^{-1} = \psi(f_3, f_2, f_1)$.

Luby and Rackoff fanden in ihrer berühmten Arbeit [29] eine obere Schranke für die Erfolgswahrscheinlichkeit eines Chosen-Plaintext Angriffs – abhängig von der Anzahl der gewählten Klartexte. Maurer [36] fand einen einfacheren Beweis. Luby, Rackoff und auch Maurer beschränkten sich auf den balancierten Fall $l = r$. Der folgende Beweis für den allgemeinen Fall basiert auf Maurers Beweis. Wir folgen dem Beweisgang, um später die Gültigkeit des Beweises sogar unter schwächeren Voraussetzungen zu demonstrieren.

Satz 4.1
Sei $g : \{0,1\}^{r+l} \longrightarrow \{0,1\}^{r+l}$ entweder eine zufällig gewählte Funktion oder von der Form $g = \psi(f_1, f_2, f_3)$ mit zufällig gewählten Funktionen $f_1, f_3 : \{0,1\}^r \longrightarrow \{0,1\}^l$ und $f_2 : \{0,1\}^l \longrightarrow \{0,1\}^r$.

Der Algorithmus A habe Zugriff auf ein Orakel zur Berechung von g. Die Ausgabemenge von A ist $\{0,1\}$. Mit P_{rand} (bzw. P_{perm}) bezeichnen wir die bedingte Wahrscheinlichkeit, daß A eine 1 ausgibt wenn g zufällig gewählt wurde (bzw. wenn g von der Form $g = \psi(f_1, f_2, f_3)$ mit zufällig gewählten Funktionen f_i ist).

Wenn A höchstens q Eingaben $(L_1, R_1), \ldots, (L_q, R_q)$ für das Orakel wählt und die kor-

[1] Die Funktion f_i bezeichnen wir als *Rundenfunktion* der i-ten Runde.

respondierenden Ausgaben $(U_1, T_1), \ldots, (U_q, T_q)$ *mit* $(U_i, T_i) = g(L_i, R_i)$ *erhält, gilt:*

$$|P_{rand} - P_{perm}| < \frac{q^2}{2^n}. \tag{4.1}$$

Beweis: Sei $g = \psi(f_1, f_2, f_3)$. Wenn g sich für die von A gewählten Eingaben wie eine Zufallsfunktion verhält, ist g nicht von einer Zufallsfunktion zu unterscheiden. Es nützt A nichts, eine Frage zu wiederholen. O.B.d.A. sei $(L_i, R_i) \neq (L_j, R_j)$ für $i \neq j$, d.h. $((L_i \neq L_j)$ oder $(R_i \neq R_j))$.

Für $i \neq j$, bezeichne $p^{(=)}(S_i, S_j)$ die Wahrscheinlichkeit, daß S_i und S_j *kollidieren*, d.h. daß $S_i = S_j$ ist. Es bezeichne $p^{(\neq)}(S_1, \ldots, S_q)$ die Wahrscheinlichkeit, daß keine *S-Kollision* auftritt, $S_1, \ldots, S_q$ also paarweise verschieden sind.

Wenn $R_i = R_j$ ist, dann gilt $L_i \neq L_j$ und $S_i \neq S_j$. Im Fall $R_i \neq R_j$ sind, da f_1 eine Zufallsfunktion ist, $f_1(R_i)$ und $f_1(R_j)$ zwei voneinander unabhängige l-Bit Zufallswerte, ebenso wie S_i und S_j. Damit gilt:

$$p^{(=)}(S_i, S_j) \leq 2^{-l}. \tag{4.2}$$

Es gibt $q(q-1)/2$ Mengen $\{i, j\} \in \{1, \ldots, q\}$ mit $i \neq j$, deshalb ist

$$p^{(\neq)}(S_1, \ldots, S_q) \geq 1 - \frac{q(q-1)/2}{2^l} \geq 1 - \frac{q(q-1)/2}{2^n}. \tag{4.3}$$

Es gilt $(L_i, R_i) \neq (L_j, R_j) \Longleftrightarrow (S_i, R_i) \neq (S_j, R_j)$. Damit folgt analog

$$p^{(\neq)}(T_1, \ldots, T_q) \geq 1 - \frac{q(q-1)/2}{2^r} \geq 1 - \frac{q(q-1)/2}{2^n}.$$

Es bezeichne p^* die Wahrscheinlichkeit, daß weder S-Kollisionen noch T-Kollisionen auftreten. Kombinieren von $p^{(\neq)}(S_1, \ldots, S_q)$ und $p^{(\neq)}(T_1, \ldots, T_q)$ ergibt

$$p^* \geq 1 - \frac{q(q-1)}{2^n} > 1 - \frac{q^2}{2^n}.$$

Wenn weder S-Kollisionen noch T-Kollisionen vorliegen, sind $f_2(S_1), \ldots, f_2(S_q)$ und $f_3(T_1), \ldots, f_3(T_q)$ voneinander unabhängige Zufallswerte. Dann sind auch die Chiffretexte $(U_1, T_1), \ldots, (U_q, T_q)$ voneinander unabhängige Zufallswerte, nur mit der Einschränkung, daß $T_i \neq T_j$ für $i \neq j$ gilt. □

4.2 Untersuchung der ersten Runde

Beim Beweis von Satz 4.1 haben wir lediglich ausgenutzt, daß unterschiedliche $(l + r)$-Bit Eingaben (L_i, R_i) und (L_j, R_j) mit hoher Wahrscheinlichkeit bei der Berechung von

$\psi(f_1, f_2, f_3)$ zu unterschiedlichen l-Bit Zwischenwerten S_i und S_j führen. Dafür muß die Funktion f_1 nicht notwendigerweise eine Zufallsfunktion sein.

Sei $\min\{d, r, m\} \geq n$. Die Funktion $F : \{0,1\}^d \times \{0,1\}^r \longrightarrow \{0,1\}^m$ bezeichnen wir als *Differenz-Konzentrator*, wenn für $K \in_R \{0,1\}^d$, alle $x \neq y$ und alle c die Gleichung

$$F(K,x) \oplus F(K,y) = c \tag{4.4}$$

höchstens mit der Wahrscheinlichkeit 2^{-n} erfüllt ist. Bei zufällig gewähltem K nennen wir $f : \{0,1\}^r \longrightarrow \{0,1\}^m$ mit $f(x) = F(K,x)$ eine *Differenz-Konzentrator-Funktion.*

Satz 4.2
Sei $F : \{0,1\}^d \times \{0,1\}^r \longrightarrow \{0,1\}^l$ ein Differenz-Konzentrator. Wenn die Bedingungen für Satz 4.1 erfüllt sind, außer für $f_1(R) = F(K,R)$ mit $K \in_R \{0,1\}^d$, dann gilt Ungleichung (4.1), also $|P_{rand} - P_{perm}| < \frac{q^2}{2^n}$.

Beweis: Sei $K \in_R \{0,1\}^d$ und $f_i(x) = F(K,x)$. Wenn Ungleichung (4.3) erfüllt ist, d.h.

$$p^{(\neq)}(S_1, \ldots, S_q) \geq 1 - \frac{q(q-1)/2}{2^n},$$

dann bleibt der Beweis von Satz 4.1 unberührt. Wenn bei der Berechung von $\psi(f_1, f_2, f_3)$ Zwischenwerte $S_i = S_j$ für $i \neq j$ auftreten, dann sind die rechten Seiten der zugehörigen Eingaben ungleich, also $R_i \neq R_j$, und es gilt

$$\begin{array}{rrcccccl} & f_1(R_i) & \oplus & L_i & = & f_1(R_j) & \oplus & L_j, \\ \Longleftrightarrow & f_1(R_i) & \oplus & f_1(R_j) & = & L_i & \oplus & L_j = c, \\ \Longleftrightarrow & F(K,R_i) & \oplus & F(K,R_j) & = & L_i & \oplus & L_j = c. \end{array}$$

Weil F ein Differenz-Konzentrator ist, kann die dritte Gleichung für alle c, R_i, R_j mit $R_i \neq R_j$ höchstens mit der Wahrscheinlichkeit 2^{-n} erfüllt sein. Deshalb gilt $p^{(=)}(S_i, S_j) \leq 2^{-n}$. Es folgt Ungleichung (4.3). □

Offenbar sind Generatoren für Zufallsfunktionen spezielle Differenz-Konzentratoren. Es gibt auch andere Differenz-Konzentratoren.

Sei $K = (k_{i,j}) \in_R \{0,1\}^{n \times r}$ ein geheimer Schlüssel. Wir definieren $F_K : \{0,1\}^r \longrightarrow \{0,1\}^n$ durch $F_K(x) = F_K(x_1, \ldots, x_r) = (F_K^1(x), \ldots, F_K^n(x))$ mit $F_K^i(x) = 0$ wenn $x = (0, \ldots, 0)$, andernfalls

$$F_K^i(x_1, \ldots, x_r) = \bigoplus_{j=1}^{r} k_{i,j} x_j.$$

Es gilt $F_K(0, \ldots, 0) = (0, \ldots, 0)$ und, für alle $x, y \in \{0,1\}^r$, $F_K(x) \oplus F_K(y) = F_K(x \oplus y)$. Also ist F_K weder zufällig noch pseudozufällig – wohl aber eine Differenz-Konzentrator-Funktion.

Satz 4.3
Die Funktion F mit $F(K,x) = F_K(x)$ ist ein Differenz-Konzentrator.

Beweis: Für alle $x, y \in \{0,1\}^r$, haben wir

$$F_K^i(x) \oplus F_K^i(y) \;=\; \left(\bigoplus_{j=1}^{r} k_{i,j}x_j\right) \oplus \left(\bigoplus_{j=1}^{r} k_{i,j}y_j\right) \;=\; \bigoplus_{j=1}^{r} k_{i,j}(x_j \oplus y_j) \;=\; F_K^i(x \oplus y).$$

Ist $x \neq y$, dann ist $F_K(x \oplus y)$ das Bit-weise XOR eines oder mehrerer zufällig gewählter n-Bit Vektoren $((k_{1,1}, \ldots, k_{1,n}), \ \ldots \ (k_{r,1}, \ldots, k_{r,n}))$ und damit selbst ein zufällig gewählter n-Bit Vektor. □

Die Bedingung „f_1 ist eine Differenz-Konzentrator-Funktion" ist hinreichend für die Gültigkeit von Satz 4.1 und tatsächlich schwächer als die ursprüngliche Forderung nach einer Zufallsfunktion. Wie der folgende Satz zeigt, ist die Bedingung sogar notwendig.

Satz 4.4
Seien die Bedingungen für Satz 4.1 erfüllt, außer bezüglich der Wahl von f_1. Sei f_1 zufällig aus einer Menge F von Funktionen gewählt. Es gebe Werte x, y und c mit $x \neq y$, so daß $f_1(x) \oplus f_1(y) = c$ mit der Wahrscheinlichkeit $p_{\bowtie}$ erfüllt ist. Dann kann A, ohne das g-Orakel aufzurufen, Eingaben (L_1, R_1) und (L_2, R_2) mit $R_1 \neq R_2$ bestimmen, für die mit mindestens der Wahrscheinlichkeit $p_{\bowtie}$ gilt:

$$R_1 \oplus R_2 = T_1 \oplus T_2.$$

T_1 und T_2 sind Teil der zu (L_1, R_1) und (L_2, R_2) gehörenden Ausgaben (S_1, T_1) und (S_2, T_2).

Beweis: A wählt $R_1 = x$, $R_2 = y$, L_1 beliebig und $L_2 = L_1 \oplus c$. Dann gilt mit der Wahrscheinlichkeit $p_{\bowtie}$

$$S_1 = L_1 \oplus f_1(R_1) = S_2 = L_2 \oplus f_1(R_2),$$

also $T_i = R_i \oplus f_2(S_i)$ und $T_j = R_j \oplus f_2(S_i)$ und deshalb $R_i \oplus R_j = T_i \oplus T_j$. □

Satz 4.4 gibt uns eine Handhabe zur Kryptoanalyse von Blockchiffren, die eine Luby-Rackoff-ähnliche 3-Runden Struktur aufweisen, vgl. Abb. 4.1. Können Angreifer effizient und mit signifikanter Wahrscheinlichkeit erfolgreich S-Kollisionen herbeiführen, ist die Chiffre unsicher gegen Chosen-Plaintext Angriffe. In Abschnitt 4.5 werden wir dies zum Angriff auf eine praktisch eingesetzte Blockchiffre nutzen.

Oberflächlich betrachtet ähneln Differenz-Konzentratoren kollisionsresistenten Hashfunktionen. Immerhin besteht für uns der Sinn des Einsatzes von Differenz-Konzentratoren darin, S-Kollisionen zu verhindern. Kollisionsresistente Hashfunktionen hängen jedoch nicht von einem geheimen Schlüssel ab, während dieser bei Differenz-Konzentratoren unverzichtbar ist. Wenn A die Funktion f_1 kennt, kann A die Werte R_1, R_2 und L_1 beliebig wählen. Mit $L_2 = f_1(R_1) \oplus f_1(R_2) \oplus L_1$ erhält A eine S-Kollision.

4.3 Eine „Abkürzung“ in der dritten Runde

Mit Bezug auf die dritte Runde, verlangen wir im Beweis von Satz 4.1, daß die Eingaben für die Zufallsfunktion f_3 paarweise verschieden sind. Im Fall $r > l$ braucht man nicht notwendigerweise eine Zufallsfunktion $f_3 : \{0,1\}^r \longrightarrow \{0,1\}^l$.

Aus den ersten beiden Runden erwarten wir q unabhängige r-Bit Zufallswerte $T_1, \ldots, T_q$ als Eingabe für f_3. Die Wahrscheinlichkeit einer T-Kollision, also der Existenz von Werten $i \neq j$ mit $T_i = T_j$, ist kleiner als $q^2/2^r$. Dies ist im Prinzip „zu gut“, denn wegen $r > l \geq n$ gilt $q^2/2^r < q^2/2^n$.

Unsere Idee ist sehr einfach. Wir ersetzen f_3 durch eine Zufallsfunktion $f_* : \{0,1\}^l \longrightarrow \{0,1\}^l$ und ignorieren die überflüssigen $r - l$ Bits des T-Wertes, etwa indem wir T als Binärzahl betrachten und $f_3(T)$ durch $f_*(T \bmod 2^l)$ ersetzen. Die Wahrscheinlichkeit, daß unter den q l-Bit Eingaben für f_* eine Kollision auftritt, beträgt $q^2/2^l \leq q^2/2^n$. Damit gilt der folgende Satz:

Satz 4.5
Sei $f_* : \{0,1\}^l \longrightarrow \{0,1\}^l$ eine Zufallsfunktion. Wenn – außer für $f_3(T) = f_*(T \bmod 2^l)$ – die Bedingungen für die Sätze 4.1 bzw. 4.2 erfüllt sind, dann gilt

$$|P_{rand} - P_{perm}| < \frac{q^2}{2^n}.$$

Selbst wenn wir $f_3(T) = f_{**}(T \bmod 2^n)$ – mit einer Zufallsfunktion $f_{**} : \{0,1\}^n \longrightarrow \{0,1\}^l$ – verwenden, ist die Ungleichung erfüllt.

4.4 Die Rundenfunktionen

Bausteine für alle mit den Methoden dieses Kapitels realisierbaren Blockchiffren sind Pseudozufallsfunktionen bzw. Differenz-Konzentratoren. Wie sollte man diese Funktionen realisieren, um zu möglichst effizienten Blockchiffren zu gelangen? Und wie effizient können die entsprechenden Blockchiffren sein?

Vorweg sei bemerkt, daß viele der im folgenden betrachteten Konstuktionen eine große Zahl geheimer Zufallsbits einsetzen. In der Regel wird man diese pseudozufällig aus einem relativ kleinen geheimen Schlüssel erzeugen.

In der ersten Runde brauchen wir keine Zufallsfunktionen, sondern können auf Differenz-Konzentratoren ausweichen. Eine Realisierung des in Satz 4.3 angegebenen Differenz-Konzentrators F benötigt rn Bits Speicherplatz. Eine naive Implementation von F würde auf einem Prozessor der Wortgröße w etwa $\frac{2nr}{w}$ Elementar-Operationen erfordern. Es ist leicht, die Rechenzeit auf Kosten eines vergrößerten Speicherbedarfs zu verringern.

Eine möglicherweise effizientere Alternative zur Realisierung von Differenz-Konzentratoren könnte auf irreduziblen CRC-Generator-Polynomen von Primzahl-Grad basieren, vgl. [52]. Man beachte, daß in diesem Fall das Polynom geheim bleiben muß.

Man kennt mutmaßliche Pseudozufallsfunktionen, die erstaunlich effizient sind – z.B. dedizierte Hashfunktionen (s.u.). Deshalb ist es unklar, ob die Verwendung einer Rundenfunktion in der ersten Runde, die zwar ein Differenz-Konzentrator ist, aber nicht pseudozufällig, zu effizienteren Blockchiffren führt als die Verwendung einer Pseudozufallsfunktion. Wir werden das in dieser Arbeit nicht weiter untersuchen, sondern uns im folgenden auf die Verwendung von Pseudozufallsfunktionen konzentrieren.

Wie wirken sich die in den vorausgegangenen Abschnitten dieses Kapitels beschriebenen Modifikationen am ursprünglichen Luby-Rackoff Schema auf den Durchsatz der Chiffren aus? Wir setzen zunächst voraus, daß wir die Konstruktion von Goldreich, Goldwasser und Micali zur Implementation von pseudozufälligen Funktionsgeneratoren aus pseudozufälligen Bit-Generatoren nutzen. Wir bezeichnen diese Funktions-Generatoren auch kurz als GGM-Generatoren. GGM-Generatoren haben wir schon in Abschnitt 2.1 beschrieben, ihre Laufzeit ist typischerweise etwa proportional zur Anzahl der Eingabebits.

Tabelle 4.1 gibt Schätzwerte für den

$$\text{Durchsatz} \approx \frac{\text{Anzahl der verschlüsselten Bits}}{\text{Anzahl der Eingabebits für GGM-Generatoren}}$$

im Vergleich mit $S_{\text{L\&R}}$, dem Durchsatz von Lubys und Rackoffs ursprünglicher Konstuktion mit $l = r$.

Die Funktion $f_1 : \{0,1\}^r \longrightarrow \{0,1\}^l$ kann als GGM-Generator oder als Differenz-Konzentrator realisiert werden. Wir kennzeichnen dies in Tabelle 4.1 durch „GGM“ bzw. „D-K“.

Für die Funktion $f_2 : \{0,1\}^l \longrightarrow \{0,1\}^r$ bieten sich keine Wahlmöglichkeiten.

Ist $l < r$, kann die dritte Rundenfunktion entweder als GGM-Generator $f_3 : \{0,1\}^r \longrightarrow \{0,1\}^l$ realisiert werden, oder, unter Ausnutzung von Satz 4.5, verkürzt als GGM-Generator $f_* : \{0,1\}^l \longrightarrow \{0,1\}^l$ (bzw. sogar $f_{**} : \{0,1\}^n \longrightarrow \{0,1\}^l$). Entsprechend schreiben wir in der Tabelle „$r \to l$“ bzw. „kurz“.

Fall	$l = r$	$l > r$	$l < r$	$l < r$	$l < r$	$l < r$
f_1	GGM	GGM	GGM	D-K	GGM	D-K
f_3	$r \to l$	$r \to l$	$r \to l$	$r \to l$	kurz	kurz
Durch-satz	$2/3$ $= S_{\text{L\&R}}$	$\frac{r+l}{2r+l}$ $> S_{\text{L\&R}}$	$\frac{r+l}{2r+l}$ $< S_{\text{L\&R}}$	$1-\text{const}$ $> S_{\text{L\&R}}$?	$\frac{r+l}{2r+l}$ $> S_{\text{L\&R}}$	$\frac{r+l}{2l}-\text{const}$ $\gg S_{\text{L\&R}}$?

Tabelle 4.1: Performance der Blockchiffre $\psi(f_1, f_2, f_3)$ bei Verwendung von GGM-Generatoren.

Realisiert man GGM-Generatoren mit Hilfe schneller dedizierter Flußchiffren wie SEAL oder PIKE, wird man wahrscheinlich über die erzielte Performance enttäuscht sein. Entweder sind bei jedem Schlüsselwechsel zeitraubende Initialisierungsmaßnahmen notwendig, wie z.B. im Fall von Rogaways and Coppersmiths SEAL [56], oder der geheime Schlüssel ist so groß, daß schon das Erzeugen eines neuen Schlüssels aufwendig ist, wie etwa im Fall von Andersons PIKE [2].

Eine praktikable Alternative zu GGM-Generatoren sind dedizierte kryptographische Hashfunktionen; denn von diesen wird ohnehin gefordert, sich wie eine Zufallsfunktion zu verhalten. Wir werden im folgenden nicht zwischen einer allgemeinen Hashfunktion $h : \{0,1\}^* \longrightarrow \{0,1\}^l$ und ihrer Einschränkung $h : \{0,1\}^i \longrightarrow \{0,1\}^l$ auf Eingaben konstanter Größe unterscheiden.

Wir betrachten Luby-Rackoff Chiffren, bei denen sich ein Block aus b Teilblöcken der Größe l zusammensetzt, d.h. $r = l(b-1)$. Wir teilen die Bit-Strings $R, T \in \{0,1\}^r$ in l-Bit Teilblöcke $R_{(1)}, \ldots R_{(b-1)}$ und $T_{(1)}, \ldots T_{(b-1)}$. Als Schlüssel verwenden wir

$$\begin{array}{lcll} K_1 & & & \in_{\mathrm{R}} \{0,1\}^r, \\ K_2 & = & (K_{2,1}, \ldots, K_{2,b-1}) & \in_{\mathrm{R}} \{0,1\}^r \quad \text{und} \\ K_3 & & & \in_{\mathrm{R}} \{0,1\}^l. \end{array}$$

Mit Hilfe von Satz 4.5 und unter Verwendung einer pseudozufälligen Hashfunktion $h : \{0,1\}^* \longrightarrow \{0,1\}^l$ können wir wie folgt verschlüsseln:

$$\begin{array}{lcll} S & = & L & \oplus \; h(K_1 \oplus R), \\ T_{(i)} & = & R_{(i)} & \oplus \; h(K_{2,i} \oplus S) \quad (1 \leq i < b) \quad \text{und} \\ U & = & S & \oplus \; h(K_3 \oplus T_{(1)}). \end{array} \tag{4.5}$$

Wenn die Funktion h pseudozufällig ist, definieren die Gleichungen $T_{(i)} = R_{(i)} \oplus h(K_{2,i} \oplus S)$ ebenfalls Pseudozufallsfunktionen $\{0,1\}^l \longrightarrow \{0,1\}^r$. In der dritten Gleichung von (4.5) können wir $T_{(1)}$ durch $T_{(i)}$ für irgendein $i \in \{1, \ldots, b-1\}$ ersetzen.

4.5 Die Verschlüsselungsfunktion von SFS

In diesem Abschnitt nutzen wir Satz 4.4 für einen Angriff auf die Verschlüsselungsfunktion von SFS. SFS ("Secure FileSystem") ist ein von Peter Gutmann [19] geschriebenes Programmpaket für die sektorweise Verschlüsselung von Daten auf Harddisks und Disketten von MS-DOS oder Windows 3.1 kompatiblen Rechnern.

Statt der zweiten Gleichung von (4.5) kann man auch mit *"cipher feedback"* (CFB) Pseudozufallsfunktionen erhalten:

$$\begin{array}{lcll} T_{(1)} & = & R_{(1)} & \oplus \; h(K_2 \oplus S) \quad \text{und} \\ T_{(i)} & = & R_{(i)} & \oplus \; h(K_2 \oplus T_{(i-1)}) \quad \text{für } 1 < i < b. \end{array} \tag{4.6}$$

Peter Gutmann spricht von der "message digest cipher", setzt CFB aber tatsächlich als zweite Runde einer Luby-Rackoff ähnlichen drei-Runden Blockchiffre ein. Mit CFB verschlüsselt man Klartexte $R = (R_{(1)}, \ldots, R_{(b-1)})$ zu Chiffretexten $T = (T_{(1)}, \ldots, T_{(b-1)})$, in Abhängigkeit von dem Schlüssel $K_2 \in_{\mathrm{R}} \{0,1\}^l$ und dem *"initial value"* (IV) S. Die Sicherheit von CFB hängt nicht von der Geheimhaltung von S ab.

Die Anpassung von CFB auf Blockgrößen, die nicht durch l teilbar sind, ist trivial. Die überflüssigen Bits von $T_{(b-1)}$ kann man einfach ignorieren.

Ein Nachteil von CFB ist seine inhärente Sequentialität. Im Falle der zweiten Gleichung von (4.5) kann man die $T_{(i)}$ parallel berechnen. Andere Nachteile von CFB sind:

- Wenn K_2 und S konstant sind, läßt eine Änderung des i-ten Klartext-Teilblöcke $R_{(i)}$ zu $R^*_{(i)}$ die ersten $i-1$ Chiffretext-Teilblöcke unverändert. $(*)$
- Schlimmer noch: Für die zu $R_{(i)}$ und $R^*_{(i)}$ korrespondierenden Chiffretext-Teilblöcke $T_{(i)}$ und $T^*_{(i)}$ gilt in diesem Fall $T_{(i)} \oplus T^*_{(i)} = R_{(i)} \oplus R^*_{(i)}$. $(**)$

Dies ist unproblematisch, wenn für je zwei verschiedene Klartexte stets (bzw. mit überwältigender Wahrscheinlichkeit) verschiedene IVs gewählt werden. Werden zwei verschiedene Klartexte unter Verwendung ein- und desselben IVs verschlüsselt, sprechen wir von einer *IV-Kollision.*

Die Blockgröße der Verschlüsselungsfunktion von SFS ist 4096 Bit, entsprechend der Sektorengröße von Disketten bzw. Harddisks bei den von SFS unterstützten Betriebssystemen. Die zweite Runde der Verschlüsselungsfunktion entspricht den Gleichungen (4.6), die dritte Runde der dritten Gleichung von (4.5). Ist es für Angreifer praktisch unmöglich, IV-Kollisionen herbeizuführen, ist die Chiffre sicher. D.h. mit einem Differenz-Konzentrator in der ersten Runde wäre die Blockchiffre sicher gegen "chosen-plaintext" Angriffe, wenn SHA eine Zufallsfunktion ist.

Im Rest dieses Abschnittes bezeichnet P_i einen 32-Bit String (ein *Wort*). Das j-te Bit des Wortes P_i ist $P_i^{(j)}$. Jedem Sektor ist ein eindeutiger *Sektor-IV* zugeordnet, ein 160-Bit Wert $(P_{-5}, \ldots, P_{-1})$. Der IV hängt von dem Sektor-IV ab. Wir können voraussetzen, daß der Sektor-IV geheim ist.

Wir können sogar voraussetzen, daß zwei verschiedene Sektoren auf einem Speichermedium stets unter Verwendung zweier verschiedener IVs verschlüsselt werden. Damit erscheint die Verschlüsselungsfunktion sicher, denn es ist praktisch unmöglich, auf einem Speichermedium IV-Kollisionen herbeizuführen.

Dies gilt, solange wir ein Speichermedium zu einem bestimmten Zeitpunkt betrachten. Welche Möglichkeiten bieten sich Angreifern, die Chiffretexte auf dem Speichermedium mehrfach zu lesen bekommen, während in der Zwischenzeit Änderungen an den Klartexten (und damit auch an den Chiffretexten) vorgenommen wurden? Der Zugang zu den Chifftetexten kann z.B. über unterschiedlich alte Backups möglich sein.

Zur Berechnung des IV aus dem Sektor-IV (P_{-5}, ..., P_{-1}) und dem Sektor-Inhalt (P_0, ..., P_{127}) benutzt SFS ein "word-wise scrambler polynomial" [19]:

$$P_i := P_i \oplus P_{i-4} \oplus P_{i-5},$$

das, in dieser Reihenfolge, für $i = 0$, $i = 1$, ..., $i = 127$, angewandt wird. Die letzen fünf Worte P_{123}, ..., P_{127} werden als IV genutzt.

Das j-te Bit $P_i^{(j)}$ von P_i hat nur Einfluß auf die fünf Bits $P_{123}^{(j)}$, ..., $P_{127}^{(j)}$ im IV. So gesehen berechnet SFS parallel 32 unabhängige Teil-Prüfsummen. Jedes Klartext-Bit beeinflußt genau eine der Teil-Prüfsummen, und jede Teilprüfsumme ist ein 5-Bit Wert.

Angreifer können leicht berechnen, wie sie eine Datei (bzw. einen Sektor-großen Teil einer Datei) modifizieren können, ohne den IV zu verändern. Wenn $i \in \{0, \ldots, 122\}$

ist, führt das Invertieren der Bits $P_i^{(j)}$, $P_{i+4}^{(j)}$ und $P_{i+5}^{(j)}$ zu keinen Veränderungen der korrespondierenden Prüfsummenbits $P_{123}^{(j)}, \ldots, P_{127}^{(j)}$ – unabhängig von Sektor-IV.

In Verbindung mit den Nachteilen (*) und (**) können Angreifer also Klartext-Änderungen mit einem „Muster" versehen, das sich in der Änderung der Chiffretexte wiederfindet.

Angreifer, die
1. Klartexte (bzw. Klartext-Änderungen) wählen können und
2. Zugriff auf die korrespondierenden Chiffretexte haben,

können damit feststellen, ob eine von ihnen gewählte Änderung vorgenommen wurde.

Ein bösartiges Programm kann damit ein SFS-verschlüsseltes Medium nutzen, um ohne Kenntnis des legitimen Nutzers Informationen zu übermitteln. Im einfachsten Fall kann es beispielsweise die Information „Programm X ist hier" übermitteln. Sei etwa X ein Programm, das eine Sicherheitslücke in Systemen zur Entdeckung und Verhinderung von Einbrüchen in den Computer öffnet. Mit Hilfe von Programm X können Angreifer in den Computer einbrechen. Dank des hier beschriebenen Angriffs können sie vorher sogar feststellen, ob ihr Einbruch gefahrlos ist.

Wir haben damit eine Schwäche in der Verschlüsselungsfunktion von SFS gefunden, die es Angreifern erlaubt, einen *Chosen-Plaintext Angriff* durchzuführen. Für die Benutzer von SFS besteht trotzdem kein Grund zur Panik. Für Angreifer genügt es nicht, nur Klartexte wählen zu können, sie müssen auch beeinflussen können, in welche Sektoren die korrespondierenden Chiffretexte geschrieben werden. Eine neue Version von SFS sollte jedoch eine neue, sicherere Verschlüsselungsfunktion einsetzen.

4.6 Pseudo-Zufälligkeit und "Sicherheit"

Um Zufallsfunktionen $\{0,1\}^a \longrightarrow \{0,1\}^b$ zu implementieren, benötigt man $b * 2^a$ Bits Speicherplatz. Sichere Blockchiffren unter Verwendung von Zufallsfunktionen sind zwar ein beeindruckendes theoretisches Konzept – insbesondere weil der Sicherheitsbeweis nicht von einer unbewiesenen Annahme abhängt –, aber für praktikable Chiffren muß man auf Pseudozufallsfunktionen ausweichen. Um den Sicherheitsbeweis zu *‚retten'*, muß man von der (natürlich unbewiesenen) Annahme ausgehen, daß die Pseudozufallsfunktionen sicher sind, also praktisch ununterscheidbar von echten Zufallsfunktionen. Der Aufwand zum Brechen der Blockchiffre und der Aufwand zum Brechen der verwendeten Pseudozufallsfunktionen sind eng miteinander verknüpft – wie eng, soll in diesem Abschnitt skizziert werden.

In der Formulierung von Satz 4.1 benutzten wir u.a. die Wahrscheinlichkeiten P_{rand} und P_{perm}. Mit P_{pseu} bezeichnen wir nun die Wahrscheinlichkeit, daß A eine 1 ausgibt, wenn das Orakel die Funktion $g = \psi(f_1, f_2, f_3)$ mit pseudozufälligen f_i berechnet. Analog zu Satz 4.1 benutzt A das Orakel höchtens q-mal. Wenn

$$|P_{\text{rand}} - P_{\text{pseu}}| \geq p + \frac{q^2}{2^n}$$

für $p > 0$ erfüllt ist, dann kann man A einfach als Test für die Zufälligkeit der Funktionen f_1, f_2 aud f_3 benutzen. Die Wahrscheinlichkeit, mit Hilfe von A zwischen zufälligen und pseudozufälligen f_i zu unterscheiden, beträgt

$$|P_{\text{perm}} - P_{\text{pseu}}| \geq p.$$

Mit anderen Worten, Angriffe auf Luby-Rackoff Chiffren sind ähnlich hart wie Angriffe auf die verwendeten Pseudozufallsfunktionen. Die Erfolgswahrscheinlichkeiten unterscheiden sich höchstens um $q^2/2^n$: *Es bezeichne t die Zeit, die erforderlich ist, um die Funktion $\psi(f_1, f_2, f_3)$ t-mal zu evaluieren. Wenn die pseudozufälligen Funktionen f_1, f_2 und f_3 (t,p)-sicher*[2] *sind, dann ist die pseudozufällige Permutation $g = \psi(f_1, f_2, f_3)$ gegen Angreifer, die ein g-Orakel genau q-mal befragen können, $(t, p + q^2/2^n)$-sicher.* Die q Zeiteinheiten zur Berechung der q Orakelfragen sind ein Bestandteil der Rechenzeit für einen Angriff.

Im Fall $q^2 \ll n$ liegt offenbar eine *verlustarme* Reduktion vor. Man beachte, daß die Anzahl aller mit einem Schlüssel verschlüsselten Klartexte eine obere Schranke für q bildet. Durch regelmäßigen Schlüsselwechsel können die Anwender unserer Blockchiffren die Bedingung $q^2 \ll n$ erzwingen. Diese Argumentation ist auch auf die im Rest dieses Kapitels beschriebenen Kryptosysteme übertragbar.

4.7 Beispiel-Chiffren

In diesem Abschnitt entwerfen wir Luby-Rackoff-Chiffren, die dedizierte Hashfunktionen als Pseudozufallsgeneratoren nutzen. Die ursprüngliche Luby-Rackoff Chiffre ist durch die Gleichungen (4.5) und den Parameter $b = 2$ zu beschreiben. Wie effizient ist ihre Realisierung mit Hilfe einer dedizierten Hashfunktion $h : \{0,1\}^* \longrightarrow \{0,1\}^l$? Wir bezeichnen den Durchsatz der ursprünglichen Luby-Rackoff Chiffre mit $S_{\text{L\&R}}$.

In jeder Runde wird h auf eine l-Bit Eingabe angewandt. Beim Verschlüsseln zweier Teilblöcke werden drei Runden durchlaufen. Man könnte naiv annehmen, daß der Durchsatz der Chiffre etwa $\frac{2}{3}$ des Durchsatzes der Hashfunktion ist, falls man den Aufwand für die XORs vernachlässigt. Die naive Annahme stimmt aber nur, wenn der Durchsatz der Hashfunktion nicht von der Eingabegröße abhängt.

Üblicherweise baut eine dedizierte Hashfunktion h auf einer internen Hashfunktion $H : \{0,1\}^\lambda \longrightarrow \{0,1\}^l$ mit konstanter Eingabelänge auf. Damit sind zum Verarbeiten eines β-Bit Wertes mindestens $\lceil \beta/\lambda \rceil$ Aufrufe von H nötig. Typische Hashfunktionen wie MD4, MD5 oder RIPE-MD verwenden z.B. eine interne Hashfunktion $H : \{0,1\}^{512} \longrightarrow \{0,1\}^{128}$. Die Anzahl der H-Aufrufe ist ein ziemlich genaues Maß für den Aufwand zur

[2] D.h., es gibt keinen Algorithmus, der in Zeit t mit mehr als der Wahrscheinlichkeit p zwischen echt zufälligen und pseudozufälligen f_1, f_2 und f_3 unterscheiden kann. Es sei an die in der Einleitung angesprochene Notation von Bellare und Rogaway [6, 7] erinnert.

Berechnung von h. Wir bezeichnen den Durchsatz von H mit S_H. Andere Hashfunktionen arbeiten prinzipiell ähnlich, im Falle des SHA und seiner Variante SHA-1 [42] werden z.B. interne Hashfunktionen mit 512 Eingabe- und 160 Ausgabebits eingesetzt.

Im folgenden entwerfen wir eine Blockchiffre mit moderat großen Blöcken (d.h. 640 Bits), basierend auf $H : \{0,1\}^{512} \longrightarrow \{0,1\}^{128}$. Um eine nur 128 Bit große Eingabe mit H zu verarbeiten, sind 384 Bits zu ergänzen. Im Hinblick auf die ursprüngliche Luby-Rackoff Chiffre sind zum Verschlüsseln eines 256 Bit großen Blocks tatsächlich $3 * 512$ Bits zu verarbeiten. Daraus folgt $S_H = 6 * S_{\text{L\&R}}$.[3]

Wie verändert sich der Durchsatz der durch (4.5) beschriebenen Chiffre, wenn wir den Wert b vergrößern? Man braucht $\lceil (b-1)/4 \rceil$ Aufrufe von H in der ersten Runde, $b-1$ in der zweiten, und einen Aufruf in der dritten – insgesamt also $b + \lceil (b-1)/4 \rceil$. Der Durchsatz ist damit proportional zu

$$\frac{b}{b + \left\lceil \frac{b-1}{4} \right\rceil} S_{\text{L\&R}}.$$

Im Fall $b = 5$ sind 6 Aufrufe der internen Hashfunktion erforderlich. Es ergeben sich Blöcke der Größe 640 Bit. Der Durchsatz unserer Variante übertrifft den der ursprünglichen Luby-Rackoff Chiffre um 25 %.

Dies ist etwas überraschend, da eine Blockchiffre mit großen Blöcken eigentlich „mehr Arbeit“ leistet als eine mit kleinen. Jedes Bit eines Chiffretext-Blocks soll schließlich auf komplexe Weise von jedem Bit des korrespondierenden Klartext-Blocks abhängen.

Der genaue Wert der Einsparung gegenüber der ursprünglichen Luby-Rackoff Chiffre hängt von den Parametern λ und l ab. Doch eine Einsparung gibt es immer, wenn das Verarbeiten von $l' > l + 1$ Bits zu einem l-Bit Hashwert mit einem höheren Durchsatz möglich ist als das Verarbeiten von $l + 1$ Bits. Wenn wir z.B. die interne Hashfunktion H' des SHA verwenden, können wir mit Hilfe der Gleichungen (4.5) und der Wahl $b = 4$ eine 640-Bit Chiffre definieren, die mit 5 Aufrufen auskommt. Die ursprüngliche Luby-Rackoff Chiffre benötigt zum Verschlüsseln eines 320-Bit Blocks 3 Aufrufe von H'. Unsere Variante ist in diesem Fall 20 % schneller.

Um die tatsächliche Geschwindigkeit unserer 640-Bit Blockchiffre abschätzen zu können, ziehen wir die Benchmark-Werte hinzu, die Roe [55] auf “Sandpiper” ermittelte, einem Rechner mit einer 133 MHz-getakteten DEC-Alpha CPU, 8 KB Primär- und 512 KB Sekundär-Cache. Die internen Hashfunktionen von MD4, MD5 bzw. RIPE-MD verarbeiten 512-Bit Blöcke zu 128 Ausgabebits. Mit $b = 5$ erhalten wir jeweils eine 640-Bit Blockchiffre. Die Verschlüsselung eines Blocks erfordert $b + 1 = 6$ Aufrufe der internen Hashfunktion, die ebenso dazu dienen könnten, $6 * 512$ Eingabebits zu einem Hash-Wert zu verarbeiten. Folglich ist die jeweilige Hashfunktion etwa um den Faktor $3072/640 = 4,8$ schneller als die korrespondierende 640 Blockchiffre entsprechend (4.5). SHA-1 erzeugt

[3] Die interne Hashfunktion von SHA (bzw SHA-1) verarbeitet 512 Eingabe- zu 160 Ausgabebits. Sei $H' : \{0,1\}^{512} \longrightarrow \{0,1\}^{160}$. Unter Verwendung von H' kommen wir zu einer 320-Bit Blockchiffre, zum Verschlüsseln mit ihr sind drei Aufrufe von H' notwendig, folglich $S_{H'} = 4,8 * S_{\text{L\&R}}$

einen 160-Bit Hashwert und ist um den Faktor 4 schneller als die korrespondierende 640-Bit Blockchiffre. Wir vergleichen unsere Chiffren mit Blowfish [57], der schnellsten von Roe untersuchten gewöhnlichen Blockchiffre, siehe Tabelle 4.2.

Tabelle 4.2: Vergleich der Durchsätze (in M Bit/Sekunde)

	MD4	MD5	RIPE-MD	SHA-1	Blowfish
Durchsatz der Hashfunktion [55]	78,77	60,02	48,00	41,51	—
Wert b für 640-Bit Blockchiffre nach (4.5)	5	5	5	4	—
Durchsatz der Blockchiffre	16,4	12,5	10,0	10,4	11,63

Am Ende dieses Abschnittes sei noch eine **Warnung** ausgesprochen! Die Hashfunktion MD4 wurde 1996 von Dobbertin [13] gebrochen. Dobbertin fand eine effiziente Methode, Kollisionen zu berechnen. Wegen der ähnlichen Konstruktionen steht auch die Sicherheit von RIPE-MD und MD5 in Zweifel. MD4, MD5 oder RIPE-MD können trotzdem pseudozufällig sein. Dobbertin selbst betrachtet die Pseudozufälligkeit von MD4 nicht, doch meldet er keine Bedenken gegen den Einsatz von MD4 als Einweg-Hashfunktion an: *"Where MD4 is still in use, it should be replaced! An exception is the application of MD4 as a one-way function."* ([13], S. 67, Hervorhebung im Original!) Für die Benutzer von Krypto-Anwendungen, die Luby-Rackoff Chiffren beispielsweise mit MD4 als Zufallsfunktion einsetzen, besteht kein Grund zur Panik. Andererseits sollte man sich beim Entwurf *neuer* Anwendungsprogramme auf Komponenten beschränken, die bezüglich ihres *primären Entwurfszieles* als sicher gelten. Die Entscheidung für eine Komponente sollte von Experten bzw. -innen getroffen werden, die den aktuellen Stand der (öffentlichen) Forschung genau kennen.[4]

4.8 Die Blockchiffren BEAR und LION

Der Ausgangspunkt für die in diesem Kapitel beschriebenen Blockchiffren war die Idee, daß Luby-Rackoff Chiffren unbalanciert sein können, daß also in Abbildung 4.1 $l \neq r$ gelten kann. Diese Möglichkeit war bis dahin in der Literatur vernachlässigt worden. Unabhängig von Lucks [33] präsentierten Anderson und Biham [4] BEAR und LION, zwei ebenfalls unbalancierte Luby-Rackoff Chiffren.

Es bezeichne $H : \{0,1\}^* \longrightarrow \{0,1\}^l$ eine Hashfunktion. Auf geeignete Weise verbindet man H mit einem geheimen k-Bit Schlüssel K zu einer Funktion $H^* : \{0,1\}^k \times \{0,1\}^* \longrightarrow \{0,1\}^l$ und erhält einen MAC $(\{0,1\}^l, \{0,1\}^k, H')$. Es bezeichne $\Sigma\{0,1\}^l \longrightarrow \{0,1\}^*$ eine Flußchiffre, die sogar eine Pseudozufallsfunktion ist.

[4] Zur Zeit der Abfassung dieser Arbeit könnte man – neben dem schon erwähnten "secure hash algorithm" SHA [40] und seiner Variante SHA-1 [42] – die folgenden dedizierten Hashfunktionen als pseudozufällige Funktionen in Betracht ziehen: HAVAL [58] und die Nachfolger [14] von RIPE-MD, nämlich RIPE-MD-128 und RIPE-MD-160. Das Konzept von TIGER [3] ist noch zu neu, als daß man diese Hashfunktion heute schon in Betracht ziehen sollte. Alle erwähnten Hashfunktionen verwenden, wie MD4, MD5 und RIPE-MD, eine interne Hashfunktion mit fester Eingabelänge $\lambda > l$.

Die Blockchiffre BEAR berechnet den Chiffretext $(U,T) \in \{0,1\}^l \times \{0,1\}^r$ aus dem Klartext $(L,R) \in \{0,1\}^l \times \{0,1\}^r$ in Abhängigkeit von zwei geheimen Rundenschlüsseln K_1 und K_3:

$$\begin{array}{rcl} S & = & L \oplus H^*_{K_1}(R), \\ T & = & S \oplus \Sigma(S) \quad \text{und} \\ U & = & T \oplus H^*_{K_3}(T). \end{array} \tag{4.7}$$

Entsprechend gelten im Fall von LION:

$$\begin{array}{rcl} S & = & L \oplus \Sigma(R \oplus K_1), \\ T & = & S \oplus H(S) \quad \text{und} \\ U & = & T \oplus \Sigma(R \oplus K_3). \end{array}$$

Für BEAR und LION gilt: $H = \text{SHA-1}$ [42] und $\Sigma = \text{SEAL}$ [56]. SEAL gilt zwar als Flußchiffre, doch weisen seine Autoren ausdrücklich darauf hin, daß SEAL mit dem Ziel entworfen wurde, sogar pseudozufällig zu sein. Damit ergibt sich $l = 160$, denn SHA-1 liefert 160 Ausgabe-Bits und SEAL benötigt 160 Eingabe-Bits.

In Abschnitt 4.7 haben wir bereits die Benchmarks von Roe [55] zitiert. Seine "Sandpiper" erreichte rund 40 Mbit/sec für die Hashfunktion SHA-1 und etwa 115 Mbit/sec für die Flußchiffre SEAL. Das Resultat für SEAL ist freilich fast schon als asymptotisch anzusehen, denn Roe berücksichtigte die sehr langsame Initialisierung von SEAL nicht. Diese ist beim Start von SEAL und nach jedem Schlüsselwechsel erforderlich. Sie dauert ungefähr so lange wie das Verarbeiten einer 102400 Bit großen Eingabe[5] mit SHA-1.

Für sehr große Blöcke fällt die Dauer der Initialisierungsphase nicht sehr ins Gewicht. Dann ist LION etwas schneller als BEAR, aber beide erreichen einen bemerkenswert hohen Durchsatz. Auf der schon mehrfach erwähnten DEC "Sandpiper" von Roe maßen Anderson und Biham 13,62 MBit/sec für BEAR und 18,68 MBit/sec für Lion, bei einer Blockgröße von 1 MBit. Zum Vergleich: Für Blowfish wurden, wie schon erwähnt, 11,63 MBit/sec gemessen.

Anderson und Biham skizzierten auch einen Sicherheitsbeweis für BEAR und LION. Allerdings beschränkten sie sich auf Known-Plaintext Angriffe, bei denen für die Angreifer nur ein einziges Klartext/Chiffretext Paar bekannt wird. Sie zeigten sogar nur, daß eine "key recovery" praktisch unmöglich ist. Damit wird der Versuch der Angreifer beschrieben, den geheimen Schlüssel herauszufinden.

Andersons und Bihams Angriffsmodell ist so schwach, daß man es getrost als *kryptograhpisch irrelevant* bezeichnen kann.[6] Bezogen auf solche Angriffe ist sogar die triviale Blockchiffre $E(K,X) = X$ sicher, denn aus dem Klartext/Chiffretext Paar (X,X) kann man nicht auf den geheimen Schlüssel K schließen. Interessanterweise ist die Vernam-Chiffre *unsicher* gegen die von Anderson und Biham untersuchten Angriffe.

[5] Schneier gibt in Kap. 17.2 seines Buchs [58] an, daß der Aufwand für die Initialisierung von SEAL dem Aufwand von etwa 200 SHA-1-Aufrufen entspricht.

[6] Diese Kritik beschränkt sich auf die formale Behandlung ihres Sicherheitsbeweises. Andere Aspekte des Beitrages von Anderson und Biham sind neu und von praktischer Relevanz.

Mit den in diesem Kapitel verwendeten Methoden kann man einen kryptographisch relevanten Sicherheitsbeweis für BEAR und LION führen, die Sicherheit von SHA-1 und SEAL natürlich vorausgesetzt. Wir beschränken uns hier auf BEAR.

Satz 4.6
Sei $H^* : \{0,1\}^k \times \{0,1\}^* \longrightarrow \{0,1\}^l$ *ein pseudozufälliger Funktionsgenerator. Seien die Funktion* $\Sigma\{0,1\}^l \longrightarrow \{0,1\}^*$ *pseudozufällig, die Schlüssel* $K_1, K_3 \in_R \{0,1\}^k$ *geheim und* $BEAR_{K_1,K_3}$ *durch die Gleichungen (4.7) definiert. Sei* $g : \{0,1\}^{r+l} \longrightarrow \{0,1\}^{r+l}$ *mit der Wahrscheinlichkeit* $\frac{1}{2}$ *zufällig gewählt. Andernfalls sei* $g(L,R) = BEAR_{K_1,K_3}(L,R)$.

Für einen Polynomialzeitalgorithmus, der q *Fragen* $(L_1,R_1),\ldots,(L_q,R_q)$ *an* g *stellt und die korrespondierenden Antworten* $(U_1,T_1),\ldots,(U_q,T_q)$ *mit* $(U_i,T_i) = g(L_i,R_i)$ *erhält, ist es praktisch unmöglich, mit einer Wahrscheinlichkeit signifikant über* $\frac{1}{2}+q^2/2^n$ *richtig zu entscheiden, ob* g *zufällig gewählt wurde.*

Beweis: Man beachte, daß die Rundenfunktion in der zweiten Runde von BEAR nicht von einem geheimen Schlüssel abhängt.

Sei $K_2 \in_R \{0,1\}^l$. Die Funktion H^* ist genau dann ein pseudozufälliger Funktionsgenerator, wenn $H^{*K_2} : (\{0,1\}^l \times \{0,1\}^k) \times \{0,1\}^* \longrightarrow \{0,1\}^l$ mit $H^{*K_2}(K,X) = H^*(K,X) \oplus K_2$ es ist. In diesem Fall ist für $K_1 \in_R \{0,1\}^k$ die Funktion $H^{*K_1}_{K_1}(X) = H^{*K_2}(K_1,X)$ eine Pseudozufallsfunktion. Ebenso ist Σ genau dann pseudozufällig, wenn $\Sigma^*(K,X) = \Sigma(K \oplus X)$ es ist.Wir definieren $\mathrm{BEAR}^*_{K_1,K_2,K_3}$ durch

$$\begin{array}{rcrcll} S & = & L & \oplus & H^{*K_2}_{K_1}(R), & \\ T & = & S & \oplus & \Sigma(K_2 \oplus S) & \text{und} \\ U & = & T & \oplus & H^*_{K_3}(T). & \end{array}$$

Es ist $\mathrm{BEAR}^*_{K_1,K_2,K_3} = \mathrm{BEAR}_{K_1,K_3}$. Der Sicherheitsbeweis für $\mathrm{BEAR}^*_{K_1,K_2,K_3}$ verläuft analog zum Beweis von Satz 4.1. □

Dem aufmerksamen Leser bzw. der aufmerksamen Leserin mag aufgefallen sein, daß wir bei der Formulierung von Satz 4.1 die Schranke (4.1) verwendeten: $|P_{\mathrm{rand}} - P_{\mathrm{perm}}| < q^2/2^n$. Dabei bezeichnen P_{rand} und P_{perm} bedingte Wahrscheinlichkeiten und q die Anzahl der Orakel-Fragen. Im Fall von Satz 4.6 haben wir hingegen keine bedingten Wahrscheinlichkeiten betrachtet. Statt dessen haben wir vorausgesetzt, daß die betrachtete Permutation mit genau der Wahrscheinlichkeit $\frac{1}{2}$ zufällig gewählt wurde. Wir erhielten $\frac{1}{2}+q^2/2^n$ als Schranke für die Erfolgswahrscheinlichkeit. Intuitiv ist klar, daß beide Formulierungen den gleichen Sachverhalt beschreiben: *Der Vorteil, den q Orakelfragen bei der Unterscheidung zwischen einer zufällig gewählten Permutation und der untersuchten Konstruktion bringen, ist durch* $q^2/2^n$ *nach oben beschränkt.*

4.9 Die Blockchiffre BEAST

Wie Lucks [34] beobachtete, kann man mit Hilfe der „Abkürzung“ in der dritten Runde (siehe Satz 4.5) eine sichere Blockchiffre definieren, deren Durchsatz sogar den von LION

übertrifft. Wir spechen vom BEAST, dem *"Block Encryption Algorithm with Shortcut in the Third round"*. Die geheimen Schlüssel sind: $K_1, K_3 \in_R \{0,1\}^r$ und $K_2 \in_R \{0,1\}^l$. Diese Schlüssel kann man pseudozufällig mit Hilfe eines kürzeren „Meister-Schlüssels" K_M und der Funktion Σ erzeugen. Wir schreiben

$$\Sigma_K^*(x) = \Sigma(K \oplus x).$$

BEAST kann man mit den folgenden Gleichungen beschreiben:

$$\begin{aligned} S &= L \oplus H_{K_1}^*(R), \\ T &= R \oplus \Sigma_{K_2}^*(S), \quad \text{und} \\ U &= S \oplus H_{K_3}^*(T^*) \quad \text{mit } T^* = \text{„}l \text{ Bits von } T\text{"}, \text{ z.B. } T^* = T \bmod 2^l. \end{aligned}$$

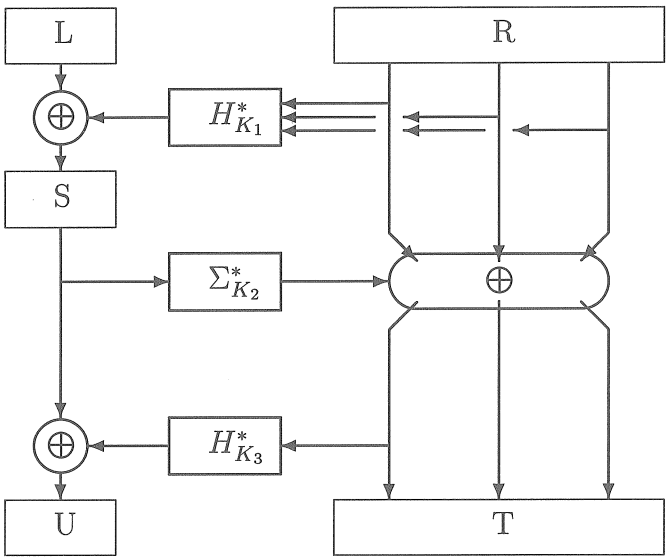

Abbildung 4.2: Die Blockchiffre BEAST.

Analog zu Satz 4.5 gilt:

Satz 4.7
Seien $H_{K_1}^* : \{0,1\}^r \longrightarrow \{0,1\}^l$, $\Sigma_{K_2}^* : \{0,1\}^l \longrightarrow \{0,1\}^r$ *und* $H_{K_3}^* : \{0,1\}^l \longrightarrow \{0,1\}^l$ *pseudozufällige Funktionen. Sei* $g : \{0,1\}^{r+l} \longrightarrow \{0,1\}^{r+l}$ *mit der Wahrscheinlichkeit* $\frac{1}{2}$ *zufällig gewählt. Andernfalls sei* $g(L,R) = BEAST(L,R)$.

Für einen Polynomialzeitalgorithmus, der q *Fragen* $(L_1,R_1),\ldots,(L_q,R_q)$ *an* g *stellt und die korrespondierenden Antworten* $(U_1,T_1),\ldots,(U_q,T_q)$ *mit* $(U_i,T_i) = g(L_i,R_i)$ *erhält, ist es praktisch unmöglich, mit einer Wahrscheinlichkeit signifikant über* $\frac{1}{2}+q^2/2^n$ *richtig zu entscheiden, ob* g *zufällig gewählt wurde.*

Im letzten Abschnitt sind wir auf die Geschwindigkeiten von BEAR und LION eingegangen, bei einer Blockgröße von 1 MBit und auf einer "Sandpiper". BEAR erreichte

13,62 MBit/sec, LION 18,68 MBit/sec. Wir wollen abschätzen, wie effizient BEAST unter den gleichen Umständen ist.

Sei $b = 1024^2 - 160$. Die Geschwindigkeit von BEAR wird dominiert durch das zweifache Anwenden der Hashfunktion SHA-1 auf b Bit in der ersten und dritten Runde sowie das Verschlüsseln von b Bit mit der Flußchiffre SEAL in der zweiten Runde. Entsprechend wird die Geschwindigkeit von LION dominiert durch zweifaches Anwenden von SEAL und das einfache Anwenden von SHA-1 auf jeweils b Bit.

Bei einer Blockgröße von 1 MBit kann man für BEAST den Aufwand in der dritten Runde vernachlässigen. Die Geschwindigkeit bestimmen eine Anwendung von SHA-1 und eine Anwendung von SEAL auf jeweils b Bit. Als Geschwindigkeit von BEAST errechnen wir damit rund 23,6 MBit/sec. Auf einer "Sandpiper" kann man mit BEAST also mehr als doppelt so schnell verschlüsseln wie mit Blowfish.

4.10 "Remote-key"-Verschlüsselung mit BEAST-RK

Beim Verschlüsseln strebt man oft an, den Schlüssel in einem vertrauenswürdigen Hardwaremodul zu halten, im folgenden als "Smartcard" bezeichnet. Um Sicherheit sogar dann zu bieten, wenn Angreifer die Smartcard physikalisch in ihre Hände bringen, sie ggf. analysieren und zerlegen können, muß die Smartcard *manipulationsgeschützt* (engl. "tamperproof") sein. Wegen des Hardware-Aufwandes für den Manipulationsschutz sind Smartcards vergleichsweise langsame Rechner.

In diesem Abschnitt beschreiben wir eine Chiffre, mit der man unter bestimmten Umständen paradoxerweise mit einer langsamen Smartcard einen hohen Durchsatz beim Verschlüsseln erreichen kann. Dabei übernimmt ein schneller Rechner, der "Host", einen Teil der Verschlüsselungsaufgaben, kennt aber den Schlüssel nicht. Die Idee zu einer solchen Aufteilung stammt von Blaze [8], siehe auch den letzten Absatz dieses Abschnittes.

Sei $r > l$. Seien K und K' zufällig erzeugte Schlüssel. Seien die Funktionen H, H^*, Σ und Σ^* pseudozufällig und wie in den letzten beiden Abschnitten definiert. Dann sind auch die Funktionen

$$\begin{aligned} F_K(X) &= H^*_K(H(X)) \quad \text{und} \\ G_{K'}(X) &= \Sigma(H^*_{K'}(X)) \end{aligned}$$

pseudozufällig. Wir können in Abbildung 4.2 daher H^* durch F und Σ^* durch G ersetzen – offensichtlich ohne daß die Blockchiffre dadurch unsicher wird. Die pseudozufälligen Funktionsgeneratoren F und G bestehen aus insgesamt vier Teilfunktionen:

1. Eine schlüsselunabhängige Kompressionsfunktion $H : \{0,1\}^{r-l} \longrightarrow \{0,1\}^l$,
2.,3. zwei Zufallsfunktionsgeneratoren $H^* : \{0,1\}^k \times \{0,1\}^l \longrightarrow \{0,1\}^l$ und $H^{**} : \{0,1\}^k \times \{0,1\}^{2l} \longrightarrow \{0,1\}^l$ (dabei ist k die Schlüssellänge) und
4. ein schlüsselunabhängiger Zufallsbitgenerator $\Sigma : \{0,1\}^l \longrightarrow \{0,1\}^{r-l}$.

Die schlüsselabhängigen Operationen verarbeiten eine l- bzw. $2l$-Bit Eingabe zu einer l-Bit Ausgabe. Im Fall $r \gg l$ kann es sich lohnen, die schlüsselunabhängigen Operationen auf dem schnellen Host durchzuführen, dem wir jedoch nicht den Schlüssel anvertrauen wollen. Die schlüsselabhängigen Operationen muß die Smartcard durchführen. Diese Überlegung führt uns zu der Blockchiffre BEAST-RK, siehe Abbildung 4.3.

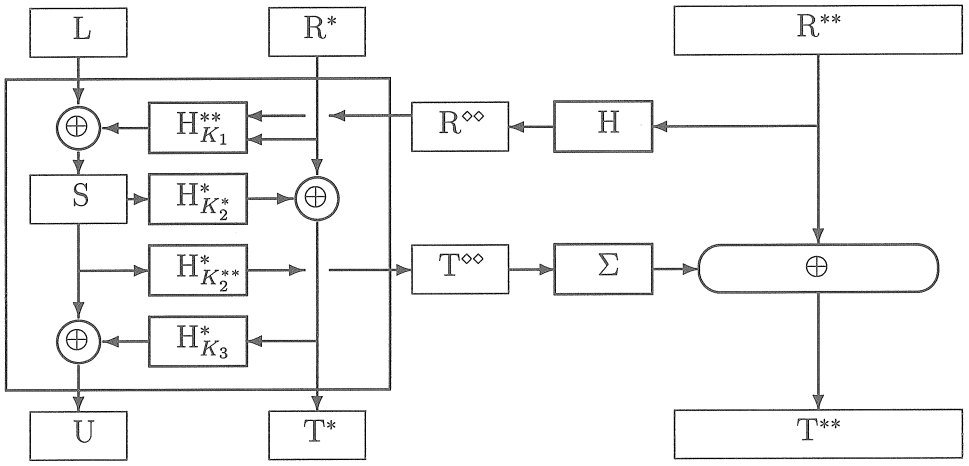

Abbildung 4.3: Die "remote-key" Variante BEAST-RK von BEAST.

Die rechte Seite R des Klartextes wird aufgeteilt in einen l-Bit Teilblock R^* und einen $(r-l)$-Bit Teilblock R^{**}. Entsprechend ist auch die Aufspaltung des Rundenschlüssels K_2 für die zweite Runde in K_2^* und K_2^{**}, sowie die Aufspaltung der rechten Seite T des Chiffretextes in T^* und T^{**}. BEAST-RK kann durch die folgenden Gleichungen[7] beschrieben werden:

$$\begin{aligned} S &= L \oplus H^{**}_{K_1}(R^* || H(R^{**})), \\ T^* &= R^* \oplus H^*_{K_2^*}(S), \\ T^{**} &= R^{**} \oplus \Sigma(H^*_{K_2^{**}}(S)) \quad \text{und} \\ U &= S \oplus H^*_{K_3}(T^*). \end{aligned}$$

Das große Rechteck in Abbildung 4.3, das u.a. alle schlüsselabhängigen Operationen umschließt, deutet die Smartcard an. Pfeile innerhalb des Rechtecks stellen den lokalen Informationsfluß der Smartcard dar, Pfeile in das bzw. aus dem Rechteck die Kommunikation der Smartcard mit dem Host. Die restlichen Pfeile repräsentieren den lokalen Informationsfluß des Hosts. Um die Schnittstelle zwischen Host und Smartcard genauer beschreiben zu können, brauchen wir Bezeichner für zwei zusätzliche Zwischenwerte:

$$\begin{aligned} R^{\diamond\diamond} &= H(R^{**}) \quad \text{und} \\ T^{\diamond\diamond} &= H^*_{K_2^{**}}(S). \end{aligned}$$

[7]Dies sind zwar vier Gleichungen, doch fassen wir die zweite und dritte zusammen als Repräsentation der zweiten Runde auf, also BEAST-RK als Luby-Rackoff Chiffre mit drei Runden.

Wir schreiben BEAST-RK-local$(L, R^*, R^{\diamond\diamond}) = (S, T^*, T^{\diamond\diamond})$.

Um die Verschlüsselungsfunktion zwischen dem Host und der Smartcard zu verteilen, wird das folgende Protokoll benutzt:

1. Gegeben L, R^* und R^{**}, berechnet der Host $R^{\diamond\diamond}$ und sendet $(L, R^*, R^{\diamond\diamond})$ an die Smartcard.

2. Diese berechnet S und $(U, T^*, T^{\diamond\diamond})$ = BEAST-RK-local$(L, R^*, R^{\diamond\diamond})$ und sendet $(U, T^*, T^{\diamond\diamond})$ an den Host. S bleibt geheim.

3. Der Host berechnet $T^{**} = \Sigma(T^{\diamond\diamond})$.

Wie ist es um die Sicherheit von BEAST-RK bestellt? Wenn die Funktionen H, Σ und H^* pseudozufällig sind, ist BEAST-RK sicher gegen Chosen-Plaintext Angriffe – siehe Satz 4.5. Um Sicherheit bei "remote-key"-Verschlüsselung zu beweisen, brauchen wir zuerst ein Modell für Angriffe auf Blockchiffren durch einen vertrauensunwürdigen Host, der einen Teil der Verschlüsselungsarbeit übernimmt. Die Besonderheit bei "remote-key"-Verschlüsselung besteht also darin, daß sich ein zusätzliches Orakel in den Händen der Angreifer befindet.

Sei $g : \{0,1\}^b \longrightarrow \{0,1\}^b$ eine Permutation. Seien $h_1 : \{0,1\}^B \longrightarrow \{0,1\}^b$ und $h_2 : \{0,1\}^b \times \{0,1\}^B \longrightarrow \{0,1\}^B$ Funktionen. Die Permutation $G : \{0,1\}^B \longrightarrow \{0,1\}^B$ sei definiert durch $G(X) = h_2(g(h_1(X)), X)$. Angreifer kennen die Funktionen h_1 und h_2, können ein g-Orakel aufrufen und ebenso ein G'-Orakel. Das G'-Orakel berechnet entweder die Permutation G oder eine Zufallspermutation. Die Aufgabe der Angreifer besteht darin, herauszufinden, ob das G'-Orakel die Permutation G berechnet oder nicht.

Beispiele: Für $b = B$ ist jede Verschlüsselungsfunktion $E : \{0,1\}^b \longrightarrow \{0,1\}^b$ ein Trivialbeispiel: Wir setzen $h_1(X) = X$, $h_2(Y, X) = Y$ und $g(X) = G(X) = E(X)$. Für ein weiteres Beispiel betrachten wir die Funktion $g : \{0,1\}^{3l} \longrightarrow \{0,1\}^{3l}$ mit

$$\begin{array}{lcll} \text{in} & = & (L, R^*, R^{**}), & \\ g(L, R^*, R^{\diamond\diamond}) & = & (U, T^*, T^{\diamond\diamond}), & \\ h_1(L, R^*, R^{**}) & = & (L, R^*, R^{\diamond\diamond}), & \\ h_2(U, R^*, R^{\diamond\diamond}, \text{in}) & = & (U, T^*, T^{**}) \quad \text{und} & \\ G(L, R^*, R^{**}) & = & h_2(g(h_1(L, R^*, R^{**})), \text{in}) & \\ & = & (U, T^*, T^{**}). & \end{array} \tag{4.8}$$

In diesem Beispiel ist G gleich BEAST-RK, und die Funktion g repräsentiert den schlüsselabhängigen Teil BEAST-RK-local von BEAST-RK.

In keinem Fall können wir Angreifer daran hindern, einen Klartext X zu verschlüsseln, wenn ihnen der Wert $g(h_1(X))$ bekannt ist oder wird. Daher beschränken wir unsere Betrachtung auf Mengen von Klartexten, die paarweise „nicht-äquivalent" sind. Wir bezeichnen zwei Anfragen $x \in \{0,1\}^b$ für das g-Orakel und $X \in \{0,1\}^B$ für das G'-Orakel als *äquivalent*, wenn $x = h_1(X)$ erfüllt ist. Wir bezeichnen zwei Anfragen $X, Y \in$

$\{0,1\}^B$ für das G-Orakel als *äquivalent*, wenn $h_1(X)$ und Y es sind. Wir bezeichnen zwei Anfragen $x, y \in \{0,1\}^b$ für das g-Orakel als *äquivalent*, wenn sie gleich sind.

Wegen der Beschränkung unseres Modells auf Angriffe, bei denen die Klartexte paarweise nicht-äquivalent sind, wäre unser Modell sehr schwach, wenn Angreifern zwei äquivalente Klartexte $X, Y \in \{0,1\}^B$ mit $X \neq Y$ finden könnten. Deshalb fordern wir, daß die Funktion $h_1 : \{0,1\}^B \longrightarrow \{0,1\}^b$ kollisionsresistent[8] sein muß.

Wir betrachten die Permutation G aus dem Gleichungssystem (4.8). *Sicherheit gegen Chosen-Plaintext Angriffe mit zusätzlichem g-Orakel* erfordert, daß die Funktion h_1 kollisionsresistent ist und die Permutation G nicht von einer Zufallsfunktion zu unterscheiden ist – für Angreifer, die h_1 und h_2 kennen und auf ein g-Orakel zugreifen können. Der folgende Satz zeigt, daß G in diesem Sinne sicher ist.

Satz 4.8
Seien die Funktionen $H^*_{K_2^*}, H^*_{K_2^{**}}, H^*_{K_3} : \{0,1\}^l \longrightarrow \{0,1\}^l$ und $H^*_{K_1} : \{0,1\}^{2l} \longrightarrow \{0,1\}^l$ pseudozufällig. Seien die Funktion $H : \{0,1\}^{r-l} \longrightarrow \{0,1\}^l$ kollisionsresistent und $\Sigma : \{0,1\}^l \longrightarrow \{0,1\}^{r-l}$ ein pseudozufälliger Bitgenerator. Sei $g : \{0,1\}^{3l} \longrightarrow \{0,1\}^{3l}$ die Funktion BEAST-RK-local. Sei $G : \{0,1\}^{l+r} \longrightarrow \{0,1\}^{l+r}$ mit der Wahrscheinlichkeit $\frac{1}{2}$ zufällig gewählt. Andernfalls sei $G(L, R^*, R^{**}) = BEAST\text{-}RK(L, R^*, R^{**})$.

Wir betrachten einen Polynomialzeitalgorithmus A, der q Fragen $Q_1 \ldots Q_q$ stellt. Ist die Frage Q_i von der Form $Q_i = (L_i, R_i^*, R_i^{**}) \in \{0,1\}^{l+r}$, dann ist die zugehörige Antwort $G(L_i, R_i^*, R_i^{**})$. Ist die Frage Q_i von der Form $Q_i = (L_i, R_i^*, R_i^{\diamond\diamond}) \in \{0,1\}^{3l}$, dann ist die Antwort $g(L_i, R_i^*, R_i^{\diamond\diamond})$. Andere Fragen sind nicht möglich. Für alle $i, j \in \{1, \ldots q\}$ mit $i \neq j$ sind Q_i und Q_j nicht äquivalent. Zu jeder Frage erhält A die zugehörige Antwort.

Dann ist es für A praktisch unmöglich, mit einer Wahrscheinlichkeit signifikant über $\frac{1}{2} + q^2/2^n$ richtig zu entscheiden, ob G zufällig gewählt wurde oder gleich BEAST-RK ist.

Beweis: Sei G gleich BEAST-RK. Analog zum Beweis von Satz 4.1 zeigen wir, daß mit hoher Wahrscheinlichkeit alle Ausgabewerte voneinander unabhängige Pseudozufallswerte sind. Aus Satz 4.5 folgt die Hilfsaussage:

> **Hilfssatz 4.10.1** *Seien die Funktionen $H^*_{K_2^*}, H^*_{K_2^{**}}, H^*_{K_3} : \{0,1\}^l \longrightarrow \{0,1\}^l$ und $H^*_{K_1} : \{0,1\}^{2l} \longrightarrow \{0,1\}^l$ pseudozufällig. Sei $g : \{0,1\}^{3l} \longrightarrow \{0,1\}^{3l}$ mit der Wahrscheinlichkeit $\frac{1}{2}$ zufällig, sonst $g(L, R^*, R^{**}) = BEAST\text{-}RK\text{-}local(L, R^*, R^{**})$.*
>
> *Für einen Polynomialzeitalgorithmus, der q Fragen $(L_1, R_1^*, R_1^{\diamond\diamond}), \ldots, (L_q, R_q^*, R_q^{\diamond\diamond})$ an g stellt und die Antworten $(U_1, T_1^*, T_1^{\diamond\diamond}), \ldots, (U_q, T_q^*, T_q^{\diamond\diamond})$ mit $(L_i, R_i^* R_i^{\diamond\diamond}) = g(L_i, R_i^*, R_i^{\diamond\diamond})$ erhält, ist es praktisch unmöglich, mit einer Wahrscheinlichkeit signifikant über $\frac{1}{2} + q^2/2^n$ richtig zu entscheiden, ob g zufällig gewählt wurde.*

[8] Kollisionsresistenz bedeutet, es ist praktisch unmöglich, zwei Werte $x \neq y$ zu finden mit $x, y \in \{0,1\}^B$ und $h_1(x) = h_1(y)$. Formal haben wir Kollisionsresistenz nur für Hashfunktionen definiert, also für den Fall $b < B$. Für $b \geq B$ ist es trivial, kollisionsresistente Funktionen zu finden.

Wären die $(U_i, T_i^*, T_i^{\diamond\diamond})$ Tripel von Zufallswerten, dann stünde A vor dem Problem, maximal q Bit-Strings $\Sigma(T_i^{\diamond\diamond})$ von Zufallswerten unterscheiden zu müssen, wobei A die Zufallswerte $T_i^{\diamond\diamond}$ nicht kennen würde. Nach Voraussetzung ist Σ ein pseudozufälliger Bitgenerator, deshalb ist diese Aufgabe für A praktisch unmöglich.

Zu der i-ten Orakel-Frage existiert ein Tripelpaar $((L_i, R_i^*, R_i^{\diamond\diamond}), (U_i, T_i^*, T_i^{\diamond\diamond}))$. Entweder ist die Frage von der Form $(L_i, R_i^*, R_i^{\diamond\diamond})$, dann ist $(U_i, T_i^*, T_i^{\diamond\diamond})$ die Antwort. Oder die Frage ist von der Form (L_i, R_i^*, R_i^{**}), dann bleibt der Wert $R_i^{\diamond\diamond}$ geheim; A erfährt nur U, T^*, und $T_i^{**} = \Sigma(T_i^{\diamond\diamond})$. Insgesamt existieren q Paare $((L_1, R_1^*, R_1^{\diamond\diamond}), (U_1, T_1^*, T_1^{\diamond\diamond}))$ $\ldots$, $((L_q, R_q^*, R_q^{\diamond\diamond}), (U_Q, T_q^*, T_q^{\diamond\diamond}))$. Hilfssatz 4.10.1 zufolge ist es praktisch unmöglich, die Tripel $(U_1, T_1^*, T_1^{\diamond\diamond}), \ldots, (U_q, T_q^*, T_q^{\diamond\diamond})$ mit einer Wahrscheinlichkeit signifikant über $\frac{1}{2} + q^2/2^n$ von Zufallstripeln zu unterscheiden.

Würde A eine Erfolgsquote siginifkant über $\frac{1}{2} + q^2/2^n$ erreichen, dann wäre die Funktion $\Sigma : \{0,1\}^l \longrightarrow \{0,1\}^{r-l}$ kein pseudozufälliger Bitgenerator – im Widerspruch zu den Voraussetzungen. □

Praktisch einsetzbar ist BEAST-RK z.B. im Bereich des Pay-TV und ähnlicher Anwendungen, wenn ein Anbieter verschlüsselte Daten übermittelt. Entschlüsseln sollen die Kunden mit ihren eigenen Decodern. Diese Decoder können ggf. sogar als Software auf PCs realisiert werden. Entschlüsseln darf aber nur für zahlende Kunden möglich sein, die eine manipulationsgeschützte Smartcard gegen Geld vom Anbieter bezogen haben.[9] Kunden müssen *ent*schlüsseln können, nicht notwendigerweise jedoch *ver*schlüsseln. Mit anderen Worten, es ist Sicherheit gegen Chosen-Ciphertext Angriffe gefordert, nicht gegen Chosen-Plaintext Angriffe (CPA). Wir haben aber lediglich bewiesen, daß BEAST-RK sicher gegen CPA ist. Deshalb verschlüsselt der Anbieter mit BEAST-RK^{-1}, der Umkehrung von BEAST-RK. Die Kunden entschlüsseln entsprechend mit BEAST-RK, und die Smartcards sind so konstruiert, daß sie bei der Berechnung von BEAST-RK-local^{-1} keine Hilfe sind.

Die Kunden können ihren Decoder und die Smartcard sogar dazu benutzen, Chiffretexte an den Anbieter zu verschicken. Dann dient BEAST-RK als *Ver*schlüsselungsfunktion. Nur der Anbieter, der BEAST-RK^{-1} anwenden kann, kann *ent*schlüsseln. Damit simulieren wir die Funktionalität eines Public-Key Kryptosystems. Freilich sind weder BEAST noch BEAST-RK Public-Key Chiffren, denn die Public-Key Eigenschaft hängt von dem Manipulationsschutz der verwendeten Hardware ab, nicht von der Chiffre.

Bei allen Anwendungen von BEAST-RK ist zu beachten, daß potentielle Angreifer nie die Blockchiffre „in der falschen Richtung" verwenden dürfen. Wenn beispielsweise ein Kunde einen Chiffretext wählt und an den Anbieter schickt, darf der Anbieter natürlich darauf hinweisen, daß der Klartext sinnlos ist – was eine sehr wahrscheinliche Folge ist, wenn man einen Chiffretext beliebig wählt und entschlüsselt –, keinesfalls aber den Klartext selber preisgeben. Dann nämlich wäre für den Kunden ein Chosen-Ciphtertext Angriff möglich. Sicher ist BEAST-RK aber nur gegen Chosen-Plaintext Angriffe.

[9] In der Komplexitätstheorie betrachtet man „Orakel" üblicherweise als abstrakte Hilfsmittel für Beweise. Hier sind die Orakel aber sogar physikalisch existent, und sie geraten buchstäblich in die Hände potentieller Angreifer.

Wie bereits erwähnt, stammt die Idee, eine Verschlüsselungsoperation zwischen einem schnellen Host und einer zwar langsamen, aber vertrauenswürdigen Smartcard aufzuteilen, von Blaze [8]. Dessen *"Remotely Keyed Encryption Protocol" (RKEP)* ist jedoch nicht sicher gegen Chosen-Plaintext Angriffe: Die mit Hilfe des Protokolls realisierbaren Blockchiffren kann man effizient von Zufallspermutationen unterscheiden [35]. Blaze selbst weist in Abschnitt 3.1 von [8] auf entsprechende kryptographische Schwächen des RKEP hin, ist aber der Meinung, *"neither of these attacks is likely to pose a serious threat to most practical applications."* Selbst bezüglich eines entsprechend schwachen Sicherheitsbegriffs fehlt ein Sicherheitsbeweis für das RKEP. Blaze selbst weist darauf hin, daß es andere, bisher unentdeckte Angriffe auf das RKEP geben könnte.

4.11 Luby-Rackoff Chiffren und zweiseitige Angriffe

Bei einem *zweiseitigen Angriff* (vgl. Abschnitt 2.2) auf die Permutation $g : \{0,1\}^{l+r} \longrightarrow \{0,1\}^{l+r}$ können Angreifer q Orakelfragen $Q_1, \ldots, Q_q \in \{0,1\}^{l+r}$ stellen, jeweils entweder an ein g-Orakel oder an ein g^{-1}-Orakel. Sie erhalten jedesmal die entsprechende Antwort $g(Q_i)$ bzw. $g^{-1}(Q_i)$. Die Angreifer sollen entscheiden, ob die Permutation g zufällig gewählt wurde oder nicht. Schon Luby und Rackoff [29] beschrieben den folgenden zweiseitigen Angriff auf die Permutation $\psi(f_1, f_2, f_3)$, (vgl. Abb. 4.1):

1. Wähle einen beliebigen Klartext $(L, R) \in \{0,1\}^l \times \{0,1\}^r$ als Frage an das g-Orakel.
2. Wähle $L' \in \{0,1\}^l$ beliebig und (L', R) als Frage an das g-Orakel.
3. Die Antwort auf die erste Frage bezeichnen wir mit (U, T), die auf die zweite mit (U', T'). Als Frage an das g^{-1}-Orakel berechne $(L \oplus L' \oplus U', T')$. Die Antwort bezeichnen wir mit (L'', R'').
4. Dann ist die Gleichung
$$R \oplus R'' = T \oplus T'$$
erfüllt – eine Beziehung, die bei einer Zufallspermutation nur mit der Wahrscheinlichkeit $1/2^r$ gelten würde.

Damit ist nachgewiesen, daß alle drei-Runden Luby-Rackoff Chiffren (wie u.A. BEAR, LION und BEAST) unsicher sind gegen zweiseitige Angriffe. Luby und Rackoff bewiesen auch, daß die vier-Runden Feistel-Chiffre

$$\psi(f_1, f_2, f_3, f_4)(L, R) = (U, V) \quad \text{mit} \quad (U, T) = \psi(f_1, f_2, f_3) \quad \text{und} \quad V = T \oplus f_4(U)$$

sicher gegen zweiseitige Angriffe ist, wenn $f_1, \ldots, f_4$ Zufallsfunktionen sind.

Luby und Rackoff beschränkten sich auf balancierte Chiffren. Unser Vorgehen, das Resultat auf unbalancierte Chiffren zu verallgemeinern, läßt sich auch für Lubys und Rackoffs Satz bezüglich $\psi(f_1, f_2, f_3, f_4)$ übertragen. Wir gehen in diesem Abschnitt einen Schritt

weiter und übertragen auch die Abkürzung in der dritten Runde (vgl. Satz 4.5) auf die letzte, d.h. vierte Runde von $\psi(f_1, f_2, f_3, f_4)$.

Die Permutation $\psi'(f_1, f_2, f_3, f_4)$, siehe Abbildung 4.4, hängt von den Funktionen

$$\begin{array}{lrcl} f_1, f_3: & \{0,1\}^r & \longrightarrow & \{0,1\}^l, \\ f_2: & \{0,1\}^l & \longrightarrow & \{0,1\}^r \quad \text{und} \\ f_4: & \{0,1\}^l & \longrightarrow & \{0,1\}^l \end{array}$$

ab. Es bezeichnen $S \in \{0,1\}^l$ und $T \in \{0,1\}^r$ Zwischenwerte, die bei der Berechnung von $\psi'(f_1, f_2, f_3, f_4)(L, R)$ auftreten. Es wird T aufgespalten in $T = T^* || T^{**}$, dabei ist $T^* \in \{0,1\}^l$, $T^{**} \in \{0,1\}^{r-l}$, und „||" bezeichnet die Konkatenation von Bit-Strings. Die Permutation$\psi'(f_1, f_2, f_3, f_4)(L, R) = (X, Y)$ ist definiert durch

$$\begin{array}{lcl} S = L \oplus f_1(R), & & \\ T = R \oplus f_2(S) & = & T^* \,||\, T^{**}, \\ X = S \oplus f_3(T) & & \quad \text{und} \\ Y = (T^* \oplus f_4(X)) \,||\, T^{**}. & & \end{array}$$

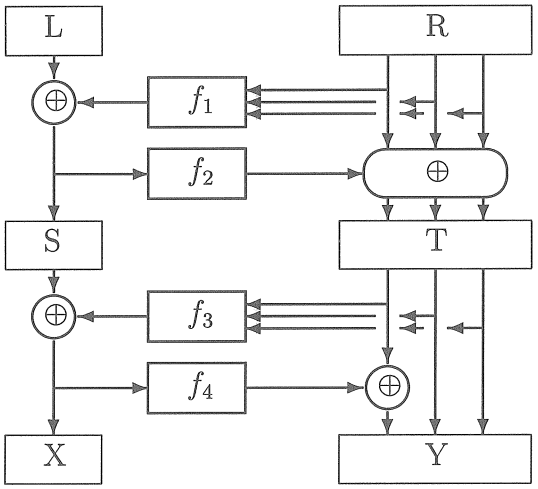

Abbildung 4.4: Permutation $\psi'(f_1, f_2, f_3, f_4)(L, R) = (X, Y)$

Satz 4.9
Sei $g : \{0,1\}^{r+l} \longrightarrow \{0,1\}^{r+l}$ *entweder als Zufallspermutation gewählt oder von der Form* $g = \psi'(f_1, f_2, f_3, f_4)$ *mit Zufallsfunktionen* $f_1, f_3 : \{0,1\}^r \longrightarrow \{0,1\}^l$, $f_2 : \{0,1\}^l \longrightarrow \{0,1\}^r$ *und* $f_4 : \{0,1\}^l \longrightarrow \{0,1\}^l$.

Der Algorithmus A habe Zugriff auf ein g-Orakel und ein g^{-1}*-Orakel. Die Ausgabemenge von A ist* $\{0,1\}$. *Mit* P_{rand} *(bzw.* P_{perm}*) bezeichnen wir die bedingte Wahrscheinlichkeit, daß A eine 1 ausgibt, wenn g als Zufallspermutation gewählt wurde (bzw. wenn Zufallsfunktionen* f_1, f_2, f_3, f_4 *gewählt wurden und* $g = \psi'(f_1, f_2, f_3, f_4)$ *ist).*

Wenn A insgesamt maximal q Orakel-Fragen stellt und die korrespondierenden Antworten erhält, gilt:

$$|P_{rand} - P_{perm}| < \frac{q^2}{2^n} * \frac{3}{2}.$$

Beweis: Die Liste der bei der Ausführung von A gestellten Orakel-Fragen und der gegebenen Antworten bezeichnen wir mit $((L_1, R_1), (X_i, X_1)), \ldots, ((L_q, R_q), (X_q, X_q))$. Dies ist unabhängig davon, ob (L_i, R_i) eine Frage an das g-Orakel und $(X_i, X_i) = g(L_i, R_i)$ die zugehörige Antwort ist oder (X_i, X_i) eine Frage an das g^{-1}-Orakel und $(L_i, R_i) = g^{-1}(X_i, X_i)$ die Antwort.

Sei $g = \psi'(f_1, f_2, f_3, f_4)$. Wenn g sich für die von A gewählten Fragen wie eine Zufallspermutation über $\{0,1\}^{l+r}$ verhält, ist g für A nicht von einer Zufallspermutation zu unterscheiden. Es bezeichne (S_i, T_i) die bei der Berechnung von $g(L_i, R_i)$ bzw. $g^{-1}(X_i, X_i)$ auftretenden Zwischenwerte.

Es hilft A nicht, eine Frage an ein Orakel zu stellen, wenn A die Antwort schon kennt. Deshalb können wir o. B. d. A. für alle $i, j \in \{1, \ldots, q\}$ mit $i \neq j$ die Ungleichung $(L_i, R_i) \neq (L_j, R_j)$ voraussetzen. Drei dazu äquivalente Ungleichungen sind $(X_i, X_i) \neq (X_j, X_j)$, $(S_i, T_i) \neq (S_j, T_j)$ und $(S_i \neq S_j) \vee (T_i \neq T_j)$.

Der Algorithmus A habe bisher $q-1$ Orakelfragen gestellt. Dann kennt A die Liste $((L_1, R_1), (X_i, X_1)), \ldots, ((L_{q-1}, R_{q-1}), (X_{q-1}, X_{q-1}))$. A kann die q-te Orakelfrage entweder an das g-Orakel oder an das g^{-1}-Orakel stellen.

Fall I A wählt $(L_q, R_q) \in \{0,1\}^l \times \{0,1\}^r$ als Frage an das g-Orakel. Für jedes $j \in \{1, \ldots, q-1\}$ gelte $(L_q \neq L_j) \vee (R_q \neq R_j)$.

Wenn $R_q = R_j$ ist, dann folgt $f_1(R_q) = f_1(R_j)$ und $L_q \neq L_j$, also $S_q \neq S_j$. Wenn $R_q \neq R_j$ ist, dann ist $(f_1(R_q), f_1(R_j)) \in_R (\{0,1\}^l)^2$. D.h., $f_1(R_q)$ und $f_1(R_q)$ sind zwei voneinander unabhängige Zufallswerte[10] aus $\{0,1\}^l$. Dann gilt auch $(S_q, S_j) \in_R (\{0,1\}^l)^2$. Für jede Frage (L_q, R_q) und für jedes $j \in \{1, \ldots, q-1\}$ ist also die Gleichung $S_q = S_j$ höchstens mit der Wahrscheinlichkeit $1/2^l$ erfüllt.

Aus $S_q \neq S_j$ folgt sogar $(T_q, T_j) \in_R (\{0,1\}^r)^2$, d.h., sogar T_q und T_j sind voneinander unabhängige Zufallswerte aus $\{0,1\}^r$. Die bedingte Wahrscheinlichkeit, daß $T_q \neq T_j$ gilt, wenn $S_q \neq S_j$ erfüllt ist, beträgt genau $1/2^l$. Für jede Frage (L_q, R_q) und für jedes $j \in \{1, \ldots, q-1\}$ ist damit die Gleichung $T_q = T_j$ höchstens mit der Wahrscheinlichkeit $2/2^l$ erfüllt.

Fall II A wählt $(X_q, Y_q) \in \{0,1\}^l \times \{0,1\}^r$ als Frage an das g^{-1}-Orakel. Für jedes $j \in \{1, \ldots, q-1\}$ gelte $(X_q \neq X_j) \vee (Y_q \neq Y_j)$.

Wenn $X_q = X_j$ ist, dann folgt $f_4(X_q) = f_4(X_j)$ und $Y_q \neq Y_j$, also $T_q \neq T_j$. Wenn $X_q \neq X_j$ ist, dann ist $(f_4(X_q), f_4(X_j)) \in_R (\{0,1\}^l)^2$, also mit der Wahrscheinlichkeit $1/2^l$ $T_q^* = T_j^*$.

[10] Es sei daran erinnert, daß „$x \in_R M$“ ein gemäß Gleichverteilung gewähltes Element aus M bezeichnet.

Die bedingte Wahrscheinlichkeit, daß $S_q \neq S_j$ gilt, wenn $T_q \neq T_j$ erfüllt ist, beträgt genau $1/2^l$; denn aus $T_q \neq T_j$ folgt $(S_q, S_j) \in_{\mathrm{R}} (\{0,1\}^l)^2$. Für jede Frage (X_q, Y_q) und für jedes $j \in \{1, \ldots, q-1\}$ ist die Gleichung $S_q = S_j$ deshalb höchstens mit der Wahrscheinlichkeit $2/2^l$ erfüllt.

Bei insgesamt q Orakelfragen sind also mit mindestens der Wahrscheinlichkeit

$$1 - \frac{2}{2^l}\frac{q^2}{2} = 1 - \frac{q^2}{2^l}$$

alle Werte $S_1, \ldots, S_q$ paarweise verschieden. Zurück zu der Situation nach $q-1$ gestellten Orakelfragen:

- Wenn im **Fall I** $T_q \neq T_j$ erfüllt ist, sind $f_3(T_q)$ und $f_3(T_j)$ für jedes $j \in \{1, \ldots, q-1\}$ voneinander unabhängige Zufallswerte aus $\{0,1\}^l$, ebenso X_q und X_j. Mit der Wahrscheinlichkeit $1/2^l$ gilt $X_q \neq X_j$, und in diesem Fall sind auch Y_q und Y_j voneinander unabhängige Zufallswerte aus $\{0,1\}^r$.
- Im **Fall II** kann man analog argumentieren: Wenn $S_q \neq S_j$ gilt, dann ist $(R_q, R_j) \in_{\mathrm{R}} (\{0,1\}^r)^2$. Dann gilt höchstens mit der Wahrscheinlichkeit $1/2^l$ die Gleichung $R_q = R_j$, und, wenn $R_q \neq R_j$, dann ist auch $(L_q, L_j) \in_{\mathrm{R}} (\{0,1\}^l)^2$.

Damit sind mindestens mit der Wahrscheinlichkeit

$$\frac{q^2}{2^n} * \frac{3}{2}$$

alle q Antworten auf As Orakel-Fragen voneinander unabhängige, gemäß Gleichverteilung gewählte Zufallswerte aus $\{0,1\}^{r+l}$. □

Wenn die Blöcke groß sind, kann man den Aufwand für die vierte Runde vernachlässigen. Die Geschwindigkeit von $\psi'(f_1, f_2, f_3, f_4)$ wird durch die beiden Hashfunktionen f_1 und f_3 sowie den pseudozufälligen Bitgenerator f_2 bestimmt. Analog zu den Abschnitten 4.8 bzw. 4.9 kann man die Hashfunktion SHA-1 zur Realisierung der Funktionen f_1, f_3 und f_4 einsetzen, die Flußchiffre SEAL zur Realisierung von f_2. Eine derartige Realisierung von $\psi'(f_1, f_2, f_3, f_4)$ ist bei großen Blöcken ebensoschnell wie BEAR, bei einer Blockgröße von 1 MBit auf der mehrfach angesprochenen "Sandpiper" also etwa 13,6 MBit/sec. Während BEAR nur sicher gegen einseitige Angriffe ist, ist $\psi'(f_1, f_2, f_3, f_4)$ sogar sicher gegen zweiseitige Angriffe.

4.12 Weitere Forschung

Der Entwurf von Blockchiffren, aufbauend auf einfachen Konstruktionsprinzipien wie dem von Luby und Rackoff, ist ein intensiv bearbeitetes Teilgebiet der Kryptographie. Daraus ergibt sich, daß in kurzen Zeitabständen immer wieder neue Resultate publiziert werden. Es sei an dieser Stelle auf zwei jüngst zur Publikation akzeptierte Papiere hingewiesen, die in engem Bezug zu den in diesem Kapitel geleisteten Arbeiten stehen:

- Abschnitt 4.10 dieser Arbeit behandelt ein System für "Remote-Key"-Verschlüsselung, das beweisbar sicher ist gegen Chosen-Plaintext Angriffe. Lucks [35] beschreibt ein ähnliches System, das sogar vor zweiseitigen Angriffen schützt. Er motiviert dies u.A. durch den Hinweis auf bestimmte Schwächen des RKEP von Blaze.

  Ebenfalls in [35] beschreibt Lucks die neun-Runden Feistel-Chiffre GRIFFIN und beweist ihre Sicherheit gegen zweiseitige Angriffe. Für große Blöcke entspricht ihre Effizienz der von LION. Dies ergänzt das Resultat aus Abschnitt 4.11 dieser Arbeit. Die dort behandelte vier-Runden Luby-Rackoff Chiffre schützt vor zweiseitigen Angriffen und ist vergleichbar effizient wie BEAR. BEAR und LION bieten keinen Schutz vor zweiseitigen Angriffen.

- Praktikable Luby-Rackoff Chiffren bestehen aus drei bzw. vier Runden. In jeder Runde wird eine pseudozufällige Funktion angewandt. In Abschnitt 4.2 dieser Arbeit wird eine Methode betrachtet, die Berechnung einer pseudozufälligen Funktion in der ersten Runde zu vermeiden und stattdessen eine Differenz-Konzentrator-Funktion zu berechnen. Dabei kann es sich um kombinatorische, nicht pseudozufällige Konstruktion handeln. Betrachtet werden jedoch nur Chiffren, die sicher gegen Chosen-Plaintext Angriffe sind.

  Naor und Reingold [43] betrachten auch Luby-Rackoff Chiffren, die sicher gegen zweiseitige Angriffe sind. Diese bestehen aus vier Runden. Die Autoren zeigen, wie man ohne Verlust der Sicherheit die pseudozufälligen Funktionen in der ersten *und* der vierten Runde durch kombinatorische Konstuktionen ersetzen kann.

  Bemerkenswert sind auch ihre neuartigen verallgemeinerten Luby-Rackoff Chiffren für besonders kleine und besonders große Blöcke, siehe Kapitel 5 und 6 ihrer Arbeit.

Literaturverzeichnis

[1] R. ANDERSON, "The Classification of Hash Functions", in: Fourth IMA conference on cryptography and coding, 83–93,1993.

[2] R. ANDERSON, "On Fibonacci Keystream Generators", in: Fast Software Encryption (1994), Springer LNCS 1008, 346–352.

[3] R. ANDERSON, E. BIHAM, "Tiger: A Fast New Hash Function", in: Fast Software Encryption (1996), Springer LNCS 1039, 89–97.

[4] R. ANDERSON, E. BIHAM, "Two Practical and Provably Secure Block Ciphers: BEAR and LION", in: Fast Software Encryption (1996), Springer LNCS 1039, 113–120.

[5] M. BELLARE, R. CANETTI, H. KRAWCZYK, "Pseudorandom Funtions Revisited: The Cascade Construction and its Concrete Security", Manuscript, gekürzte Fassung in 37th FoCS, IEEE 1996.

[6] M. BELLARE, P. ROGAWAY, "Optimal Asymmetric Encryption—How to Encrypt with RSA", in: Eurocrypt '94, Springer LNCS 950, 92–111.

[7] M. BELLARE, P. ROGAWAY, "The *Exact* Security of Digital Signatures—How to Sign with *RSA* and *Rabin*", in: Eurocrypt '96, Springer LNCS 1070, 399–416.

[8] M. BLAZE, "High-Bandwidth Encryption with Low-Bandwidth Smartcards", in: Fast Software Encryption (1996), Springer LNCS 1039, 33–40.

[9] M. BLAZE, B. SCHNEIER, "The MacGuffin Block Cipher Algorithm", in: Fast Software Encryption (1994), Springer LNCS 1008, 96–110.

[10] P. CAMION, J. PATARIN, "The Knapsack Hash Function proposed at Crypto '89 can be broken", in: Eurocrypt '91, Springer LNCS 547, 39–53.

[11] J. DAEMEN, "Limitations of the Even-Mansour Construction", in Asiacrypt '91, Springer LNCS 495–498.

[12] I. B. DAMGÅRD, "A Design Principle for Hash Functions", in: Crypto '89, Springer LNCS 435, 416–427.

[13] H. DOBBERTIN, "Cryptanalysis of MD4", in: Fast Software Encryption (1996), Springer LNCS 1039, 53–69.

[14] H. DOBBERTIN, A. BOSSELAERS, B PRENEEL, "RIPEMD-160, A Strengthened Version of RIPE-MD", in: Fast Software Encryption (1996), Springer LNCS 1039, 71–82.

[15] J.-B. FISCHER, J. STERN, "An Efficient Pseudo-Random Generator Provably as Secure as Syndrome Decoding", in: Eurocrypt '96, Springer LNCS 1070 245–255.

[16] M. GAREY, D. JOHNSON *Computers and Intractability*, Freeman, New York, 1979.

[17] M. GIRAULT, J. STERN, "On the length of cryptographic hash-values used in identification schemes", in: Crypto '94, Springer LNCS 839, 202-215.

[18] O. GOLDREICH, S. GOLDWASSER, S. MICALI, "How to Construct Random Functions", in: 21st ACM SToC, 25–32, 1989.

[19] P. GUTMANN, Documentation zu SFS Version 1.20 – Teile `SFS6.DOC` und `SFS7.DOC`, URL: `http://www.cs.auckland.ac.nz/~pgut01/sfs.html`, 1995.

[20] J. HÅSTAD, "Pseudo-Random Generators under Uniform Assumptions", in: 22nd ACM SToC, 395–404, 1990.

[21] A. HOFFMAN, P. WOLFE, "History", in [28], 1–15.

[22] R. IMPAGLIAZZO, L. LEVIN, M. LUBY, "Pseudo-random generation from one-way functions", in: 21st ACM SToC, 12–24, 1989.

[23] R. IMPAGLIAZZO, M. NAOR, "Efficient Cryptographic Schemes Provably as Secure as Subset Sum", in: FOCS '89, 236–241.

[24] D. JOHNSON, C. PAPADIMITRIOU, "Computational complexity", in: [28], 37–85.

[25] M. JÜNGER, G. REINELT, S. THIENEL, "Provably good solutions for the traveling salesman problem", in: Zeitschrift für Operations Research 40 (1994), 183–217.

[26] D. KNUTH, *The Art of computer programming*, Vol. 2, Addison Wesley, 1981.

[27] X. LAI, *On the Design and Security of Block Ciphers* (Dissertation), ETH Series in Information Processing (1), Hartung-Gorre Verlag, Konstanz, 1992.

[28] E. L. LAWLER, J. K. LENSTRA, A. H. G. RINNOY KAN, D. B. SHMOYS (eds.), *The Traveling Salesman Problem*, Wiley, 1985.

[29] M. LUBY, C. RACKOFF, "How to construct pseudorandom permutations from pseudorandom functions", in: SIAM J. Computing, 1988, Vol. 17, No. 2, 373–386.

[30] G. LÜTGE, „Räuber im Netz", in: Die Zeit, Nr 35, 23.08.96, S. 27.

[31] S. LUCKS, "How to Exploit the Intractability of Exact TSP for Cryptography", in: Fast Software Encryption (1994), Springer LNCS 1008, 298–304.

[32] S. LUCKS, "How Traveling Salespersons Prove their Identity", Fifth IMA Conference on Cryptography and Coding, Springer LNCS 1025, 142–149.

[33] S. LUCKS, "Faster Luby-Rackoff Ciphers", in: Fast Software Encryption (1996), Springer LNCS 1039, 189–203.

[34] S. LUCKS, "BEAST: A Fast Block Cipher for Arbitrary Blocksizes", in: P. Horster (Hrsg.), IFIP Conference on Communications and Multimedia Security (1996), Chapman & Hall, 144–153.

[35] S. LUCKS, "On the Security of Remotely Keyed Encryption", in: Fast Software Encryption (1997), Springer LNCS, erscheint demnächst.

[36] U. MAURER, "A Simplified and Generalized Treatment of Luby-Rackoff Pseudorandom Permutation Generators", in: Eurocrypt '92, Springer LNCS 658, 239–255.

[37] R. C. MERKLE, M. HELLMAN, "Hiding information and Signature in Trapdoor Knapsack", in: IEEE Trans. on Inf. Theory 24 (1978), 525–530.

[38] R. C. MERKLE, "A Certified Digital Signature (That Antique Paper from 1979)", in: Crypto '89, Springer LNCS 435, 218–238.

[39] R. MOTWANI, P. RAGHAVAN, *Randomized Algorithms*, Cambridge University Press, 1995.

[40] National Institute of Standards and Technology (NIST), *Secure Hash Standard*, FIPS 186, US Department of Commerce, January 1992.

[41] NIST, *Announcing the DATA ENCRYPTION STANDARD (DES)* FIPS 46-2 (revised standard), US Department of Commerce, December 1993.

[42] NIST, *Secure Hash Standard*, FIPS 186-1 (revised standard), US Department of Commerce, May 1994.

[43] M. NAOR, O. REINGOLD "On the Construction of Pseudo-Random Permutations: Luby-Rackoff Revisited" (Preliminary Version), akzeptiert für STOC 1997.

[44] M. NAOR, M. YUNG, "Universal One Way Hash Functions and Their Cryptographic Applications", in: 21st ACM SToC, 33–43 (1989).

[45] R. NEEDHAM, D. WHEELER, "Tea, a Tiny Encryption Algorithm", in: Fast Software Encryption (1994), Springer LNCS 1008, 363–366.

[46] M. PADBERG, G. RINALDI, "A Branch–and–Cut Algorithm for the Resolution of Large-Scale Symmetric Traveling Salesman Problems", in: Siam Review 33, No. 1 (1991), 60–100.

[47] J. PATARIN, P. CHAUVAUD, "Improved Algorithms for the Permuted Kernel Problem", in: Crypto '93, Springer LNCS 773, 391–402.

[48] J. PIEPRZYK, B. SADEGHIYAN, *Design of Hashing Algorithms*, Springer LNCS 756, 1993.

[49] D. POINTCHEVAL, "A New Identification Scheme Based on the Perceptrons Problem", in: Eurocrypt '95, Springer LNCS 921, 319–328.

[50] B. PRENEEL, *Analysis and Design of Cryptographic Hash Function*, Ph. D. Thesis, Katholieke Universiteit Leuven (Belgien), 1993.

[51] M. RABIN, *Digital Signatures and Public Key functions as Intractable as Factorization*, MIT Laboratory for Computer Science Report TR-212, 1979.

[52] M. RABIN, *Fingerprinting by Random Polynomials*, Harvard Univ. Tech. Rep. TR-15-81, 1981.

[53] *Race Integrity Primitives Evaluation (RIPE): final report*, RACE 1040, 1993.

[54] G. REINELT, TSPLIB 1.1, URL: `ftp://softlib.cs.rice.edu/pub/tsplib`, Dezember 1990.

[55] M. ROE, "Performance of Block Ciphers and Hash Functions — One Year Later", in: Fast Software Encryption (1994), Springer LNCS 1008, 359–362.

[56] P. ROGAWAY, D. COPPERSMITH, "A Software-Optimized Encryption Algorithm", in: Fast Software Encryption (1993), Springer LNCS 809, 1994, 56–63.

[57] B. SCHNEIER, "Description of a new variable-length key 64-bit block cipher (Blowfish)", in: Fast Software Encryption (1993), Springer LNCS 809, 1994, 191–204.

[58] B. SCHNEIER, *Applied Cryptography*, Wiley, 1995.

[59] B. SCHNEIER, "Differential Cryptanalysis", in: Dr. Dobb's Journal No. 243, January 1996, 42–48.

[60] B. SCHNEIER, J. KELSEY, "Unbalanced Feistel Networks and Block Cipher Design", in: Fast Software Encryption (1996), Springer LNCS 1039, 121–144.

[61] C. P. SCHNORR, M. EUCHNER, "Lattice Basis Reduction: Improved Practical Algorithms and Solving Subset Sum Problems", in: FCT '91, 68–85.

[62] C. P. SCHNORR, H. H. HÖRNER, "Attacking the Chor-Rivest Cryptosystem by Improved Lattice Reduction", in: Eurocrypt '95, Springer LNCS 921, 1–12.

[63] A. SHAMIR, "An Identification Scheme based on Permuted Kernels", in: Crypto '89, Springer LNCS 435, 606–609.

[64] C. E. SHANNON, "Communication theory of secrecy systems", in: Bell Systems Technical Journal 27 (1949), 656–715.

[65] J. STERN, "A new identification scheme based on syndrome decoding", in: Crypto '93, Springer LNCS 773, 13–20.

[66] J. STERN, "Designing Identification Schemes with Keys of Short Size", in: Crypto '94, Springer LNCS 839, 164–173.

[67] D. R. Stinson, *Cryptography: Theory and Practice*, CRC Press, 1995.

[68] S. THIENEL, *ABACUS A Branch-And-CUt System*, Dissertation, Universität zu Köln, 1995.

[69] I. WEGENER, *Effiziente Algorithmen für grundlegende Funktionen*, Teubner, 1989.

[70] WEI DAI, *Algorithm benchmarks*,
URL: `http://www.eskimo.com/~weidai/benchmarks.txt`, 1996.

[71] M. WIENER, *Efficient DES Key Search*, Manuscript (auf der Rump Session der Crypto '93 vorgestellt),
URL: `http://www.cert-kr.or.kr/doc/des_key_search.ps.asc.html`, 1993.

Lebenslauf

Stefan Lucks	geboren am 10.05.1965 in Lüdenscheid deutsche Staatsangehörigkeit

Schulbesuch und Bundeswehr

1971 – 1975	Adolf-Kolping-Grundschule, Lüdenscheid
1975 – 1984	Bergstadt-Gymnasium, Lüdenscheid
1984	Abitur
Juli 1984 – September 1985	Wehrdienst in Iserlohn und Altenstadt

Studium und Beruf

WS 1985/86 – WS 1992/93	Studium der Informatik an der Universität Dortmund
1988	Vordiplom in Informatik, Nebenfach BWL
1992	Diplom in Informatik, Nebenfach Mathematik
1993–1997	Wissenschaftlicher Mitarbeiter am Institut für Numerische und Angewandte Mathematik, Georg-August-Universität zu Göttingen